Lehrstuhl für Informatik
der Technischen Universität
München

The Metamodelling Language *M2L*

An Approach for Seamless Language Engineering and Formal Metamodelling

Stefano M. Merenda

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Lehrstuhl für Informatik
der Technischen Universität
München

The Metamodelling Language *M2L*

An Approach for Seamless Language Engineering and Formal Metamodelling

Stefano M. Merenda

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Hans-Joachim Bungartz
1. Prüfer der Dissertation:	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Prüfer der Dissertation:	Univ.-Prof. Dr. Mark Minas Universität der Bundeswehr München

Die Dissertation wurde am 26.08.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.01.2012 angenommen.

Appropriateness is the only reason why we suffer about languages at all.
Otherwise we can use Turing machines for everything.

To my grandparents Ida and Paul,
and my son Lukas-Benedikt.

Abstract

The constantly increasing complexity of software demands increasingly higher abstractions of the systems to be developed in order for them to remain ascertainable for humans. On this occasion, the most divers modelling techniques in the field of systems engineering evolve for the different aspects of a system, which is in most cases a combination of hardware and software nowadays.

For this reason, a novel class of modelling languages is necessary, covering total system development from requirements engineering over system development to testing. Thus, such languages become very large and are called *mega-languages* in this work. In contrast to traditional languages, such mega-languages consist of many sub-languages which have to match in a sound way. In addition, many consistency conditions will arise within and between these sub-languages. Each requirement, for example, will have to be implemented by a part of the system developed. On the other hand in most situations, only an excerpt of the model will be necessary for e. g. analysis or synthesis purposes due to the size of the language. Thus, complex queries should be able to be requested to a model repository in order to be able to work with the essential excerpt of the model. This makes it even insufficient to store the models in conventional, textual-based repositories.

To overcome these problems, the very basic need is having a suitable metamodelling language including a corresponding tooling environment. Hereby the tight integration of metamodelling language and tooling environment builds up the crucial issue of successful implementation of mega-languages in the industry. In such a scenario the metamodelling language comprises not only the abstract syntax definition (which defines the structural relationships between the concepts of the modelling language) but also concrete syntax definition (such as textual, tabular or diagrammatical syntaxes), process definition, as well as semantics of the modelling language. Altogether, such a metamodel which consists of these four metamodelling aspects exactly defines the behaviour of the tooling environment: The abstract syntax says what can be modelled and what is the data-structure in behind; the concrete syntax specifies how the models are presented to the user and how the models can be edited; the process definition defines which process role is allowed to edit what part at what time; and finally the semantics says how the built models can e. g. be simulated or what target code should be generated. This vision in total we call *Seamless Language Engineering*.

Therefore, within the present work the metamodelling language *M2L* is developed. Hereby *M2* stands for *metamodel* and *L* stands for *language*. It is supposed to combine the clear structural semantics of existing formal approaches to appropriate solutions already used in industry in order to solve the above-mentioned issues.

The essential part of this work concentrates on providing a theoretical foundation for building metamodelling languages which fulfils the above-mentioned requirements. Hereby we focus on the first two aspects of metamodelling languages: abstract syntaxes and concrete syntaxes. In the field of the concrete syntaxes we again focus on *textual* concrete syntaxes. The major theoretical concepts and improvements are:

- The use of partially ordered multi-sets (pomsets) will allow for handling a mixture of ordered sets as well as sets including duplicates in a sound way. Pomsets also contribute significantly to the expressiveness of the query-language.
- Abstract and concrete syntax definition are strictly separated within a language specification. Nevertheless they are strongly related to each other. This also allows for an easy definition of multiple concrete syntaxes, all related by abstract syntax in a sound way.
- In analogy to a textual word in formal languages, a distinct definition of abstract words is introduced by model-graphs (M-graphs), which can be defined independently of a metamodel. This will form the basis for being able to define a metamodel as simply being a set of constraints on such M-graphs.
- In order to define such constraints in an appropriate way, *Edge Algebra* is introduced. Comprising the four core operators *navigation*, *closure*, *edge inverse*, and *selection*, it provides a powerful query-language on M-graphs.
- Based on Edge Algebra, a set of useful metamodelling concepts on a formal basis will additionally be introduced. In particular, these are *conditional properties*, *context-sensitive domains*, *local keys*, *namespaces*, and *instantiating properties*.

All these concepts will result in the overall metamodelling language *M2L*. It allows for a definition of both abstract and textual concrete syntaxes in a formal but also appropriate way and is sufficient for creating a suitable tooling environment: The open-source project *OOMEGA* [OOMEGA, 2010] provides a ready-to-use tooling environment based on *M2L*. For example, the abstract and textual concrete syntaxes for the modelling languages *FOCUS* [Broy and Stølen, 2001] and *COLA* [Kugele et al., 2007] are modelled in *M2L*. Based on these language specifications OOMEGA provides sufficient IDEs in the form of Eclipse plug-ins [Eclipse, 2010b] including a database back-end.

Acknowledgement / Danksagung

Since an acknowledgement is the most personal part in such a theoretical work, I have decided to write it in my mother tongue. However this is the only section all my relatives will read indeed – so German is even the better choice!

Eine Dissertation zu schreiben ist eine wirklich langwierige Angelegenheit – und zugegebener Maßen auch langwieriger als ich zunächst angenommen hatte. Auf der anderen Seite muss ich zugestehen, dass mir die Arbeit am Lehrstuhl, an den damit verbunden Projekten – insbesondere auch *BASE.XT* zusammen mit BMW – sowie auch die eigentliche Arbeit an meiner Dissertation mit der dazugehörigen Open-Source-Implementierung *OOMEGA* einfach zu viel Spaß gemacht haben, als dass es mich früher von der Universität weggezogen hätte. Und das verdanke ich Ihnen, Herr Broy! Sie haben mir immer den großzügigen Freiraum für meinen Dickschädel in meinem wissenschaftlichen Arbeiten gegeben, den sich so manch anderer nur wünschen kann – mit Sicherheit der Hauptgrund, warum ich die Zeit bei Ihnen am Lehrstuhl immer in so guter Erinnerung behalten werde. Und trotz der Tatsache, dass Ihr Terminkalender im Viertel-Stunden-Rhythmus getaktet ist, hatten Sie immer ein offenes Ohr für meine Anliegen. Na ja – Du, liebe Silke, hast das Deinige dazu beigetragen: "Oh nein, Stefano, schon wieder einen Termin?!?" – Und immer hast Du einen gefunden! Und wenn ich dann – nein, eigentlich war es immer Herr Broy – den Termin überzogen hatte, hast Du den Nachfolger immer erfolgreich beschwichtigt.

Es waren aber auch die Kollegen, die meinen Dissertationsalltag zu einer unvergesslichen Zeit gemacht haben! Zunächst gab es die obligatorischen Fachsimpeleien am Gang, in der Kaffeeküche und auch auf der Toilette (!). In diesem Zusammenhang möchte ich mich auch bei allen Kollegen aus dem BASE-XT-Team für die tolle Zusammenarbeit bedanken. Ohne Euch hätte das Projekt nur halb so viel Spaß gemacht.

Was die inhaltliche Arbeit meiner Dissertation angeht, waren die Diskussionen mit Dir, lieber Markus, die spannendsten wenn auch – oder gerade weil – die anstrengendsten. Darüber hinaus hast Du auch einen wesentlichen Beitrag zur Implementierung geleistet. Danke an dieser Stelle nochmals für Deine Unterstützung!

OOMEGA und damit auch meine Dissertation wäre heute nicht das, was es ist, wenn mein Bruder, Du, lieber Christian, nicht all das aus Überzeugung mit geschultert hätte. Es ist immer wieder toll, mit Dir fachlich zusammenzuarbeiten! Auch Dir, lieber Sevi, danke für Deine jahrelange und tatkräftige Unterstützung bei der Implementierung des Texteditors.

Aber auch die fachfremden Themen haben am Lehrstuhl große Freude bereitet: Ich weiß nicht, wie oft ich mit Dir, lieber Bernd, beim "Weißwurschten" war – was ich aber noch weiß, ist, dass ich erstens meine Heimarbeit am Freitag des öfteren wegen genau einem

Termin unterbrochen habe: dem morgendlichen “Weißwurschten” und zweitens, dass ich Dich von meiner Theorie überzeugen konnte, dass man mit einem Weißbier deutlich besser wissenschaftlich arbeiten kann!

Es gab aber natürlich auch “ernstere” Themen: Allen voran meine Verlobung! Niemand anderes als Ihr, liebe Doris, liebe Sabine, habt dafür gesorgt, dass ich endlich mal in die Pötte komme. Sie sehen, Herr Broy, auch die wirklich wichtigen Dinge im Leben werden bei Ihnen am Lehrstuhl diskutiert und auch tatsächlich *erfolgreich* bearbeitet. Dich, liebe Doris, habe ich neben solchen Sonderthemen auch als meine treue Zimmerkollegin schätzen und lieben gelernt. Denn es gab durchaus auch Momente beim Schreiben der Dissertation oder auch der Projektarbeit, die – sagen wir einmal – nervig gewesen sind. Und diese Themen konnte man wunderbar mit Dir diskutieren und danach ging’s mir auch gleich wieder besser. Und auch unsere unterschiedlichen Ansichten zu Licht an/aus, Heizung an/aus oder Musik an/aus haben wir – so denke und hoffe ich – wunderbar in den Griff bekommen.

Trotz all der tollen Erfahrungen kam der Wechsel in die Industrie doch noch vor dem Abschluss meiner Dissertation. Ein – wie bekannt ist – gefährliches Unterfangen für die Fertigstellung der Arbeit. Vielen Dank an dieser Stelle an BMW und allen Voran an Euch, lieber Stefan, lieber Heiko, dass Ihr mir immer den Rücken freigehalten habt, um meine Dissertation neben meiner Arbeit wie geplant fertig zu stellen.

In dieser finalen Phase, haben Sie, lieber Herr Minas, als Zweitgutachter nochmals wertvollen Input geliefert. Vielen Dank für Ihre Bereitschaft, das Zweitgutachten zu übernehmen und auch Ihr Interesse an meiner Arbeit einschließlich Ihrem detailliertem Feedback.

Zu guter letzt bleibt mir noch meine Familie ohne dessen Unterstützung – wie so oft – nichts geht. Ihr, liebe Hildegard, lieber Ulf, habt uns immer wieder tatkräftig im Haushalt unterstützt. Ohne Euren gezielten Eingriff wäre unser Garten sicherlich nur noch mit einem Buschmesser zu betreten. Euch, liebe Mama, lieber Papa, danke für Euer unermüdliches gutes Zureden. Wenn alle anderen die Hoffnung aufgegeben haben, habt Ihr die Fahne immer noch hochgehalten und ich konnte mich daran hoch ziehen.

Du, meine allerliebste Alice, hast die Arbeit mit Sicherheit am stärksten zu spüren bekommen. Ich glänzte zuhause durch ständige Abwesenheit – und wenn mal nicht körperlich, dann zumindest geistig! Zum Glück ist das Sprichwort “Löcher in die Wand starren” nicht wörtlich zu nehmen, ansonsten wäre unser Zuhause nur noch ein Schweizer Käse. Danke für Deine scheinbar nie endende Geduld! Hinzu kommt, dass Du noch während Deiner Schwangerschaft die in Deutsch vorliegende Passagen für mich ins Englische übersetzt und dann das komplette sprachliche Korrekturlesen übernommen hast. Du hattest damit die undankbare Aufgabe, knapp 300 Seiten auf Punkt und Komma zu überprüfen. Danke auch für diese tatkräftige und zugegebenermaßen langwierige Unterstützung!

Nun bleibt noch der jüngste im Bunde: Mein Sohn Lukas-Benedikt. Du, liebster Lukas, hast mir mit Deiner Geburt am 22. April 2011, den finalen Schubs gegeben, dass ich nun endlich meine Doktorarbeit zum Abschluss bringe – und das habe ich genau vier Monate später am 22. August 2011 nun auch endlich getan. Danke dafür – und danke dafür, dass es Dich gibt!

München, 22. August 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The classic compiler construction with its programming languages was able to concentrate entirely on programmers. In contrast to that, when introducing mega-modelling languages, the potential user group of modelling languages scales up dramatically: Different levels of abstraction are provided by such mega-modelling languages in order inspect the system to develop by different process roles from different point of views: Some examples are requirements view, system architectural view, implementation view, testing view, or even a project management view.

The goal is a novel class of modelling languages which finally aims at all roles of a model-based development process involved. This dramatic expansion of the potential user groups of modelling languages led to new requirements in the field of language engineering, which shall be represented in the following. Note that parts of the two introductive chapters Chapter 1, *Introduction*, p. 1 and Chapter 2, *Metamodels and Seamless Language Engineering*, p. 21 are published in [Broy et al., 2010].

Contents

1.1. Current Situation

Nowadays, model-based development is more or less consequently adopted in the practical development of automotive and avionic systems. The pervasive use of models allows engineers to abstract from implementation details, thus raising the level of abstraction at which the systems are developed. As a consequence, model-based development promises to increase both productivity and quality of software development for embedded systems.

The following is cited from [Broy et al., 2010] page 526-528 including some minor changes in order to adopt it to this work:

> *However, model-based development approaches often fall short due to the lack of integration at both the conceptual and tooling level. Even if artefacts are modelled explicitly, they are based on separate and unrelated modelling theories (if foundations are given at all), which in turn renders the transition from one artefact to another non distinctive and error-prone within the development process. Current tools usually focus on particular development steps and support single modelling paradigms (see Figure 1.1). Although many of these tools show good results in their limited domain, many models have to be constructed during the development of a system beginning with the initial requirements up to a running implementation in hard- and software. In practice, several isolated tools are necessary to construct these models, and the transition between them is often far from being distinct. Consequently, engineers adopt ad-hoc integration solutions that are far from being disciplined engineering. Both theories (whenever they are applied) and tools do not fit to each other, which complicates the reuse of models in different phases. Instead of refining and transforming the already existing models, they are often rebuilt from scratch which involves a lot of effort and loss of information. The overall information about the developed product is only implicitly available in the engineers' minds.*

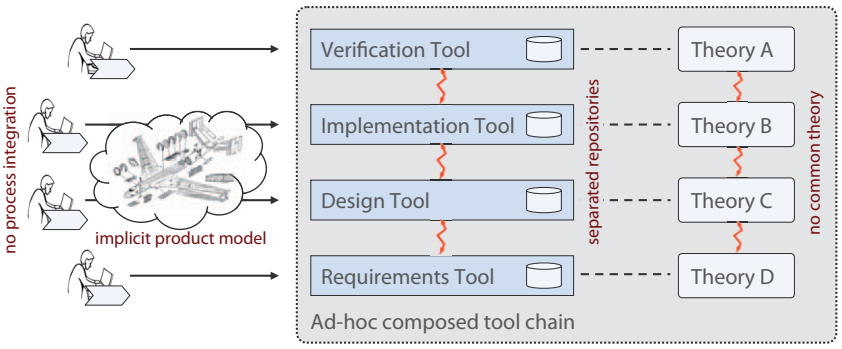


Figure 1.1.: Today's engineering environments: ad-hoc composed tool chains.

> *The real benefits of the models take effect if they are used throughout the entire development process in a seamless way. Requirements, for instance, are the inputs for an initial system design and for test case generation. This workflow requires a deep integration of the requirements, the system design, and the tests in an integrated product model. Such integration can only be implemented in a model engineering environment supporting the reuse of the information that is captured within the models. To achieve the vision of seamless model-based development (illustrated in Figure 1.2), we need the following three integral parts: 1) a comprehensive modelling theory serving as a semantic domain for the formal definition of the models, 2) an integrated architectural model describing the detailed structure of the product (product model) as well as the process for developing said (process model), and 3) an integrated model engineering environment guaranteeing a seamless tool support for authoring and analysing the product model according to the process defined in the process model. Instead of working with isolated models, engineers access a common model repository which explicitly stores the overall product model via dedicated views. All required views are formally defined and based on one comprehensive modelling theory which enables*

the construction and distinct semantic interpretation of the product architecture. Compliance with the process model is ensured by a common workflow engine.

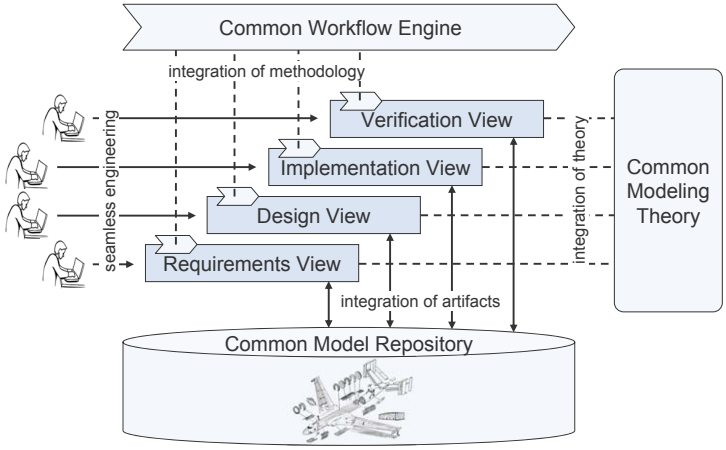
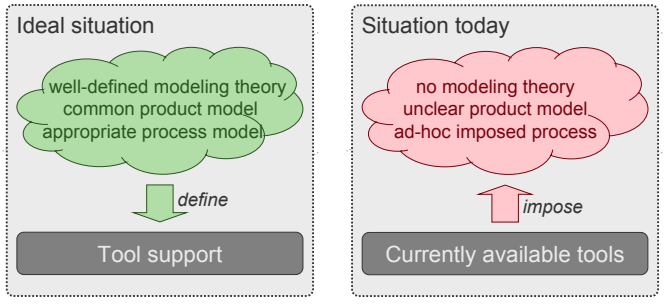


Figure 1.2.: Vision of an integrated model engineering environment.

One of today's major impediments to the advent of seamless model-based development in the industry is the lack of tools. Ideally, we should be able to define *the requirements for tooling (see Figure 1.3(a)) based on modelling theories and a common product model. In reality, however, a small set of tools (most of the time off-the-shelf) has to be used for a particular project. Thereby, these tools* impose *the modelling aspects that are treated and consequently the product model that is to be built (see Figure 1.3(b)). Due to the high costs with regard to developing and maintaining tools, the development of in-house tools that are specific enough and tailored to a certain project is not an option. As a consequence, engineers need to adapt their process to commercially available tools and often they have to alter their desired product design in a way imposed by the tools at hand. The top-down dependency between the modelling theory and its implementation in tools (as promoted by the model-based development itself) is in reality inverted by the already available tools that dictate the modelling technique that has to be followed in a project.*



(a) wanted (b) unwanted

Figure 1.3.: The tyranny of current tools.

End of citation.

Modelling languages must be made as comprehensible as possible as many process roles – in contrast to programmers – are only temporarily confronted with the modelling language and training periods should be kept as short as possible. Graphic ways of representation, such as utilised by UML for example, are therefore preferred to formal languages. Nowadays, the syntax of such diagrammatic languages is mainly only defined in a semi-formal way. It has, however, turned out to be practical to split syntax definition into two components.

Abstract syntax is to describe the elementary structure of the models, independent of the concrete notation. That is exactly what the so-called meta-modelling is used for: Current metamodels normally describe right the abstract syntax of modelling languages. At the same time, abstract syntaxes are very well suited for building a taxonomy of the domain to be modelled. Thus, correlations between domain and modelling language can be clearly identified from the very beginning. Concrete syntax forms the second part of syntax definition: in general, concrete syntax lists the concrete graphic elements and maps each of them to one of the metamodelling elements. This strict separation between abstract and concrete syntax does not exist in the field of formal languages. Grammars only implicitly define the abstract syntax of a word of the language by the application order of the productions. This results in the so-called AST (abstract syntax tree) which does not only come closest to abstract syntax due to its name.

As long as the textual representation so described is the only one necessary, this monolithic syntax definition is sufficient. Nowadays, several concrete representations of the models are, however, required ever more often. On the one hand, this results from the fact that different representations are perceived to be more comprehensible by process roles due to different qualifications and other background knowledge, which leads to the fact that it must be possible to represent the same issue in different ways. On the other hand, there are technical reasons why an encoding of models is required in different ways. An XML format might be required for exchanging the models, while the models are to be managed within a relational database system. Normally, a requirements engineer or system designer does not want to create and process the models neither in XML nor in SQL. Rather, further textual and diagrammatic syntaxes are required for the actual processing.

In terms of a tool architecture which can be maintained but also of a comprehensible syntactical definition of the modelling language including all representations, the aim is to be able to define several concrete syntaxes based on a mutual abstract syntax for the modelling language. Herein, the conversions between the individual concrete syntaxes should already be explicitly defined via the abstract syntax.

The separation of abstract and concrete syntax allows for an introduction of so-called canonical concrete syntaxes, a crucial class of concrete syntaxes. These syntaxes are characterised by that their definition can be applied to any abstract syntax. Thus, the encoding rule is already explicitly determined just by the abstract syntax. One major field of application of canonical concrete syntaxes are technically-motivated model representations. It would be helpful, for example, if the corresponding XML encoding and relational mapping would already be explicitly defined by a given abstract syntax. Together with an also canonical object representation in the programming environments for the tools (such as Java), the foundation would be laid for generating the entire model management layer in the tools. Only by that, for example the tool prototyping, which is so important in software engineering, could be performed much more efficient in the scientific field.

Most model representations which are well comprehensible to humans, however, cannot be represented by canonical syntaxes. Nonetheless, canonical concrete syntaxes also result in advantages here, too: language engineering normally is an iterative process. In the early stages, the language designer wants to concentrate more on the conceptual design of the

language and thus on the abstract syntax of the language. In order to be, however, able to evaluate the viability of the concepts, it must be able to define instances of the abstract syntax already at that early stage. Here, a canonical concrete syntax provides assistance. A seamless transition to an improved (no longer canonical) syntax as regards legibility, could then be ensured via customising the canonical syntax.

In particular with regard to the fact that the domain-specific portions of the languages increase due to the high level of abstraction, thus considerably limiting the fields of application at least in the first step, efficiency in language design plays an increasingly more vital role. In return, the number of languages required also increases as each one of them is tailored to very specific aspects. In particular the canonical syntaxes, no matter whether they are technically driven in the back-end of the tools or whether they are a fast way of expressing model instances, allow for the more efficient language engineering required.

All in all, an increasingly stronger trend away from grammarware towards modelware becomes obvious due to the reasons mentioned above. Accordingly, metamodels will be employed more and more often for the primary and relevant description of the modelling language according to which all further representations will have to orient themselves, rather than using a grammar, such as in form of a EBNF. In the same way, also the primary storage format for models is changing. More and more development tools no longer store their models in a number of text files, but rely on for example a central database. Examples are tools like IBM Rational Doors [IBM, 2010] or MetaEdit+ [Tolvanen, 2004]. A preliminary stage is storing data in a set of XML-files: it can be more easily integrated in existing file-based repositories like Subversion [Tigris.org, 2010].

Although modelling abstract syntaxes resembles data modelling in most parts, extended data modelling concepts are required for metamodelling. Suitable metamodelling concepts would have to be provided for common issues such as namespaces, definition of validity ranges, encapsulation and re-use by instantiation or explicit sub-specifications by consistency conditions in several steps. Today's metamodelling languages are not yet characterised by such concepts which is the reason why they do not significantly differ from data modelling languages.

Basically, the use of UML class diagrams or comparable approaches has proved effective in practice and is commonly used for describing abstract syntax. Up to now, these metamodelling languages only are semi-formal, however, which strongly limits the benefits of metamodels in many situations. Nowadays, the metamodel cannot be used for the formal definition of abstract syntax in the scope of the specification of a modelling language. As a workaround, for example an EBNF is defined, but the correlation to the metamodel thereof is compelled to be only loose, however. This renders for example the most important question unclear, namely whether both language descriptions are consistent, i.e. whether each instance of the metamodel can be explicitly be mapped to one word of the language defined via grammar. In general, evidence regarding the metamodel cannot be obtained. Examples hereof would be, whether a given instance is valid with regard to a given metamodel, or whether a valid instance for a given metamodel does exist at all. Moreover, formal semantics cannot be defined in a consistent manner based on abstract syntax in the form of the metamodel. Instead, another formalism such as the EBNF is used.

Upon the implementation of a modelling tool, the metamodel is finally supposed to be transformed into a suitable form for the respective programming environment. As has already been described above, normally several transformations will be required, such as into a Java object representation, a relational scheme and an XML scheme. Due to the lack of formalisation, subtle deviations from abstract syntax will occur. This in turn will result in models in a concrete tool which are not allowed in the abstract syntax of the modelling

language at all. Neither do such false models dispose of a defined semantics, nor can be ensured that they can be processed without errors in the tool.

Due to the reasons mentioned above, the metamodel is nowadays often only used for a better explanation of a different, formal definition of the modelling language. If, in addition to the metamodel, an additional formalism is omitted, the entire specification will become informal. According to the author, this is one of the main reasons why experts already question the fundamental necessity of metamodels.

1.2. Three levels of integration

When discussing the term *integration* in industry, researchers intend to address a wide range of issues in most situations. In the broadest sense, companies want to optimise their development processes. Although many issues can be solved by coaching the persons concerned, an appropriate tool support will become reasonable and necessary at a certain point: The complexity of process artefacts – such as requirements documents containing thousands of pages or implementations containing millions of lines of code – makes it difficult to handle them manually. To sum it up, it can be stated:

The tool-supported re-use of artefacts in downstream process phases is indispensable for a further process optimisation.

This major issue is in turn the starting point of the three layered levels of integration as illustrated in Figure 1.4.

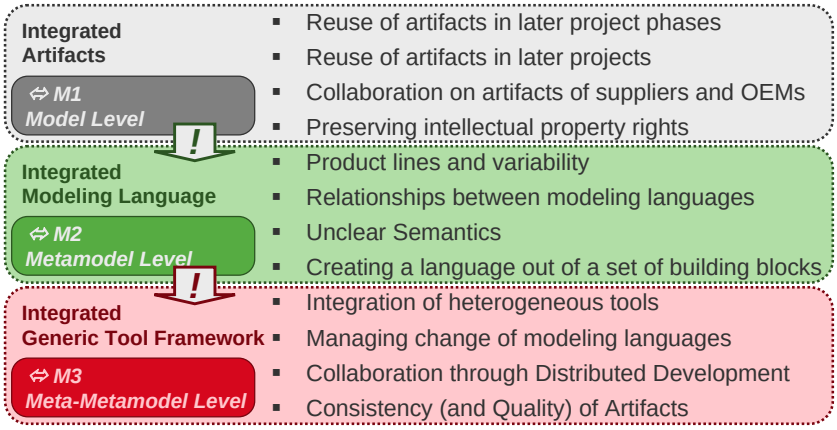


Figure 1.4.: The three levels of integration.

These levels of integration can be directly associated to the metamodelling stack as e. g. propagated in [Bézivin and Lemesle, 1999]:

- The level for *integrated artefacts* corresponds to the *model level (M1)* since the artefacts which are modelled by the engineers are exactly the M1-models.
- The second level for *integrated modelling languages* corresponds to the *metamodel level (M2)*: metamodels specify and describe the modelling languages: An integrated modelling language results from an integrated metamodel. Note that at the end a *product data model* or *artefact model* will exactly describe a such integrated modelling language and is thus also described by a metamodel. Nevertheless, product data

models of today are often reduced to high level descriptions. The detailed metamodel parts of the modelling languages such as how to exactly model automatons are skipped by using the definitions of dedicated tools.

- The level for *integrated generic tool framework* corresponds to the *meta-metamodel level (M3)*: The meta-metamodel is the specific metamodel for describing metamodels themselves in a bootstrapping way, thus describing the metamodelling language. Since a generic tool framework does not depend on a specific modelling language, such a framework needs to be configured by a concrete metamodel which describes a concrete modelling language. Due to this fact the power of the generic tool framework strongly depends on the power of the metamodelling language. If e. g. the generic tool framework is supposed to be able to check complex consistency constraints, the metamodelling language will have to provide constructs to formulate such constraints. Or if else the generic tool framework is supposed to be able to provide specific model editors, the metamodelling language will have to provide constructs to formulate concrete syntaxes.

Nevertheless, the focus of companies designing their own products – such as OEMs or suppliers in the automotive or avionic industry – is on the integration of artefacts which is the first level of integration. Nonetheless, they also have to care about the other two levels since the first level strongly depends on the others. This fact is often neglected, resulting in unsatisfactory solutions:

One common example for reusing artefacts is generating test cases from the requirements within the testing phase. In order to realise such a scenario, a semantically sound relationship between formal requirements and test cases must be established within the modelling language used.

Besides all the scientific questions concerning test case generation it is also a pragmatic tooling issue in terms of an integrated modelling language: The generated test-cases should have a link to their original requirements. This need results for example in an integration of the partial modelling languages for requirements as well as test cases, since the test cases are supposed to reference the requirements. If a company uses an integrated modelling language, establishing such a link becomes relatively easy; if not, the result will be endless integration discussions.

But even if an integrated modelling language exists in terms of a metamodel, this is no guarantee that this metamodel can be easily adopted to the existing tooling platform. If the metamodel is implemented by several heterogeneous tools – may be by their own model repositories – such a link between requirements and test cases may fail because of implementation issues: The result might be that enabling such a link would be too expensive, since a return on invest can not be achieved. Only a generic tooling platform which can be easily configured by the metamodel itself can solve this problem in general.

This dependency between those three integration layers results in a huge number of issues which, as it seems, have to be solved simultaneously. This fact makes it difficult for companies to work out a clear tooling strategy since the field of issues is too large. Therefore, the hierarchy of integration levels given herein allows for a clearer structuring of the issues to be solved. It should also show that the levels strongly depend on each other even though they seem to be far from the original issue of reusing artefacts in downstream phases of the process.

Based on this differentiation of integration levels, another important aspect can be worked out as well:

Different types of engineers have to solve the problems on different integration levels: model engineers, language engineers and tool engineers.

Model engineers are responsible for creating integrated artefacts. Normally, these engineers can be found at OEMs or suppliers. These engineers use the development tools provided. Their know-how is very specific as regards the concrete products to be developed and the corresponding development processes. Model engineers do rely on an appropriate modelling language.

Language engineers are responsible for creating an integrated modelling language. Normally, these experts are to be found at business-domain specific tool vendors. In order to achieve an optimal result, they receive their requirements from model engineers. Their know-how focuses on a specific business domain, such as automotive industry, in order to understand their detailed needs. In contrast to model engineers, they must have a detailed knowledge of language engineering and metamodelling, thus how to develop an integrated modelling language in general. In particular, language engineers need to know all details of the metamodelling language used and thus rely on an appropriate metamodelling language.

Tool engineers are responsible for creating an integrated generic tool framework including an appropriate metamodelling language configuring this framework. Normally, these experts are to be found at general purpose tool vendors such as database vendors (e. g. Oracle or IBM). Their knowledge is completely independent of a specific business domain. They concentrate on solutions which can be adopted by various business domains. They are the specialists in how to create efficient data backbones, how to support a model-based versioning or how to support complex consistency checks in general. Especially this third integration layer is neglected by OEMs and suppliers. The challenges of this layer are too far from their mayor problems. Nevertheless, the quality of this third layer has great influence on the final solution provided to the OEMs and supplies.

1.3. Approach

Up to now, a huge number of metamodelling approaches already exists, not to mention the related domains such as databases, mark-up languages, and ontologies. All of them define their own metamodelling language. Nevertheless, the degree of formalization is much lower than in formal languages. In order to discuss semantics and expressiveness of metamodelling languages in a more sound way, we will have to increase the degree of formalization. The crucial weak point of metamodelling is the lack of a *formal* and *appropriate* semantic domain. There is a set of formal approaches, such as KM3 [Jouault and Bézivin, 2006], but it is not suitable for industrial practice. Important needs to mention are the support of orders and duplicates, uniqueness of canonical keys, context-sensitive domains for properties (thus to restrict the set of possible values for one property), and an instantiation concept. On the other hand, suitable approaches such as MOF [OMG, 2006a] are informal, or the formalization is extremely complex, since these languages themselves provide a huge set of constructs. Although, however, it is rather complex, MOF does not support all of the above listed needs.

Informally, a metamodel defines the set of models which is valid according to the given metamodel. This definition leads to the question about what a model is. In most cases it is a graph-like structure, but the exact definition varies for each metamodelling approach. In contrary, the theory of formal languages has a very clear understanding of what a model is:

It is called word and is simply defined as a sequence of terminal symbols, which are elements from the so-called alphabet. On that basis it is easy to discuss the definition of languages: A language is defined by a set of valid words.

Up to now, no similar and suitable definition is available in the metamodelling domain. We therefore intend to propose the corresponding terms *abstract alphabet* and *abstract word* in order to allow for a clear definition of *abstract languages*. This is the reason why we are going to introduce a special kind of graph which will represent an abstract word. In other words we will define a semantic domain for metamodelling: Once we know the definition of an abstract word, we will be able to discuss how to define an abstract language (which is defined as a set of abstract words). To specify the semantics of a metamodelling language, we need to define for each abstract word whether it is specified by a dedicated metamodel within the abstract language or not.

As we will see, we strictly separate models (abstract words) from metamodels. Thus, a model can exist without any metamodel. Formal languages follow this paradigm in a natural way: We can write down a word based on a dedicated alphabet without knowing anything about the grammar. Afterwards we can discuss whether it is part of a given language or not. XML [Bray et al., 2006] follows the same paradigm: we can write down a XML document without having any scheme definition. In contrast, most of the metamodelling approaches do not have this separation. The most explicit formulation of the dependency of models to metamodels is given in the formalization of KM3 in [Jouault and Bézivin, 2006]: "A model $M = (G, \omega, \mu)$ is a triple where [...] ω is a model itself (called the reference model of M)." Thereby ω represents exactly the metamodel. Nevertheless there are some relevant reasons why a separation of models from metamodels is crucial:

1. *Support of bottom-up language engineering.* When specifying a new language, writing down a concrete model may often be the first step in order to get a clearer idea about the language to be created. Afterwards the language definition (metamodel) itself will be specified. This procedure is only possible if the metamodel is not required for specifying a model. This is also an advantage of XML and textual languages in general: Neither an XML scheme nor a grammar are necessary for defining models in the first step.

2. *Ability to correlate different metamodelling approaches.* Often, model data has to be transformed from one technical space as described in [Bezivin and Kurtev, 2005] to another. For example, if we want to convert models stored within an object database into an XML document for exchange reasons, or if we want to represent the same model by a textual language. In such cases we will have to correlate the corresponding approaches for defining the structure of models such as ODL [Cattell et al., 1999], XML scheme [Fallside and Walmsley, 2004], and EBNF [ISO, 1996]. It is difficult to correlate metamodelling approaches without a generic formal framework for models. As a matter of course, such a definition cannot comprise metamodelling aspects, as then, it could not be independent of a specific metamodel.

3. *Avoidance of recursive model definition.* When discussing metamodelling approaches, different modelling levels are often introduced. In general, such definitions make sense, of course. Nevertheless such approaches lead to self-describing meta-metamodels and hence they cause a problem when we want to formally define the meta-metamodel by itself, i. e. by something we do not know yet as we are still about to define it.

4. *Suitability of metamodel evolution.* An important aspect in metamodel evolution is the migration of existing models. During the migration of models we have to deal with a switch from one metamodel to another. To talk about the state of the model during the migration between these two metamodels it is helpful to use a description of the model which is independent of both metamodels.

Up to now, we have discussed the needs related to the *formality* of the semantic domain for metamodelling. Now we want to motivate the needs related to the *suitability* in industrial practice. Hereby, we want to address the gap between theory and practice. When analysing the differences which are relevant for defining a semantic domain, the following two major issues are currently not sufficiently addressed by formal metamodelling approaches although they are necessary in real applications:

1. *Dealing with attributes and enumerations.* No practically feasible model can do without attributes. Most of them also rely on enumerations. We want to show how they can be mapped to our approach without explicitly extending it by such concepts. This guarantees for an easy basic theory which covers these additional but important concepts as well. All operators introduced for abstract words can also be applied to attributes and enumerations.

2. *Dealing with orders and duplicates.* A second issue of practically feasible metamodelling environments is the support for orders and duplicates. In MOF for example we can add $\{ordered\}$ and $\{bag\}$ to an association end in order to indicate that such an association is (totally) ordered and allows duplicates, respectively [OMG, 2007]. Formal approaches are mainly based on the set theory and thus do not take orders and duplicates into account. In contrast, our formal framework is based on partially ordered multi-sets (pomsets). As we will see, pomsets allow you to handle orders and duplicates as well as all combinations thereof.

1.4. Related work

As described in both sections above, metamodelling is of great importance for practice when developing modelling languages. This is the reason why many solution approaches exist, wherein the most important of which will be represented in the following. Now that the development of programming languages in compiler construction is already well-established in the field of informatics, well-known and proven theories will have to be examined with regard to their relevancy in metamodelling. The most important representatives influencing the present work above-average are Relational Algebra [Codd, 1970], graph grammars [Rozenberg, 1997], description logics [Nardi et al., 2003], MOF [OMG, 2006a] with its formalization KM3 [Jouault and Bézivin, 2006], GME [Ledeczi et al., 2001], and OCL [OMG, 2006b]. All in all, the following works and topics will have to be considered for the individual questions:

- **Definition of the term metamodelling.** [Kurtev et al., 2006] provides a good overview of the issues in metamodelling. Among the early approaches of systematically dealing with the topic of metamodelling are [Geisler et al., 1998] by Geisler and [Bézivin and Lemesle, 1999] by Bezivin. The results were documented in the scope of the specification of KM3 in [Jouault and Bézivin, 2006]. In [Harel and Rumpe, 2004], Harel and Rumpe describe the elements of a language definition.

  One vital branch in metamodelling is the transformation of models. Important representatives of transformation languages which evolved specifically in the domain of metamodelling are GReAT [Agrawal et al., 2006] and [Vanderbilt, 2010], ATL

[Jouault et al., 2006a], BOTL [Braun and Marschall, 2003] and QVT [OMG, 2005a]. These and other transformation approaches are compared in [Falkowski, 2005]. Model transformation can be understood as a translational definition of semantics. The challenges are described in [Cleenewerck and Kurtev, 2007].

- **Mathematical formalisation of abstract words.** In the field of databases, relational algebra form the general fundament based on sets and relations. Relational Algebra was primarily introduced by Edgar F. Codd in 1970 [Codd, 1970]. Up to now, it still forms the formal basis for most commercial database systems including their query-language SQL [ISO, 2008]. A detailed introduction is provided in [Abiteboul et al., 1995]. First, Codd defines a database state as a set of relations. Based on that, the algebra on relations allows to alter the database state with a small set of operators as well as to formulate consistency conditions. Due to that, Relational Algebra mainly inspires the present work besides formal languages. The weak point of relational approaches is dealing with e.g. multi-sets or ordered sets. Complex attributes are also not provisioned in the original theory. Needless to say that for each of the problems mentioned, extensions of Relational Algebra will evolve over time. A good overview is provided in [Hull, 1986]. The embedded relation's model is for example discussed in [Jaeschke and Schek, 1982]. The semantics of the language Alloy, described in [Edwards et al., 2004] and [Jackson, 2006], is for example defined on the basis of sets and relations.

  Dar and Agrawal introduce a closure operator for SQL in [Dar and Agrawal, 1993], and Stephane Grumbach and Tova Milo extend the Relational Algebra to an algebra for pomsets in [Grumbach and Milo, 1995]. Another approach from the field of databases which resembles the techniques of metamodelling more than the relational approach, is formed by object-oriented databases. Object-oriented databases as formalised by Georg Gottlob, Gerti Kappel and Michael Schrefl in [Gottlob et al., 1990], allowing for multivalued attributes. The corresponding counterpart to SQL is OQL which is specified in [Cattell et al., 1999]. Object-oriented databases are, however, as yet not established. This also the reason why ODMG, the standardising medium for OQL, was cancelled. Many of these concepts were, however, not realised in object-relational databases.

  Nevertheless, the resulting theories loose the impressive simplicity of the Relational Algebra. An exception is the mentioned in the work of Grumbach and Milo, but in contrast to our approach they order the tupels partially, while our approach bases on partially ordered edges which allow a one-to-one mapping to the metamodelling domain. Another difference to the Relational Algebra is that in our approach, a model is described by a set of edge-functions instead of a set of relations. This re-definition allows you to easily solve the major weak points of the Relational Algebra that occur upon adaption for the metamodelling domain without loosing its simplicity: 1. Multivalued attributes are fully supported, 2. a closure operator is provided, and 3. ordered sets and multi-sets are supported.

  Graphs are another possibility of representing instances of abstract syntaxes. Such an approach is described in [Ebert and Franzke, 1994]. Based on these works, also the Graph Exchange Language (GXL) was developed, the concepts of which being described in [Holt et al., 2002] and [Winter et al., 2002].

- **Description language for abstract syntaxes.** As described in [HR00], the definition of structural semantics is most important in connection with metamodelling. There are some works about formalising the structural semantics of UML class diagrams, such as [Breu et al., 1997], [France et al., 1997] and

[Henderson-Sellers and Barbier, 1999]. In the scope of pUML, [CEK01, CEK02] also formalised metamodelling amongst others. The concepts of UML are, however, very comprehensive, resulting in an unnecessary complex formalisation.

The description language for metamodels which is cited most often is the Meta Object Facility (MOF) of OMG. It is specified in [OMG, 2006a] along with [OMG, 2007]. MOF provides a standard for defining the abstract syntax of modelling languages. There are a number of realizations of MOF, such as the Eclipse Modelling Framework (EMF) [Budinsky et al., 2003], which is currently probably the one most widely used. Due to the complexity of MOF these implementations rarely realise the entire specification. As the MOF standard as such is lacking a formal foundation, several attempts have been made to define a formalization.

Poernomo presents a formalization of MOF [Poernomo, 2006] which is based on constructive type theory (CTT). This formalization is particularly suited to prove the correctness of metamodels through well-typedness. In [Boronat and Meseguer, 2008], Boronat and Meseguer present an algebraic semantics for the MOF standard in membership equational logic (MEL). As they have operationalised this semantics within the Maude language, it can be used to perform formal analyses on models and metamodels. In contrast to these formalizations of MOF, we propose a formal framework for metamodelling in general which can be instantiated to formalize MOF.

In [Jouault and Bézivin, 2006], Jouault and Bezivin present a formal semantics of their textual language KM3 which addresses a subset of MOF called Essential MOF (EMOF). This formal semantics is based on Prolog and defines a number of predicates for nodes, properties and edges to relate a model to its metamodel. These predicates relate a model element to its corresponding metamodel element. Due to space constraints, they only show the formalization of classes including inheritance, and references including bi-directionality. The authors claim that the attached Prolog specification provides formalization of packages, primitive data types, enumerations and attributes. The formalization of enumerations, however, seems to be missing in the Prolog specification. KM3 comes closest to the approaches of this work, does, however, only provide few basic constructs which in turn leads to the fact that important concepts required in language development, cannot be realised. For this reason, the approach presented herein, goes far beyond KM3.

If graphs are chosen for representing abstract syntaxes, the graph grammars are the most original form of description. Graph grammars have been invented in the early seventies in order to generalize textual grammars [Rozenberg, 1997]. As a consequence, they also advocate a strict separation between a graph and its corresponding grammar. The linear and context-free forms thereof will be treated in [Pavlidis, 1972] and in detail in chapter 3 of [Rosenberg and Salomaa, 1997]. However, graph grammars become more constructive by providing rules to produce all graphs belonging to one language. In contrast, our framework is more descriptive with regard to that it constrains the graphs belonging to one language. While allowing the specification of duplicates, graph grammars do not cater for orders. In practice, this way of description has not proven to be effective, as the productions – similar to context-sensitive grammars – cannot be grasped intuitively. Moreover, most questions cannot be answered in graph grammars.

Context-free grammars, as can be described in EBNF [ISO, 1996] for example, do not differentiate between concrete and abstract syntax. By constructing the grammar, an abstract syntax is instead defined implicitly, which will then manifest in the AST. As context-free grammars also define an abstract syntax implicitly, the concepts can be used on their own by withstanding from using terminal symbols. With the help of

Zephyr, such an approach is provided in [Wang et al., 1997]. Only tree-like structure can be described, however.

Another modelling environment tailored to language engineering is the General Modelling Environment (GME), which is presented in [Ledeczi et al., 2001] and [Davis, 2003]. The Generic Modelling Environment (GME) provides a metamodelling formalism for defining a modelling environment [Ledeczi et al., 2001]. The underlying multi-graph architecture emerged from a generalization of component-based embedded systems [Biegl, 1995]. Even though the origin of this architecture is quite formal, there is no up-to-date formalization of the approach. In addition, there is neither support for duplicates nor for orders. A detailed documentation of version 5 can be found in [Vanderbilt, 2005]. In the field of the XML, XML scheme, defined in [Biron et al., 2004], [Fallside and Walmsley, 2004] and [Thompson et al., 2004] of the W3C, has established. It is, however, tailored to the requirements of XML and not to those of metamodelling. [Edwards et al., 2004] shows that the type theory is also well-suited for describing abstract syntaxes.

The requirements to description languages for abstract syntaxes are pretty close to those for ontology description languages. Description Logics (DL) is a family of languages for knowledge representation to describe ontologies in a formally well-understood way. D. Nardi, R. J. Brachman, F. Baader, and W. Nutt provide a detailed insight into DL in [Nardi et al., 2003]. Most of the DLs are a decidable subset of first order logic which makes them attractive for inferring new knowledge from already existing knowledge. In contrast to our approach, the expressiveness is much more restricted which makes it insufficient for metamodelling. DLs differentiate between a terminological box (tbox) and an assertional box (abox). The first describes the concepts of a domain, whereas the latter deals with knowledge about concrete instances. This distinction corresponds to that of metamodels and models in our domain. Please note that a concept in our approach is not the same as in DLs: In our approach each individual (node) is mapped to exactly one concept, while an individual may belong to many concepts in DL. Thus, a concept in DL can be seen as a node evaluation in our approach which makes it easy to use DLs in our approach. The W3C proposes OWL along with RDF for ontologies. The structural semantics of RDF is described in [Hayes and McBride, 2004], the one of OWL in [Patel-Schneider et al., 2004]. An introduction to the theory of description logics used is for example provided in [Nardi and Brachman, 2003]. Similarities and differences between ontologies and metamodels are discussed in [Gitzel et al., 2004]. [Saeki and Kaiya, 2006], however, tries to establish a common basis for metamodelling and ontologies. First approaches for a mutual development of ontologies are described in [Hepp et al., 2006] or [WikiOnt, 2010].

Regardless of the concrete description technique, a differentiation must be made between textual and diagrammatic metamodelling languages. UML for example applies a diagrammatic syntax. KM3 [Jouault and Bézivin, 2006] in contrast relies on a textual description language. Emfatic represents a proposal for a textual description language for Ecore models. Up to now, however, only a first draught exists, which can be found at [IBM, 2004].

- **Extension of metamodelling by context-sensitive concepts.** Attribute grammars, as presented by Knuth in [Knuth, 1968], [Knuth, 1971] and [Knuth, 1990], are often proposed for describing context-sensitive concepts. [Hedin, 2000] describes the problems arising thereupon and provides a respective solution. Context-sensitive proportions are described by OCL [OMG, 2006b] in UML. The Object Constraint Language (OCL) provides a standard for an expression language in order to navigate

and constrain models [OMG, 2006b]. As precise semantics is not part of the standard, there have been a number of attempts to formalise OCL. Brucker and Wolff propose a semantics for OCL based on a shallow embedding in Isabelle/HOL in [Brucker and Wolff, 2002]. In [Kyas et al., 2005], Kyas et al. present a mapping of OCL constraints to the PVS theorem prover. In [Markovi and Baar, 2006], Markovi and Baar propose a formal semantics for OCL constraints based on their evaluation as QVT model transformations. While these formalizations map the OCL constraints to a separate semantic domain, our approach provides an algebra directly working on edges. In contrast to OCL, wherein constraints cannot be evaluated in a metamodel-independent way, edge algebra does not require a metamodel to be present. In contrast to OCL, edge algebra is a very tiny language which is therefore much easier to understand as well as to be implemented. Nevertheless due to its pomset support, it is even more powerful than OCL in constraining bags and lists. No metamodel is provisioned for the language OCL itself. [Reichmann et al., 2004] introduces a respective OCL metamodel. The metamodelling language Kermeta – described in [Fleurey et al., 2007] – extends the concepts of EMF by methods and constraints in order to be able to express static semantics.

- **Modularisation of language specifications.** MOF provides basic coarse-grained operators for the composition of modelling languages by for example importing, merging or combining packages [OMG, 2006a]. Blanc et al. motivate the need for a new operator that allows to reuse and generalize concepts when combining packages [Blanc et al., 2005]. Clark et al. provide a new composition operator that allows to equate concepts before merging the packages [Clark et al., 2002]. Karsai et al. propose more fine-grained operators that allow for the composition of modelling languages by for example uniting two concepts, or finer control over inheritance relationships between two concepts [Karsai et al., 2004]. Balasubramanian et al. show how to apply these operators to the integration of existing model-based development tools [Balasubramanian et al., 2007]. Estublier et al. provide similar constructs, but allow not only for the composition of the generated editors, but also consider a composition of corresponding model interpreters [Estublier et al., 2005]. Emerson and Sztipanovits envision metamodel templates that enable a more flexible generalization and customization of modelling languages [Emerson and Sztipanovits, 2006].

- **Language evolution.** When a specification changes, probably all existing instances will have to be reconciled in order to conform to the updated version of said specification. Since this problem of *coupled evolution* affects all specification formalisms (e. g. database or document schemata, types or grammars) alike, numerous approaches for *coupled transformation* [Lämmel, 2004] of a specification and its instances have been proposed. The problem of schema evolution, which has been a field of study for several decades, has probably received the closest investigation [Rahm and Bernstein, 2006]. Recently, literature provides some work that transfers ideas from other areas to the problem of metamodel evolution. In order to reduce the effort for model migration, Sprinkle proposes a visual, graph-transformation based language for the specification of model migration [Sprinkle and Karsai, 2004]. Gruschko et al. envision to automatically derive a model migration from the difference between two metamodel versions [Becker et al., 2007, Gruschko et al., 2007]. Wachsmuth adopts ideas from grammar engineering and proposes a classification of metamodel changes based on instance preservation properties [Wachsmuth, 2007].

- **Description of formal languages based on abstract syntaxes.** Many approaches, such as [Efftinge and Völter, 2006] with xText or [Alanen and Porres, 2003] describe a regulation which extracts the abstract syntax of a grammar in the form

of a metamodel. The resulting metamodels do, however, not substitute for an individually defined metamodel and therefore, they need to be converted into the target metamodel. The other way round defines a concrete textual syntax based on a given abstract syntax. As the abstract syntax does not contain any information about the concrete notation, this approach results in a canonical textual syntax. One example thereof is HUTN, which is specified in [OMG, 2004]. Experiences with HUTN are reported in [Muller and Hassenforder, 2005]. [Gargantini et al., 2006] provides a first, but rather technical approach defining a concrete syntax based on the abstract syntax. The result thereof is a parser, providing a metamodel instance. The language TCS, presented in [Jouault et al., 2006b], is an approach which reverts to the definition of an abstract syntax (which, in this case, is defined in KM3) for describing textual concrete syntaxes. An alternative but similar approach is described in [Muller et al., 2006], wherein said approach bases on EMF. [Guerra et al., 2005] and [Baar, 2006] show that besides concrete textual syntaxes even diagrammatic syntaxes can be defined. A really good introduction is provided in [Viehstaedt and Minas, 1995] within the scope of the project called DiaGen [Minas, 2010].

- **Mapping to the technical spaces of XML, SQL and Java.** [Kurtev et al., 2002] describes the technical spaces in detail. [Bezivin and Kurtev, 2005] and [Bézivin, 2005] describe the interaction of different technical spaces with the help of a higher-level metamodelling language and respective technical projectors. [Bézivin et al., 2005] describes a concrete conversion between GME and EMF. The Java Metadata Interface (JMI) [Microsystems, 2002] defines the mapping of MOF to a Java environment. XMI [OMG, 2005b] defines a canonical XML encoding of any models specified in MOF. Unfortunately, said encoding is rather long and illegible. This was also one of the reasons for the definition of HUTN. In contrast to the approach presented herein, XMI does not offer a suitable metamodel for each given XML document. The works concerning object-relational mappings, such as [Ambler, 1999], [Cabibbo and Porcelli, 2003] and [Cabibbo and Carosi, 2005], are decisive for mapping metamodels to relational databases. One of the most important implementations is the OpenSource project Hibernate [Hibernate, 2010].

- **Canonically binary encoded representation of model instances.** The XML Binary Characterization Working Group (XBCWG) of the W3C describes the requirements to a canonical binary encoding, for which calls have long since been issued, in [Goldman and Lenkov, 2005], [Williams and Haggar, 2005], [Cokus and Pericas-Geertsen, 2005a] and [Cokus and Pericas-Geertsen, 2005b]. In the context of BinaryXML, [Geer, 2005] discusses about the relevancy of efficient encodings. [Merenda, 2005] defines a simple binary encoding and checks the properties thereof with regard to the requirements of the XBCWG. In [Davis, 2003], GME as well describes the support of an efficient binary encoding.

- **Metamodelling as a means for model-based tool development.** Examples for approaches of integrated tool architectures are GeneralStore [Reichmann et al., 2004] as well as the commercial further developments thereof in Aquintos [Aquintos, 2010] or else the project Artwork [Artwork, 2002], [Artwork, 2003b] and [Artwork, 2003a] within the scope of which the work [Günzler, 2005] evolved. All approaches are based on a common metamodel having different levels of detail as regards the concept. Conventional approaches, such as [Braun and Marschall, 2003], rely on coupling heterogeneous tools by model transformation. Upon the implementation of AutoFOCUS [Hölzl, 2010], a dedicated metamodel is also defined in [AutoFocus, 2006], in order to generate the code for the persistence layer of AutoFOCUS therefrom. As with all tool implementations of that kind, the exact behaviour of the metamodel classes generated

can only be read in the generated code, as no suitable metamodelling language having a formal structural semantics was available. Scientists tried to eliminate that problem in AUTOSAR by explicitly defining a specification of the structural semantics of the metamodelling constructs allowed in [Autosar, 2006] – although, this was only done in prose.

- **Integrated language engineering.** In this context, the huge number of Language Workbenches, Code-Generator Frameworks and also Greenfield's Software-Factories [Greenfield and Short, 2004] must be named. Important representatives are the Microsoft DSL Tools or the Eclipse Modeling Framework (EMF) [Budinsky et al., 2003]. Another important aspect of integrated language engineering is the re-use of metamodels, which allows for the development of metamodel libraries. Concepts for the re-use by composition are described in [Emerson and Sztipanovits, 2006]. Amongst others, Charles Simonyi coined the term *intentional programming* in [Simonyi, 1995]: In order to take the different intentions of the modellers into account, one and the same model information is supposed to be represented in different ways, and thus it relies on the concept of separating abstract and concrete syntax, in order to ensure an integrated tool environment with regard to its editors. Being a former Microsoft manager, Simonyi founded the company called Intentional Software [IntentionalSoftware, 2010].

In literature, some approaches for tool integration can already be found. In [Becker et al., 2005], the authors propose a model-based approach to integrate tools working on interdependent documents. Wrappers for each tool allow you to abstract from technical details and provide homogenised access to documents by way of graph models. The different documents are kept consistent by graph transformation rules which allow you to propagate changes in an incremental development process. In [Becker et al., 2007], the authors provide a more detailed description of their algorithm for incremental and interactive consistency management. The authors of [Karsai et al., 2005] explain and compare two architectural design patterns which allow for tool integration. The first architecture is based on an integrated model and adapters for each tool translating the data to the integrated model. The second architecture is based on a messaging system, which routes data according to a workflow specification, and which implements a pairwise integration among tools. [Margaria, 2005] presents the extension of the ETI (Electronic Tool Integration) platform by web service technology. The integrated tools interact with each other by using web services, which allow to decouple the different tools from each other and which therefore ease integration and maintenance activities. In [Königs and Schürr, 2006], the authors present their rule-based approach MDI (Multi Document Integration) for data integration of multiple data repositories. Metamodels are used to provide an abstract specification of the different models, a separate model is used to specify correspondence links between the models, and rules are used to specify consistency between the models. The declarative rules, specified in the form of triple graph grammars, are used to derive code for creating and checking the consistency of correspondence links as well as for a forward and backward propagation of changes. TOPCASED [Farail et al., 2006] is an open-source CASE environment for model-based development of critical applications and systems. Their ambition is to build an extensible and evolutive CASE tool that allows its users to access various models and associated tools.

Besides these academic approaches, some tool developers already offer integrated tool support. For the automotive domain, Vector has developed the tool eASEE [Vector, 2010] which is intended to be a data backbone that stores the product data in a central repository. This tool is not designed as a generic tool integration platform, but focuses on supporting predefined modeling functionalities. The tool PREEVision from Aquintos [Aquintos, 2010] follows a similar direction.

1.5. Contribution

Due to the reasons mentioned above, the present work is to formalise metamodelling in order to develop an integrated, comprehensible and efficient language. The present work focuses on a systematic separation between abstract and concrete syntax. This is the only way of distinguishing the description of the structure and concepts of a metamodelling language from the concrete representations of the models. The following questions will be dealt with in detail:

- **Definition of the term metamodelling.** Nowadays, the term metamodelling is unfortunately used inconsistently both in the field of science and in practice. Thus, a somehow extended definition will be provided, clarifying which components a metamodel exactly comprises, which properties will need to be fulfilled by the individual aspects and how to describe them. This detailed description will result in a metamodelling language which can be described by its own in the sense of bootstrapping.

- **Mathematical formalisation of abstract words.** If the language modelling concepts are abstracted from concrete encoding, the words of the modelling language would also have to be abstracted from a concrete symbolism as a consequence. For this purpose, a specific class of labelled, directed multi-graphs is proposed. They form the mathematical basis for the formal definition of abstract syntaxes.

- **Description language for abstract syntaxes.** The set of all labelled, directed multi-graphs in the sense of the definition for abstract words describes the most general abstract language: Each abstract word possible is valid. Based on that, more specific abstract languages can be defined by abstract syntaxes which only allow a subset of all abstract words. Hereby, the modelling language developed uses concepts of well-proven metamodelling approaches, similar to the UML class diagrams, extends them and defines a formal structural semantics in the sense of valid abstract words of the modelling language.

- **Extension of metamodelling by context-aware concepts.** When designing languages, there are some important and common concepts for which it is very difficult or even impossible to be expressed in current metamodelling languages. Suitable metamodelling concepts will be provided for namespaces, definition of validity ranges, encapsulation and re-use by instantiation.

- **Description of formal languages based on abstract syntaxes.** Nowadays, abstract syntax in the form of a metamodel and a grammar for defining the formal language are facing each other in a rather independent manner. The present work shows a way of defining textual concrete syntaxes on the basis of an abstract syntax. Only the combination of abstract and textual concrete syntax will define the formal language. Due to the context-sensitive concepts of the abstract syntax, this kind of specifying formal languages is more expressive than context-free grammars.

- **Metamodelling as a means for model-based tool development.** The rules for mapping and encoding as well as the formal definition of concrete syntaxes allows for the development of respective code generators for tool development. Metamodelling is therefore turned into a vital part of model-based tool development. All in all, tool development becomes more systematic and more efficient due to that, which is of major significance both for domain-specific languages and for early experiments of newly developed languages in the scientific field.
- **Integrated language engineering.** The vital result of the present work is to enable an integrated development of modelling languages. On the one hand, the abstract syntax of the language describes a suitable taxonomy for the domain to be modelled. On the other hand, this results in a formal definition of the structure of the language. Concrete syntaxes – which are of textual nature in the present work – will be defined as an extension of abstract syntax. In general, several concrete syntaxes may also be given for different fields of application. The interaction will, in turn, be provided by abstract syntax. Moreover, the formal language definition by abstract and concrete syntax serves so as to generate parts of the corresponding tools.

1.6. Overview of the present thesis

Chapter 2, *Metamodels and Seamless Language Engineering*, p. 21 describes the components of metamodelling and thus provides a definition of the terms *metamodel* and *seamless language engineering*. The topics dealt with in the present work will be classified by way of the above-mentioned definition.

Chapter 3, *Running example: Modelling dataflow algorithms*, p. 45 provides an informal introduction to the running example which is a language for modelling dataflow algorithms. In order to illustrate most of the M2L modelling constructs, it also contains some sophisticated constructs such as instantiation. The concrete illustration model implements an integrator over time.

Chapter 4, *Pomsets in the context of metamodelling*, p. 61 introduces partially ordered multisets as a generalisation of sets and lists (which are totally ordered sets). Pomsets form the basis for defining abstract words in the following chapter.

The next four chapters form the core specification of the metamodelling language M2L itself:

- Chapter 5, *Models as Abstract Words*, p. 81 provides a mathematical description for abstract words in the form of a specific class of labelled, directed multi-graphs.
- Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 describes an algebra to derive new properties from already existing properties of abstract words. It forms the basis for the description of conditions for models.
- Chapter 7, *Abstract Syntaxes in M2L*, p. 119 specifies the metamodelling language tailored methodically to language engineering to define abstract syntaxes in the sense of a restriction of the valid abstract words. It is characterised by its specific possibilities to specify context-sensitive language properties.
- Chapter 8, *Textual Concrete Syntaxes in M2L*, p. 153 deals with describing formal languages based on a given abstract syntax. The definition of a formal language therefore consists of two components: abstract syntax as well as textual concrete syntax. Only a combination thereof defines a formal language.

Chapter 9, *The overall specification of M2L*, p. 171 provides the syntactical definition of the metamodelling language M2L in a bootstrapping way. This overall picture could not be presented in the previous chapters, as bootstrapping requires that the entire metamodelling language is already known. This specification is also used to provide a first detailed case study for M2L. The present chapter describes all concepts of the metamodelling language in a systematic way.

Chapter 10, *Summary, evaluation, and outlook*, p. 271 concludes by illustrating the advantages resulting from the present work as regards the development of novel modelling techniques and possible connecting factors for future research.

Chapter 2

Metamodels and Seamless Language Engineering

Since the present work deals with creating a suitable language to write down *metamodels* in order to allow a *seamless language engineering*, these two topics will be discussed and clarified in this chapter. Although in particular the term *metamodel* is wide-spread, in most cases it is a more semi-formal way of usage. Nevertheless, the rough understanding is quite consistent: A metamodel defines the *abstract syntax* of a language. Apart from that, a language definition consists of much more than abstract syntax. This chapter will reveal, however, that a metamodel should also comprise *concrete syntaxes*, *process definitions*, and *semantics*.

These four *aspects* of a metamodel form the first dimension of a seamless language engineering. Seamless language engineering focuses on integration and consistency in three dimensions: 1. integration of the four already mentioned metamodel aspects, 2. consistency of the three metamodel purposes *specification*, *documentation*, and *tooling*, as well as 3. integration and tailoring steps in order to achieve modularisation.

This detailed point of view leads to the requirements to a metamodelling language and also to a formal top-level definition of a metamodel.

Contents

2.1. Metamodels – comprising the four vertical tooling aspects

When looking at the very basic meaning of the term *metamodel*, we come to the prefix *meta* which has its origin in the Greek word of μετά, which can be literally translated by *with*, *amidst* or as regards time *after*, and figuratively also *above* or *behind*. As a conclusion,

metamodels would therefore be *models above models*, thus *describing* models which finally corresponds to a language definition.

In order to get a more precise understanding of which aspects should be part of a metamodelling language, in this section we will return to development tools which can be seen as implementations of modelling languages. A consequent realisation of a seamless language engineering results in the fact that such development tools are automatically generated from of the language definition and thus from a metamodel. Due to the foregoing, the concepts of model-based development are applied to the engineering of development tools themselves.

In the first step, we will decompose the more or less monolithic development tools of today into the four generic components *Repository*, *Editors*, *Workflow*, and *Analysis/Synthesis* (Section 2.1.1, *Decomposing monolithic development tools*, p. 22). Based on this, we are able to extract parts of development tools, that are independent of the concrete modelling language (Section 2.1.2, *Extraction of horizontal tooling aspects*, p. 24). Such non-variable parts are called horizontal tooling aspects, which will be *Common Model Repository*, *Generic Editor Framework*, *Workflow Engine*, and *Model Interpretation Engine*. These horizontal tooling aspects are generic and must therefore be configured by the language specific aspects: As a result, a corresponding vertical tooling aspect has to be defined for each of the horizontal tooling aspects (Section 2.1.3, *Vertical tooling aspects: the four aspects of a metamodel*, p. 26). These will be *Abstract Syntax* (configuring the common model repository), *Concrete Syntax* (configuring the generic editor framework), *Process Definition* (configuring the workflow engine), and *Semantics* (configuring the model interpretation engine).

Based on these four vertical tooling aspects, we are now able to configure a complete development tool. According to that, these four aspects in combination form the necessary information in order to build a concrete development tool in the sense of model-based development for development tools themselves. The author therefore proposes to enrich metamodels to all of these four aspects. Thus, the term metamodel will be used that way in the following.

2.1.1. Decomposing monolithic development tools

To all intents and purposes, the integrated modelling language should be operationalised by a tooling environment that supports the creation, transformation, analysis and subsequent processing of all artefacts needed. Due to the fact that tool development is extremely expensive, industry sees no alternative to already existing commercial tools. These tools are often of general nature only and are not tailored to the specific needs of the engineers of a specific industry. Many efforts trying to develop their own tailored integrated engineering environment fail because of huge development efforts but even more due to substantial maintenance costs. This is due to the fact that besides the core business functionality, tools require a lot of infrastructure for the management of models.

Nowadays, development tools mostly have a very monolithic character. Most tools do therefore not require any special platform besides an operating system. The internal structure of commercial tools is often not visible to users and thus a deep integration of two different development tools (e.g. to optimise development processes) results in huge efforts and sometimes turns out to be even impossible as no APIs are available.

Nevertheless, functionality of every development tool can, from a conceptual point of view, be classified in four classes, namely *Repository*, *Editors*, *Workflow*, and *Analysis/Synthesis* as shown in Figure 2.1:

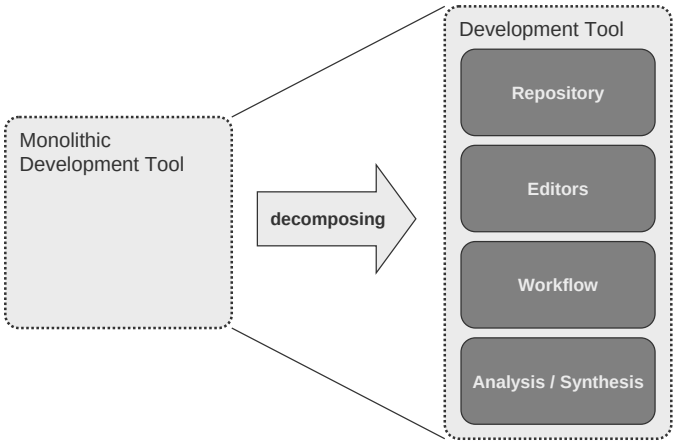


Figure 2.1.: Decomposing development tools.

- **Repository** subsumes all data handling, such as loading or saving models, handling files or accessing databases. It also includes for example all export and import functionality.

- **Editors** address all editing and viewing functionality. Note that not the entire GUI should be considered here, as all four parts also result in GUI elements. Menu items for saving models, for example, belong to *Repository*.

- **Workflow** addresses all process support of development tools. Most of today's development tools no longer represent a workflow support for a comprehensive systems engineering process. Nevertheless some functionality – such as deactivation of menu items – can be considered as a workflow support. If, for example, a model remains unchanged, the *Save* item is deactivated. Hence, the user is guided (at least somehow) depending on the current situation.

- **Analysis/Synthesis** addresses all functions which e. g. check the correctness of code, consistency of models or modelling guidelines (*Analysis*). *Synthesis* subsumes all compiler and code generation functionality in particular.

In order to reduce development costs, the tooling platform has to factor out that functionality which is independent of the specific product model. The tooling platform can then be parametrised by a modelling language which operationalises a certain product model. Our aim is therefore to achieve a strict separation of horizontal and vertical tooling aspects by means of an integrated tooling platform. Tooling aspects, such as the central model repository, which are independent of a specific modelling language, are termed *horizontal*. Tooling aspects, such as the syntax of a certain modelling language, which are specific to a certain modelling language, are called *vertical*. In today's development tools, these horizontal tooling aspects are interwoven with the implementation of the vertical tooling aspects. The lack of a separation between horizontal and vertical tooling aspects hampers the implementation of a central model repository which is crucial for introducing an integrated engineering environment.

2.1.2. Extraction of horizontal tooling aspects

The following is cited from [Broy et al., 2010] page 537 including some minor changes in order to adopt it to this work:

> *Tooling aspects are called horizontal if they are independent of a certain modelling language. Horizontal tooling aspects, such as a model repository, are often re-implemented by each isolated tool. However, using different technologies for one model repository complicates seamless tool integration, as models must be transformed to enable data exchange between the tools. We therefore propose a tooling platform that factors out horizontal tooling aspects. As shown in Figure 2.2, we identified the following horizontal tooling aspects required for large-scale seamless system development: Common Model Repository, Generic Editor Framework, Workflow Engine, and Model Interpretation Engine. In the following, we will deal with the different horizontal tooling aspects and their requirements in more detail:*

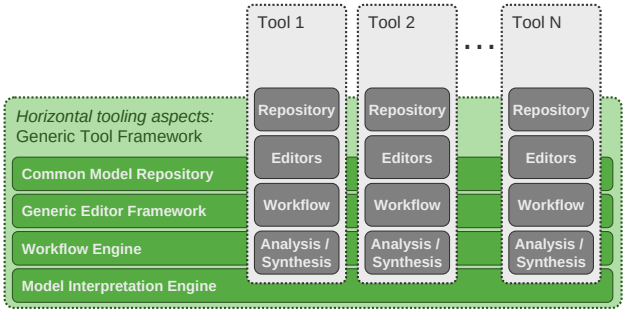


Figure 2.2.: Extract horizontal tooling aspects.

- ***Common model repository.*** *A central model repository is crucial for maintaining the dependencies between the different models produced during the development process. As the models of industrial systems are becoming quite large, a database system is required to store all models and their respective dependencies. The central model repository is also responsible to ensure the overall consistency of the models. A model is consistent if it fulfils the constraints defined by the modelling language.*

  In order to efficiently handle a distributed development of systems, the database system has to be distributed. The models may be partitioned according to the different companies participating in the development of a system as each company needs to have sovereignty over its own models. Furthermore, as some companies may not be permitted to access or modify the models of other companies, the model repository has to provide individual rights by access control.

  When distributed parties are simultaneously working on the same models, conflicts will arise, leading to inconsistencies. In order to prevent or repair conflicts, configuration management is to keep track of the different versions of the model. Furthermore, configuration management is to define which version of different models fit together.

  Object-oriented database systems are best suited for implementing common model repositories, as they can efficiently handle graph-like model structures.

Traditional file-based configuration management systems such as CVS and SVN do not meet the needs of model-based development. Current configuration management systems for models such as Odyssey-VCS mainly support a certain modelling language such as UML [Oliveira et al., 2005]. However, there is also research on configuration management systems which can be parametrised by a modelling language (e. g. ModelCVS [ModelCVS, 2010]).

- ***Generic editor framework.*** *A front-end provides a user interface for authoring models in the repository. The front-end should constitute a generic framework that can be parametrised by the modelling languages applied. The front-end provides editors to author a model in its concrete representation by using the concrete syntax of the modelling language. Furthermore, the front-end offers the modeller those operations which are defined by the modelling language (vertical) and operations that are common to all languages (horizontal).*

  These operations have to be intuitive to support the engineers in working with the models in an efficient manner. Those operations that support configuration management for example, should allow engineers to commit the changes on models and to update parts of the models. In case of a conflict, a merge operation is required that allows the visualisation of the differences between models in their concrete syntax. The Eclipse platform is a perfect example for a front-end, as its service-oriented architecture renders it highly extensible [Eclipse, 2010b].

- ***Workflow engine.*** *Our experiences show that most often, a defined process is not followed by its participants as long as it is not supported by the modelling tool. To prevent deviation from the process, developers should be guided through the defined process by the tooling platform. In order to operationalise the process, the workflow engine interprets the process definition of the modelling language. When interpreting a process model, progress and current activities that need to be performed are always available by the workflow engine. To force a modeller to perform the current activities, all operations and interpreters not required for the activity have to be suppressed. The rights management of the tooling platform has to ensure that certain activities are only performed by certain roles. When modellers log on to the tooling platform, they can only perform activities which are currently available based on the process definition and which correspond to one of the roles they own.*

- ***Model interpretation engine.*** *It provides the facilities to perform complex tasks such as analysis and synthesis based on the semantical definition of the language. To perform complex editing and refactoring facilities, an in-place model-to-model transformation engine is necessarily integrated in the front-end. For an automated generation of code and other process artefacts, an out-of-place model-to-model as well as a model-to-text transformation engine is required. As such generation tasks might require a lot of time and computing power they should be located at a different machine in the back-end. In order to be able to execute an operational semantics, a generic simulation framework is necessary which should also be located in the back-end because of resource consumption issues.*

End of citation.

By extracting the horizontal tooling aspects as described above, a generic tool framework as shown in Figure 2.3 is formed:

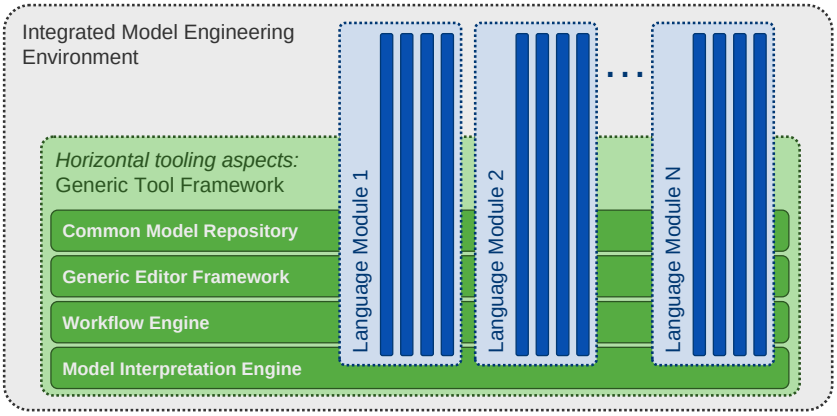


Figure 2.3.: Specific tools as language modules.

The specific tools become language modules which are plugged into that framework. These language modules are just a formal description of the corresponding modelling languages and thus metamodels. However, an integrated modelling language is rather complex and thus difficult to develop in one step. In order to ease language development, an integrated modelling language should result from the composition of reusable, modular modelling languages which can be customised to the specific needs of engineers. Furthermore, appropriate tool support is required for model migration in order to be able to improve a modelling language that is already under use.

2.1.3. Vertical tooling aspects: the four aspects of a metamodel

In the previous section it was already described that we are generally able to extract horizontal tooling aspects in form of a generic tooling platform. In order to configure this generic platform, a set of metamodels forming the language modules will be necessary now. As shown in Figure 2.4, these language modules build the vertical tooling aspects. For each horizontal tooling aspect, a corresponding vertical tooling aspect exists: *Abstract Syntax* configures the *Common Model Repository*, *Concrete Syntax* configures the *Generic Editor Framework*, *Process Definition* configures the *Workflow Engine*, and *Semantics* configures the *Model Interpretation Engine*.

While horizontal tooling aspects are independent of a business domain, vertical tooling aspects are specific for the concrete business domain. Thus, the generic tool framework can be used for various applications; by plugging in a specific language module, the engineering environment is configured for a specific purpose. The separation of horizontal and vertical tooling aspects also points out the different skills for developing an integrated engineering environment: A generic but deep tool development know-how including DBMS and GUI implementation is necessary for the horizontal aspects. In contrast to vertical aspects, a dedicated know-how for the concrete business domain, their development processes and even company specific issues are crucial. This separation of skills – especially concerning horizontal aspects – is often neglected when implementing such an integrated engineering environment resulting in a tool architecture of minor value.

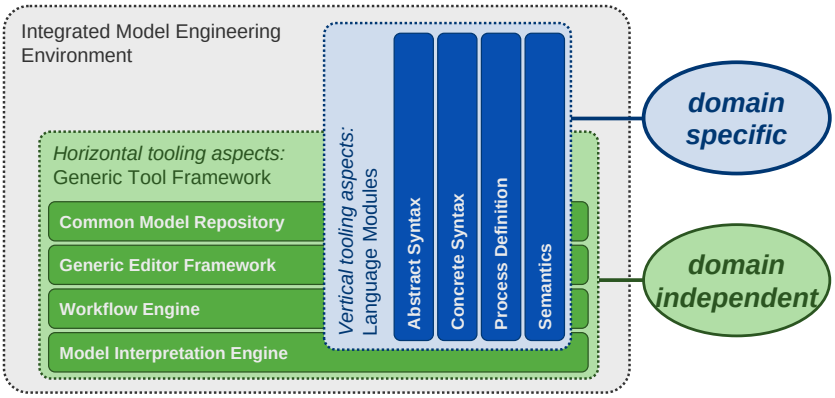


Figure 2.4.: Horizontal and vertical tooling aspects.

Figure 2.5 shows again the four parts *Repository*, *Editors*, *Workflow*, and *Analysis/Synthesis* into which a development tool can be decomposed. It also provides a more detailed overview of how these parts are made up of a generic and a specific fragment. The generic fragment is outside the *Generic Tool Framework* (horizontal aspect) and a specific one is provided by the *Language Module* which configures the generic framework (vertical aspect). Thus, tooling aspects are called vertical if they are specific for a certain modelling language. The generic tool framework on the other hand supports tool builders to implement vertical aspects easily. It should be easily possible for a company to adapt or develop a modelling language appropriate to its needs. In order to enable a cost-effective development of such modelling languages, so-called meta languages are required to describe the different elements of a modelling language. The tooling aspects related to supporting modelling languages are partitioned into the following elements: *Abstract Syntax*, *Concrete Syntax*, *Process Definition*, and *Semantics*.

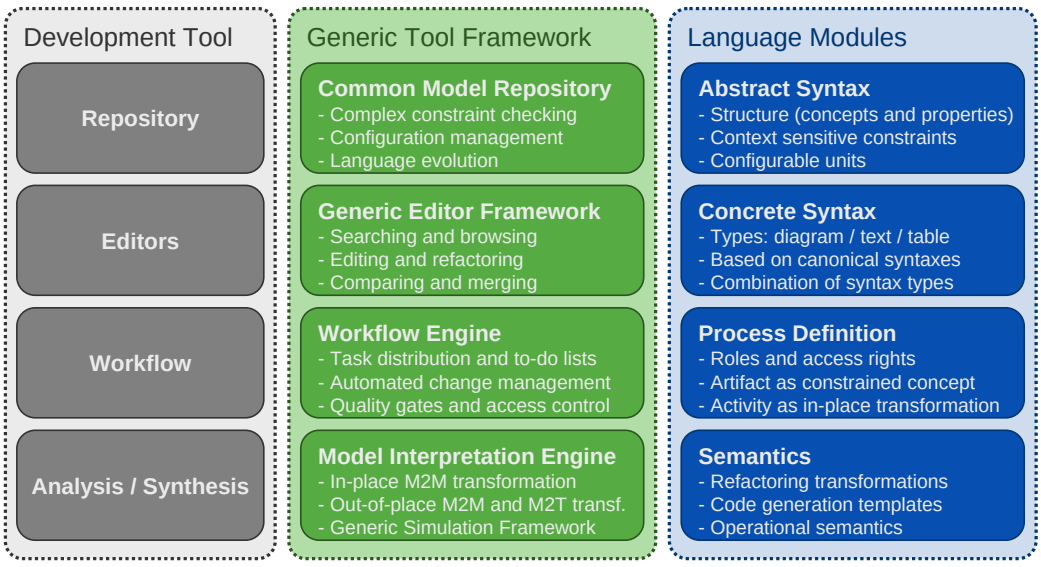


Figure 2.5.: Detailed relationship between generic tool frameworks and language modules.

The following is cited from [Broy et al., 2010] page 535-536 including some minor changes in order to adopt it to this work:

The different elements of modelling languages and their requirements will be examined in more detail:

- ***Abstract syntax.*** *Abstract syntax defines the concepts of a modelling language and their respective relationships. When a modelling language corresponds to a domain, it enables engineers to directly reflect the domain concepts and relations in their models. By using domain appropriate languages, engineers can work on a higher level of abstraction and in direct analogy to domain knowledge.*

  Abstract syntax determines the validity of models and can therefore be used to enforce the construction of valid models. Domain semantics of languages can be encoded in an abstract syntax by restricting syntactically correct models to those that are meaningful in the domain [Evermann and Wand, 2005]. Abstract syntax usually consists of constructive and descriptive parts: constructive parts describe how valid models are to be built and descriptive parts further restrict the number of valid models by constraints. As an integrated modelling theory needs to describe the relationship between different models, a model is required to have a graph-like structure.

  Abstract syntax in the centre of a modelling language definition as shown in Figure 2.6. Other elements of a modelling language definition (concrete syntaxes, process definition and semantics) are then specified in relation to abstract syntax. This enables the rapid development of modelling languages and provides a very clear structure of a language specification. Furthermore, different modelling languages are best integrated in terms of their abstract syntax.

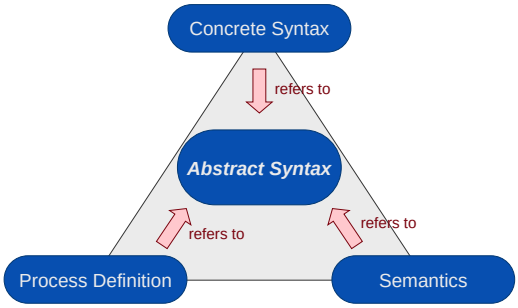


Figure 2.6.: Central position of abstract syntax.

Literature provides a large number of examples for languages to define the abstract syntax of a modelling language. The Object Management Group (OMG) even standardised languages to define the abstract syntax of object-oriented modelling languages: the Meta Object Facility (MOF) [OMG, 2006a] representing the constructive part and the Object Constraint Language (OCL) [OMG, 2006b] representing the descriptive part. MOF, however, provides too many constructs to be entirely understood and implemented. The most commonly known implementation of a subset of MOF, called EMOF (Essential MOF), is the Eclipse Modelling Framework (EMF) [Budinsky et al., 2003].

- ***Concrete syntax.*** *Concrete syntax defines the representation of a model in a human-readable manner. There are different forms of concrete syntax: diagrammatic, textual and tabular. Diagrammatic syntax shows the model in the form of diagrams with layout information, textual syntax visualises the model as linear texts, and tabular syntax illustrates the model in the form of two-dimensional tables.*

  As real-world models can become quite large, the concrete representation of a whole model becomes incomprehensible. As a consequence, we need to be able to define a concrete syntax only for viewing the model. Only the direct sub-components of a component, for example, are visualised in a diagram. Furthermore, the representations of the different views need to be related to each other. The black-box of a component is depicted in the diagram for its parent component, whereas the white-box is shown in a different diagram.

  Some modellers may prefer diagrammatic concrete syntax, while others prefer the textual one. As a consequence, there might be several representations of the same view in different variations of concrete syntax. The consistency between different representations has to be ensured by means of abstract syntax. Furthermore, it should be possible to combine several variations of concrete syntax for one view. A diagrammatic representation of a state machine for example may contain textual representations of the transition guards.

  As we place abstract syntax in the centre of language definition, concrete syntax has to be defined as a function that maps an abstract representation of a model into a concrete representation. If this function is bidirectional, it can be employed to provide authoring for the model. Otherwise, it provides just a read-only representation of the model. Note that there may be more than one concrete syntax defined for the abstract syntax.

  There are already some approaches to define concrete syntax on top of an abstract syntax. Textual Concrete Syntax (TCS) provides a template language to define a bidirectional function that maps EMF models into textual representations [Jouault et al., 2006b]. The Graphical Modelling Framework (GMF) provides a language to specify a diagrammatic syntax for EMF models and allows for a generation of an authoring tool from that specification [Eclipse, 2010a]. Diagram Interchange Mapping Language (DIML) provides a language to define a mapping from abstract syntax to a diagrammatic syntax, and a tool architecture to reconcile the diagrams based on model transformations [Alanen et al., 2007]. Most of the approaches towards concrete syntax definition do not provide a distinct separation between abstract and concrete syntaxes. This makes it difficult to define alternative concrete syntaxes for the same abstract syntax.

- ***Process definition.*** *Part of language definition is also the methodical way of modelling, defining at what time which parts of the model need to be developed. For each development phase it defines both which operations are available and what properties need to be fulfilled at the end of the phase. Process definition is interpreted by (and thus parametrises) a workflow engine.*

  A process definition consists of the activities that need to be performed, the roles responsible for certain activities, and the artefacts produced in the course of certain activities. Abstract syntax defines the possible structure of

artefacts, whereas concrete syntax defines the different views to the model. The roles come along with access rights which regulate the access to certain views to the model. Activities may be performed sequentially, in parallel as well as iteratively. For a better overview, activities should be structured hierarchically. A basic activity may be fully automated, such as code generation, and may then be specified by an interpreter of the modelling language. On the other hand, a basic activity may have to be performed manually, such as requirements elicitation, and may then be supported by the operations defined by the modelling language. Furthermore, the transition from one activity to the next may be protected by quality gates which ensure the quality of the activity's result. This can be achieved by integrity constraints or by the execution of complex analyses by interpreters. Integrity constraints do actually not only depend on the modelling language, but also on the progress of the process. Each requirement has to be implemented at the end of the process for example, but is not implemented after requirements elicitation, of course.

- ***Semantics.*** *Generally, there are three ways of specifying semantics: The first one is to describe the semantics of the modelling language by a calculus (axiomatic semantics), the second one is to define the relationship to another formalism (denotational and translational semantics), and the third one is to specify a model interpreter (operational semantics).*

  The first way results in syntactical transformation rules preserving the semantics. It is possible to provide these rules with regard to tool support in the form of refactoring functionality which is being realised via an in-place transformation engine (the original model is thus altered directly). In general, it must be differentiated between postulated rules (axioms) and deducible rules (theorems). In the scope of a language definition, however, axioms would be sufficient in principle. As theorems are, however, generally not deducible in an automated way but are particularly relevant in practice for refactoring, they should nonetheless be formulated explicitly in language definition. From a formal point of view, the syntactic transformation rules complete syntax definition to form a calculus.

  The second way maps each model to a model of another formalism according to syntax definition (referred to as semantic domain). This may be a mathematical formalism such as logic or set theory (denotational semantics), but also a programming language such as C or Java (translational semantics). Note that this kind of semantical definition always depends on another formalism which needs to be formalised itself. All in all, this results in a system of modelling languages which are correlated to each other by semantical mapping. According to our integrated tooling framework, the specified transformation rules are performed by an out-place transformation engine, i. e. the original model is not altered.

  The third way describes how a valid model is interpreted as sequences of computational steps. Afterwards, these sequences make up the meaning of the model. In the context of generic tooling environments, it is therefore possible to use operational semantics to parameterise a generic simulation framework. Kermeta [Drey et al., 2008] is aiming at such a solution.

End of citation.

2.2. The three dimensions of seamless language engineering

In the last section, it was discussed in detail what aspects should be part of a language definition in order to be able to create a comprehensive engineering environment. When talking about a seamless language engineering this is only one of three dimensions – illustrated in Figure 2.7 – in which consistency and integration must be ensured:

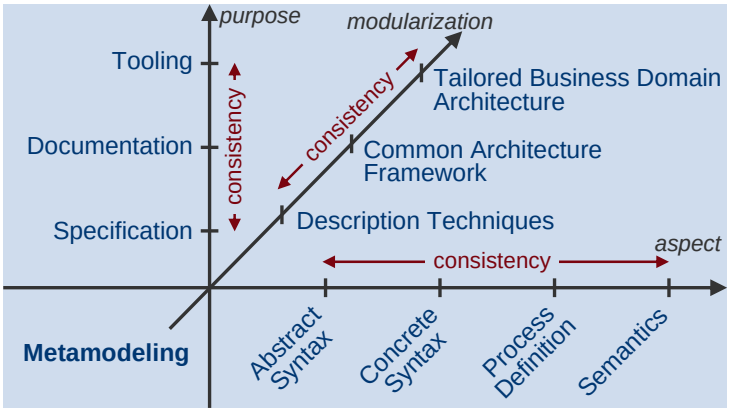


Figure 2.7.: Dimensions of seamless language engineering.

- In the first dimension, (**aspect**) consistency and integration means that the specification of each of the four language aspects fits to each other. If, for example, abstract syntax is described by a MOF diagram and textual syntax by an EBNF, the relationship between these two specifications is not clearly defined (thus resulting in a bunch of glue code which has to be written by hand). The claim for a seamless language engineering is already shown in Figure 2.6 which illustrates that this integration is done by the abstract syntax in the centre. Because of its importance, this dimension has already been discussed in the previous section in detail and no additional section will be addressed to said herein. Please refer to Section 2.1.3, *Vertical tooling aspects: the four aspects of a metamodel*, p. 26 for further details.

- The second dimension focuses on the **modularisation** of modelling languages. When talking about integrated modelling languages beginning at requirements and continuing with system design and implementation up to testing, a extremely huge modelling language arises. Such a language (should) have an internal structure from a more top-level structure down to the detailed basic languages in order to describe concepts such as interfaces or state machines. In order to handle the complexity of such a language it must be decomposed into language modules. A seamless language engineering claims that these language modules fit together in a way that relationships and dependencies are defined in a fine-grained way. Nowadays, the top-level structure is often defined by a file directory structure on the contrary. The detailed models are stored in proprietary file formats and thus, it is nearly impossible to define relationships between elements *within* two of these files.

  Section 2.2.1, *Modularisation Dimension: Steps creating an integrated domain-appropriate modelling language*, p. 32 provides a way of how to build such a language step-by-step by building and using language modules.

- The third dimension concentrates on the **purpose** of defining a metamodel. On the one hand, a metamodel can be used to *specify* a modelling language. When, for example,

an EBNF is defined, the (context-free part of) textual syntax is specified. Note that at the beginning, said EBNF is only a specification. Based on this specification, a more specific input for e. g. a parser generator can be manually created. Often additional information (such as context-sensitive additions of the language) is also necessary for this implementation step. On the other hand, a metamodel can be used to provide a *documentation* of a modelling language. UML class diagrams, for example, are used to describe taxonomies for a concrete modelling language. Finally, a metamodel is directly used to implement a development tool. EMF models are examples used for that purpose in most cases.

A seamless language engineering asks for a strong relationship between these different purposes in order to ensure better consistency and re-usability. In Section 2.2.2, *One language description resulting for specification, documentation, and tooling*, p. 33, an overview of these purposes is provided.

2.2.1. Modularisation Dimension: Steps creating an integrated domain-appropriate modelling language

It is obvious that integrated as well as domain-appropriate modelling languages can be extremely large. Due to that we have to think about how to modularise language definition. One important issue within this context is the re-use of language modules such that different stakeholders can utilise them (maybe even for different purposes). The following three steps will therefore be proposed:

- First of all, the basic language models are built (see *Step 1: defining a set of available description techniques (*basic languages*)*).
- Secondly, the basic languages are combined to form a domain-agnostic integrated language (see *Step 2: relating modelling techniques to a set of engineering views (*common architecture framework*)*).
- Finally, that integrated language is tailored to a specific company or even a product (see *Step 3: tailoring of the common architecture framework (*business-domain languages*)*).

Step 1: defining a set of available description techniques (*basic languages*). During a comprehensive system development process, a huge set of different engineering views becomes necessary. Each of these engineering views can be described by one or more description techniques. All available description techniques are defined in this first step. They can be seen as the basic language modules which are used to build more complex languages. For example, such a basic language may be dataflow networks or automata. It is obvious that the term *basic* is subjective: A language which is built from other basic languages may be used as a basic language itself. Whenever a new description technique is necessary, this set of basic languages is extended.

Step 2: relating modelling techniques to a set of engineering views (*common architecture framework*). Up to now, a set of description techniques has been defined which are (mainly) unrelated to each other. In this second step, the entire set of all engineering views provided is defined. Main focus is laid on the integration aspect: all different engineering purposes during the course of engineering from the problem domain to the solution domain have to be related to each other. Nevertheless, the resulting language called *common architecture framework* is still domain-agnostic and can thus be used in a generic way. It forms the basis

of the tailing within the third and last step. A system comprising three architectural views (functional, logical, and technical architecture), for example, builds a very simple common architecture framework.

Step 3: tailoring of the common architecture framework (*business-domain languages*). The concepts from the generic language, such as *component*, which are defined by the common architecture framework will now be linked to concepts from the business-domain such as *wing*. In addition, the new concepts are more restricted than generic ones as a concept's properties such as *wing* are more specific than those of a concept such as *component*. Tailoring of DOORS according to the ATA chapters would make DOORS a requirements engineering language tailored to the avionics domain for example.

A stepwise domain tailoring from generic to more and more domain-appropriate languages is proposed, which leads to several levels of architectures and architectural frameworks. (The term architecture will be used instead of architecture frameworks when approaching a concrete system to be developed.) Similar to other approaches, the following levels ordered by generality will be distinguished, beginning with the most concrete one:

- **Product Architecture (pA)** defines the architecture for a concrete product (e. g. X3 hybrid car, having 180 PS).
- **Product-line Architecture (plA)** defines the architecture for a specific product line (e. g. X3 series).
- **Organisation-specific Architecture Framework (oAF)** tailors the domain-specific Architecture Framework for a concrete company, such as BMW or Toyota.
- **Industry-specific Architecture Framework (iAF)** adds concepts, which are specific for a dedicated domain (e. g. automotive industry as opposed to avionic industry). The AUTOSAR architecture is an element of this level.
- **Common Architecture Framework (cAF)** defines concepts, which are necessary for any kind of system (e. g. the architecture as is known from software engineering textbooks). Important representatives comprise functional, logical and technical architecture.
- **Meta Architecture Framework (mAF)** is the most generic layer, which is fully independent of any type of system to be developed. It introduces terms such as component, interface, view, viewpoint and concern.

2.2.2. One language description resulting for specification, documentation, and tooling

The result of the previously defined steps in Section 2.2.1, *Modularisation Dimension: Steps creating an integrated domain-appropriate modelling language*, p. 32 is an integrated modelling language covering all necessary views in the course of engineering. Due to its tailoring to a specific company or even a product, it is business-domain-appropriate as well. Due to this, language description already covers the first two categories of conceptualisations, namely *reality* and *purpose*. In addition, it is recommended to use this language definition for configuring both the tools as well as the repository. As our language definition covers all four language aspects now, namely *abstract syntax*, *concrete syntaxes*, *process definition*, and *semantics*, this configuration can be done more extensively. Due to the foregoing, the two remaining categories of conceptualisations, namely *tool* and *repository*, can also be covered.

In summary, the metamodel for integrated and domain-specific modelling language is used as language specification, documentation, and implementation:

Language specification. The specification should provide a precise definition of the modelling language concerning all four language aspects. Target group of the language specification are tool vendors who want to implement a development tool supporting this language. A requirement specification document for such a development tool for example becomes much more precise when adding a formal and detailed definition of abstract syntax. This advantage can be additionally improved when the other three language aspects can also be provided.

Language documentation. The documentation should provide a detailed description of the modelling language concerning all four language aspects. Target group of the language documentation are those engineers who will use the modelling language. A precise taxonomy for the conceptualisations of the modelling language provided can be given by abstract syntax in particular.

Language Implementation/Tooling. Assuming that a generic tool framework exists that is able to interpret the metamodel including all four language aspects, a (basic) language implementation can be derived in a straightforward manner. Such an approach would ensure a break-down of the tyranny of tools as both the tools as well as the repository will now always be based on the conceptual language definition itself.

2.3. Requirements to a metamodelling language

Before beginning to define the metamodelling language M2L, the requirements to such a language shall be discussed. On the one hand, these requirements result from previously discussed ideas about metamodelling and seamless language engineering in particular. On the other hand, requirements result from defining concrete modelling languages using existing metamodelling approaches such as MOF or KM3. During these activities many weak points of those approaches come up and result in these requirements. They are basically classified in the following six main categories:

- **Simplicity.** Using the metamodelling language should be as simple as possible. Especially as it should not only specify a language for implementation but should also be used as a *readable* documentation.
- **Formality.** The metamodelling language has to provide a clear semantics for all syntactical elements of the metamodelling language. This forms the basis for using this language for formal specifications as could be done by a formal grammar.
- **Homogeneity.** All four metamodel aspects (abstract syntax, concrete syntax, process definition, and semantics) must fit together in a formal way. A fully automated tool generation is impossible without a homogeneity like this.
- **Expressiveness.** Informally, each language to be described can be described by this metamodelling language. If a metamodelling language does not fulfil such a requirement, every use of this metamodelling language holds the risk that it will be impossible to express desired future extensions.

- **Appropriateness.** Those metamodelling language support mechanisms necessary from a methodological point of view. Here, for example metamodel evolution and metamodel modularisation are important issues.
- **Autonomy.** The metamodelling language is independent of a concrete tool implementation platform. EMF for example is closely related to Java and Eclipse which makes it difficult to use EMF in other environments such as C#.

In the following, each of these categories shall be discussed in more detail.

2.3.1. Simplicity

First of all it must be noted that the question whether a modelling language – in this case a metamodelling language – is *simple* or not, strongly depends on the background of the respective person. The different technical spaces in metamodelling (in particular XML, ontologies, relational databases or MOF) are a good example. For an XML expert, for example, writing down an XML scheme definition is simple. Defining a set of SQL create table statements might be much more difficult for him/her. If the respective person is a database expert it will be exactly the other way round.

Modern language engineering affects each of the different technical spaces because an integrated development environment requires them all: Databases are necessary for persistence, XML for data exchange, ontologies for taxonomies, and so on. So when talking about simplicity of a metamodelling language, the average of all different specialists has to be considered. This point is also much concerned with appropriateness as described in Section 2.3.5, *Appropriateness*, p. 39. In detail, the following is desired:

Common issues are easy to express. When defining a modelling language there are a lot of issues you come across every day. Examples thereof are definition of orders or multiplicities, using characters and numbers, but also issues such as namespaces or concepts as regards re-use. All these day to day issues must be easy to model. It is also important that this can be done in an explicit way such that another person, but also code generators, can easily recognise these concepts and then act in an appropriate way.

Rare issues may be more complex to express. On the other hand, we can think about a huge number of issues that are more specific for a dedicated language. An example may be the use of white-spaces in order to influence semantics.

So as to avoid extending the metamodelling language, the addition of special solutions for such special cases should be avoided. Otherwise the complexity of the language will increase dramatically. An applicable metamodelling language should nevertheless allow for a handling of such issues, although it might be a little more complicated to define said as there is no special construct for it.

Homogeneity of resulting languages. Simplicity not only concentrates on tasks when defining new languages but also on using the newly defined languages. In this context an important issue is the homogeneity over a set of languages created. When coming across a canonical name, for example, it should always be presented the same way (for example using a dot to separate elements of the name). If things like these are not uniform for different languages (e. g. one uses a dot in canonic names, others use a slash or a backslash), a user

gets confused when using more than one language. This is particularly true when a large integrated language is created by combining a set of sub-languages.

No redundancy. Languages often comprise more than one way to express a special issue. One common example is to allow a user to skip a parameter resulting in a default value. An example from the metamodelling domain is skipping the lower limit of multiplicity. Thus, a valid multiplicity is [1]. Problem occur upon reading such a multiplicity as the question arises whether this notation means [0..1] or [1..1]. As such a short-cut only serves three characters, it should be avoided. This procedure does not only prevent misunderstandings but also forces the modeller to think explicitly about the lower limit. Another example is a reserved word *notnull* to indicate that the lower limit is at least 1. Again, such a construct does not simplify modelling but only extends the complexity of the modelling language.

Multiple views. In order to fulfil the requirements of different types of users of a metamodelling language, the metamodel specification should be capable of being presented in different ways. These views should in particular cope with the three metamodel purposes, namely specification, documentation, and tooling. There should be both a graphical as well as a textual representation for example. One textual representation may be a very formal and short one which is tailored especially to metamodelling specialists. Another textual representation is a little longer in representation as more reserved words are used instead of symbols. It contains also all information of the short representation but can be read a little more intuitively because of the use of reserved words. A graphical representation may focus on the taxonomical content and thus may skip all formal details.

At first sight, a call for multiple views contradicts the request for no redundancy. Indeed, the call for multiple views results in redundancy. The difference is that here, redundancy is performed in a very explicit way and focuses on different types of users. When concentrating on one view there should still be no redundancy.

2.3.2. Formality

In the past, one of the major weak points of metamodelling approaches was the lack of formality. Whenever a metamodel is depicted in the form of a UML-like diagram it always brings along its informality. This is also one of the major reasons why the present thesis came actually into being. Note that formality does not only include syntactical parts (which would only result in semi-formality) but also semantics.

Formal syntactical definition. The basis for each formal definition is a formal syntax. Although a diagrammatic syntax can even be formal, a textual representation is often much easier to handle when talking about formal syntax. In particular if the number of concepts rises in order to capture all details of the metamodelling language, a textual syntax becomes essential. (Note that in most graphical representations the textual portion is also significant.) In addition to large metamodels containing a lot of concepts, a textual representation is more useful. Nonetheless an additional graphical syntax is helpful.

Formal semantics. While formal syntax only defines the structure of a language nothing is stated about its meaning. Therefore a formal semantics becomes necessary. Note that in the context of metamodelling (as we are concentrating on abstract and concrete syntax in the present work), semantics is a pure structural semantics, thus only defining the impact

of the metamodel on the *structure* of a model. Semantics of *process definition* defines how the structure of a model evolves over time. Even the semantics of the semantical definition within a metamodel can be reduced to a model-to-model transformation which is also of syntactical nature.

Once a formal semantics exists, the door is open for various topics: First, as there is a clear understanding of what a concrete metamodel means, it is much easier to abstract from a concrete implementation (and thus tooling). Not till then a metamodelling language can be used as a basis for multiple implementations and thus allows to switch more easily between different tool frameworks. Secondly, retrieving information from a large model, e. g. within a database, can be optimised by a formal query optimisation. Thirdly, arbitrary formal proofs can be performed as e. g. check for language emptiness (which is important for combining languages in particular). Moreover, a disjunction and conjunction of languages can afterwards be defined easily which is e. g. important for language evolution.

Bootstrapping. Another important aspect of a formal metamodelling approach is the bootstrapped way of defining the metamodelling language itself. Upon that, the metamodelling language is entirely defined by its own constructs. Thus, no other mechanism for defining a language is necessary than the one defined itself. This approach has an intrinsic problem, of course: The definition is useless as the definition itself is required to understand the definition. This is again one of the major issues to be solved when formalising a metamodelling language. Although a bootstrapping definition is sensitive and desired, there must be a mathematical way of defining the metamodelling language without using the language itself. One of the crucial points of the present work is to find a suitable solution for the above-mentioned problem.

2.3.3. Homogeneity

It has already been assumed that a language specification consists of four aspects, namely *abstract syntax*, a set of *concrete syntaxes*, *process definition*, and *semantics*. Homogeneity claims that these four aspects are tightly connected to each other as already discussed in Section 2.1.3, *Vertical tooling aspects: the four aspects of a metamodel*, p. 26 and Section 2.2, *The three dimensions of seamless language engineering*, p. 31. Thus, the relationship between the four aspects is clearly defined and redundancies are minimised. As the multiplicities of properties should have already been defined within the abstract syntax, a concrete syntax does not need to define said again. Operators such as * or + as, known from regular expressions, will not be necessary when defining a concrete syntax.

Abstract syntax in the centre. In compiler construction textual syntax was always the central representation of a model. The abstract syntax tree only was a means to an end. When time diagrammatical syntaxes came up, abstract syntaxes became more important. Once alternative concrete syntaxes are required, a common abstract syntax is indispensable. An abstract syntax concentrates on the concepts of a language provided and their respective relationships independent of a concrete notation. As both aspects, namely *process definition* as well as *semantics*, can (and should) be described independently from a concrete notation, abstract syntax is the right and common abstraction from various concrete notations. In a homogeneous metamodelling language, abstract syntax holds together all language aspects. Please refer to Figure 2.6 as well.

Concrete syntaxes based on abstract syntax. As already mentioned, an EBNF describes a textual language in such a comprehensive way that no other additional specification will be necessary. Thus, having an EBNF, it can be decided whether a textual representation is part of the language or not – assuming of course that the language is context-free. If a language contains context-sensitive restrictions (as most languages do), a context-free skeleton of the language may nonetheless be obtained. On the other hand, the relationship to an abstract syntax is not obvious in such a scenario. Here, in most situations some kind of glue code will therefore be necessary. As in addition some coding information (such as multiplicities) is redundant from a syntactical point of view, abstract syntax and EBNF may be entirely contradictory.

When talking about a homogeneous metamodelling language, concrete syntax should be entirely based on abstract syntax. Thus, only the delta information is coded within concrete syntax. When following these specifications, redundancies will be completely avoided on the one hand, and on the other, the relationship between abstract and concrete syntax will be defined in such an obvious way that no glue code will be necessary at all. As a result of the tight integration of abstract and concrete syntaxes, the more powerful concepts for context-sensitive restrictions of abstract syntax can be used for concrete syntax. Note that because concrete syntax relies on abstract syntax, concrete syntax itself does not need to define a language at all (such as an EBNF does).

Process definition based on abstract syntax. Nowadays, a process is defined in many situations without talking of any concrete product model (and thus a metamodel). Thus, the artefacts involved in such a process definition are defined informally to a large extent. These situations make it difficult to operationalise such a process definition in a straightforward way.

In a homogeneous metamodelling language, the artefacts involved in the process definition should rely directly on the concepts defined from abstract syntax. If a suitable constraint language is at hand, process quality gates can be defined. Thus, depending on a concrete process phase, a set of corresponding conditions has to be valid. (The implementation phase cannot be left until each requirement is implemented.)

This approach also leads to a new understanding of activities within a process definition: Activities can be defined in a formal way by dedicated in-place transformations. Thus, an activity is nothing but a dedicated altering of the model (which, in turn, is an instance of the metamodel). Top-level and thus coarse-grained activities can be refined by more fine-grained activities. In the end, basic activities such as *Adding a new state to an automaton* or *Renaming a state* will be described. Note that each of these basic activities is linked to more high-level activities and thus cannot be invoked any time but only within the process phases intended for them.

Semantics based on abstract syntax. Here, the term *semantics* shall not be discussed in a highly scientific way. It shall be noted that semantics should be described in a way that corresponds to abstract syntax and not – as common in the past – to textual concrete syntax. In practice, semantics is often described by transforming a language to another – already known – language as e. g. compilers or code generators do. Thus, formal compilers and code generators are just transformation engines.

The present thesis wants to claim that they operate solely on abstract syntax. Nowadays, the input for common compilers represents a textual file, and the output of code generators in turn are most often textual files. The desired scenario for a homogeneous metamodelling language is that both source and target language are described by abstract syntax as claimed

by the metamodelling language (and some concrete syntaxes in addition but right that should not be of any importance at that point). Then, semantics is simply defined by a model-to-model transformation which transforms a given source model to a corresponding target model – wherein both models are based on an abstract syntax level. Note that this transformation is called out-place as the source model itself is not altered but a new model is created.

Another important way of defining semantics is to provide transformation rules which preserve semantics. Formally, the result is a calculus. Here again, these model-to-model transformation rules are claimed to be defined on the basis of an abstract syntax. This time it is an in-place transformation as the source and target model, upon which these transformations operate, are the same: The transformations alter the model directly. Note that such transformation rules have a highly practical field of application: They can be used in a straightforward way for refactoring issues.

2.3.4. Expressiveness

Requirements on expressiveness are often difficult to formulate. A simple answer would, of course, be to require Turing completeness. In most cases, however, this is much too general and leads to other problems such as undecidability. When talking about expressiveness from a language engineering point of view, two extremes will have to be considered: The first one is to handle everything by semantics. The second is to handle as much as possible by syntax.

Now, a simple example for this from compiler construction (in particular the programming language C) will be provided: The former one states that any arbitrary combination of characters is a valid C program (thus the syntax states nothing). In such a situation, semantics has to care about everything. The behaviour for every meaningless combination of characters is supposed not to terminate. For all the correct C programs (the term *correct* is informal here) known semantics is defined. The latter one may state that a C program is only valid from a syntactical point of view if the program terminates. This time the syntax cares about a maximum of restrictions and thus semantics only has to care about the valid C programs. Obviously the syntax of the former one is not useful as it states nothing and the latter one is not useful as it is undecidable. Because of this situation, in practise the truth is somewhere in the middle. This vague border between syntax and semantics may also be the reason why the context-sensitive, *syntactical* characteristics of a language are often called *static semantics*.

After this discussion, the questions comes up of how much should be covered by syntax and what should remain in the field of semantics. A very informal answer may be: All that can be checked within a short time such that it can be e. g. visualised within an editor within seconds. Thus, in a formal way it can be said that syntactical restrictions must be 1) decidable and 2) of a complexity of at most e. g. $O(n^3)$. A suitable requirement could be that the metamodelling language allows an exact modelling of context-sensitive languages.

2.3.5. Appropriateness

Appropriateness is the only reason why we suffer about languages at all. Otherwise, Turing machines can be used for every purpose. In the present case it must be ensured that the metamodelling language supports engineering of new languages even from a methodical point of view. Important issues within this context are:

Intuitive for language and product model engineers. As computer languages have been designed as long as computers are invented themselves, there are of course various techniques for describing such languages. Up to now, these languages have been used by language engineers as well as product model engineers. When introducing a new metamodelling language it should not neglect these already existing languages although the new language may have great advantages. The power of common knowledge along with widely-proven concepts must not be underestimated. Because of that, an appropriate metamodelling language should combine the good and proven concepts of earlier metamodelling techniques. These are UML class diagrams, in particular MOF [OMG, 2006a], the Standard Query Language (SQL) [ISO, 2008], the Extended Backus-Naur Form (EBNF) [ISO, 1996], the programming language Java [Microsystems, 2010], and the Object Constraint Language (OCL) [OMG, 2006b].

Suitable for creating models prior to a metamodel. In language engineering it is often vital to define a model before having defined any metamodel. This advantage comes up upon defining a textual or in particular an XML language. Basically, a semi-structured paradigm of the underlying formalism is claimed. An XML file can be written down without having any XML scheme definition. This might be particularly helpful in early stages of language engineering. Language engineers, for example, may show the customer a prototype model and may thereupon validate the requirements for the language to be designed. These prototype models may help even the engineers themselves a lot to get a better understanding. Note that this advantage it not a matter of course: In traditional metamodelling environments, such as in Eclipse EMF [Budinsky et al., 2003], it is not possible to create a model without a metamodel.

No "contamination" of technical details. When claiming that a metamodel should be usable for all three purposes, namely *specification*, *documentation*, and *tooling*, the metamodelling language must be powerful enough such that the most detailed metamodel (which is intended for the purpose of tooling) is not contaminated by technical details. Note: The fact that there may be various views of one and the same metamodel having different detailing levels for documentation, specification, and tooling does not contradict this requirement.

If nowadays re-use concepts shall be implemented within a language, these attempts often extend the metamodel until it becomes unreadable. Here, the metamodelling language would not be sufficient for serving both documentation and implementation.

Because of that, a metamodelling language should support concepts such as instantiation, namespaces, ordered sets, or a closure operator. As such concepts are often syntactical sugar, this issue leads to a trade-off to the simplicity of the metamodelling language (see Section 2.3.1, *Simplicity*, p. 35).

From rough to detailed specification. Developing a comprehensive modelling language is a tough and time-consuming process. In order to structure such a process in a much clearer way, language engineering should be performed step-by-step. In the first step, for example, the concepts involved will be defined. In the next step the relationships between the concepts will be specified. Then the multiplicities will be defined, and so on. Upon that a language becomes more and more precise over time. Finally, the very specific constraints will be added to the language specification. Such a stepwise language engineering should be fully supported by a metamodelling language.

Customising of canonical concrete syntaxes. As claimed in Section 2.3.3, *Homogeneity*, p. 37, concrete syntax definitions are based on abstract syntax. Thus, abstract syntax must be defined first. In order to start testing the language by creating first models, it is necessary to have a concrete syntax in very early stages of language engineering. At this stage, the definition of a concrete syntax does not make sense, which renders a canonical concrete syntax extremely useful. Such a canonical concrete syntax may be of textual, diagrammatical or tabular nature. It is named canonical because it can be derived from abstract syntax without any additional concrete syntax specification. One possible way might, for example, be bracketed property value pairs in the form of a textual syntax. Based on that, it becomes very easy to write down first models.

In order to allow a migration from canonical concrete syntaxes to individually defined concrete syntaxes, the individually defined ones should be handled as customisations of the canonical ones. Then, an individual syntax can evolve from the initial canonical one step-by-step.

Language modularisation. In order to operationalise an integrated model theory for practice, a company may aim at defining an integrated modelling language covering the entire development process. As a consequence, such an integrated modelling language is quite extensive and thus difficult to develop. The development costs are reduced by developing an integrated modelling language that can be re-used for several companies. This approach is, however, usually not feasible, as a company may request a modelling language tailored to its specific needs.

Nevertheless, integrated modelling languages of different companies will be identical in some parts or similar in others. Many automotive companies, for example, prefer to use dataflow networks for modelling embedded systems. Re-using these parts can be achieved by modularising modelling languages. An integrated modelling language will then be built by a number of predefined modelling language modules.

A modelling language module consists of the elements that have already been described: abstract syntax, concrete syntax, process definition and semantics. In addition, a modelling language module must provide an interface so that it can be connected to other modules. A module for modelling software design provides e. g. a connector for deployable units which can be connected to a suitable connector in a module for modelling deployment on hardware. As abstract syntax is placed in the centre of language development, the interface of a module is defined in terms of abstract syntax.

Furthermore, companies might want to adapt a modelling language module to fit their specific requirements. Because of the associated costs, companies do not want to rebuild the adapted modelling language from scratch. For this reason, a means has to be provided that allows for the customisation of modelling language modules. There are several possibilities for fulfilling this purpose: a language can be customised by constraints (lightweight extensions), sub-concepts (heavyweight extensions) or parameters of the module.

Supporting language evolution. With appropriate support for language evolution, modelling languages can be developed in an evolutionary way. A version of a modelling language is created and deployed to be assessed by the modellers. The feedback of the modellers can then easily incorporated in a new version of the modelling language which, again, will be deployed for further assessment. Elements of a modelling language other than abstract syntax do not have to be defined in a first version of the modelling language, but will be completed in later versions. By way of an evolutionary development of modelling languages, domain appropriateness can thus be reached iteratively.

In order to be prepared for the inevitable evolution of modelling languages, appropriate support is required to safely change or extend a modelling language when already deployed [Herrmannsdoerfer et al., 2008]. Abstract syntax will be modified first to meet the new requirements. As the other elements of a modelling language all depend on abstract syntax, they need to be adapted to the modified abstract syntax and maybe extended with respect to the new requirements. It is, however, most important to migrate existing models so that they can be used with the modelling language evolved.

As there may be a large number of models, model migration has to be automated. Further automation can be provided by reusing recurring migration scenarios. However, model migration becomes quite complex, when it is motivated by changes in the semantics of the modelling language. For this reason, appropriate tool support also needs to account for manual, expressive migrations.

Suitable for generating comprehensive development tools. As described in Section 2.1.3, *Vertical tooling aspects: the four aspects of a metamodel*, p. 26, a language specification should in particular be suitable for configuring a generic tool framework such that the result is a comprehensive development tool in the form of an integrated model engineering environment. To achieve the foregoing, the model-based engineering paradigm is applied to the development of development tools themselves. Note that it will not be distinguished between configuring a generic tool framework (metamodel is being interpreted) and generating a development tool (metamodel is being "compiled").

In order to achieve this requirement, all necessary information for tool generation must be extracted from the metamodel. No additional information sources should be present simultaneously. Information of how to pretty print a textual representation for a given abstract syntax must also be defined for textual concrete syntaxes. Up to now, it has been uncertain whether such a requirement could be achieved at all.

2.3.6. Autonomy

As described above, a metamodel should be suitable for creating a concrete tooling environment. On the other hand, it is important to remain independent of a concrete tooling environment. Eclipse EMF, for example, focuses on both Java and the Eclipse framework. Even if these technologies are wide-spread, other technologies such as *Visual Studio* and *C#* by *Microsoft* and *Java Netbeans* by *Sun Microsystems* are also available. An integrated development platform has to support all different technologies and must thus be independent of a concrete one. The modelling language for example must not be based on constructs provided by specific platforms such as basic data types as they may be different from one platform to another. Note that the requirement of a general autonomy does not demand more than one implementation at the beginning. It only states that another implementation on another platform is possible without restrictions.

2.4. Procedure specifying the (self-describing) metamodelling language *M2L*

As described above, a metamodel consists of the four aspects called *abstract syntax*, *concrete syntax*, *process definition*, and *semantics*. As the present work will concentrate on the first two aspects, namely *abstract syntax* and *concrete syntax* (in the case of concrete syntaxes

the present thesis focuses on textual concrete syntaxes), the procedure of how to formally define the metamodelling language *M2L* shall be described in the following.

M2L simply stands for ***M**eta**m**odelling **L**anguage* and is structured into abstract syntax and concrete syntax. *M2L* itself also is – as the name already suggests – a language that needs to be well-defined. As *M2L* represents the language used for describing languages, it would be necessary to already use the language *M2L* for describing itself (bootstrapping). This cyclical language description leads to the commonly known exceptional position, always taken by metamodelling languages: The metamodel describing the metamodelling language itself is often called meta-metamodel and is characterised in that it describes itself by its own means. Cyclical language definition has up to now not yet been entirely solved in the field of metamodelling, also resulting in approaches within metamodelling which are described in a rather semi-formal way.

The formalisation which will be presented in the following, allows for a cycle-free language definition. It is mainly based on decoupling a model from its metamodel. In current approaches within metamodelling, models cannot exist without their respective metamodel. The most explicit formulation of the dependency of models to metamodels is given in the formalisation of KM3 in [Jouault and Bézivin, 2006]: "A model $M = (G, \omega, \mu)$ is a triple where [...] ω is a model itself (called the reference model of M)." Thereby ω represents exactly the metamodel. In contrast to that, the model concept will be defined independent of another model, and in particular independent of a metamodel as well. This procedure allows for a definition of the model concept independent of metamodels. In the end, this procedure will be adopted from the field of formal languages: The set of all words will be defined entirely independently of a concrete language. Only an alphabet will be assumed. This concept will be transferred in the form of abstract words to our needs. The respective alphabet will be composed of a given set of concepts C and properties P.

A strict separation between model and metamodel is also closely correlated to the concepts of XML: The term well-formedness allows for a definition of an XML document – which would correspond to a model – without an XML scheme or a DTD having been described. At the same time, a superset of any kind of languages possible will therefore be defined by the set of all well-formed XML documents. Only in a second step will the valid XML documents be marked as valid words of the language by a specific XML scheme.

The problem of the cycle-free language definition does, however, not only come up in connection with a definition of abstract syntaxes. A similar phenomenon occurs upon definition of a concrete textual syntax for *M2L*. The concrete textual syntax can only be defined as soon as the second part of *M2L* has been defined for the definition of concrete textual syntaxes. Due to that, focus shall at first be on the abstract syntax of *M2L*.

As shown in Figure 2.8, the following stepwise definition of *M2L* will result:

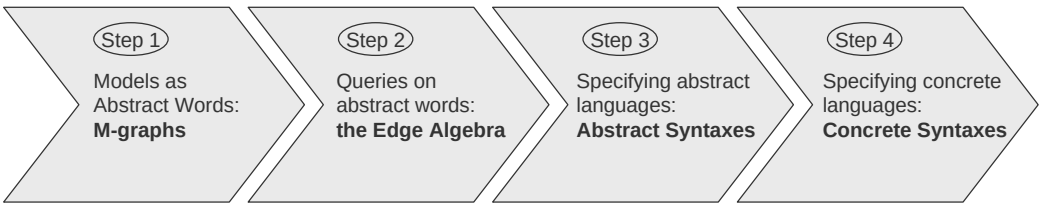


Figure 2.8.: The four steps of specifying M2L.

1. **Models as abstract words: M-graphs.** After some mathematical basics, a formalisation of models will first be given. In the style of formal languages they are

called *abstract words*. The formalisation is done by a special type of directed, labelled multi-graph, which is called *model-graph* or just *M-graph*. (See Chapter 5, *Models as Abstract Words*, p. 81.)

2. **Queries on abstract words: Edge Algebra.** Once the M-graphs are introduced, an algebra on edges of such graphs will be defined. This allows for a powerful inspection of models including information extraction and constraint valuations. Until now, metamodels have not been mentioned at all – we were just talking about models. (See Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95.)
3. **Specifying abstract languages: Abstract Syntaxes.** Based on Edge Algebra, it will now be possible to define the meaning of abstract syntaxes by formulating constraints on abstract words. Thus, an abstract syntax can be completely transformed to an according Edge Algebra statement. As a concrete syntax is not available up to now, a canonical textual syntax will be introduced. Later on, this canonical textual syntax can be customised as requested in Section 2.3.5, *Appropriateness*, p. 39. (See Chapter 7, *Abstract Syntaxes in M2L*, p. 119.)
4. **Specifying concrete languages: Concrete Syntaxes.** Finally, it shall be defined how to specify textual concrete syntaxes using *M2L*. As the concrete syntax of *M2L* cannot be used because it currently being defined, every step must be performed on the basis of abstract syntaxes and abstract words. (See Chapter 8, *Textual Concrete Syntaxes in M2L*, p. 153.)

Up to now, the metamodelling language has not been described in a bootstrapping way. This important step will finally be performed in Chapter 9, *The overall specification of M2L*, p. 171. The overall metamodelling language *M2L* will therefore be postulated as a known language and thus this step cannot be performed until the overall specification has already been provided.

Chapter 3

Running example: Modelling dataflow algorithms

In order to get a better impression of what the benefits of a suitable metamodelling language are, a suitable running example shall be introduced. It should go along with all explanations and definitions throughout the following chapters. The present thesis is not aiming at providing a proof that the metamodelling language *M2L* is suitable for defining all different kinds of languages. Instead, the reader should get a clear idea of the strengths and weaknesses of *M2L*.

First of all, an overview of the criteria for selecting a suitable running example will be provided. Secondly, the industrial project context for dataflow diagrams will be introduced to provide an understanding of practical relevance. Having given an informal description of the language to be defined, a first semi-formal metamodel in the form of a UML class diagram will be introduced in order to provide a first structural understanding of abstract syntax. Finally, two concrete dataflow models will be introduced as examples, followed by a set of issues that shall be expressed by a formalised metamodel such as *M2L*.

Contents

3.1. Criteria for selecting a suitable running example

When selecting a running example, it must be discussed what example the right choice for such a exemplary language might be. For the present work, the following five (contradictory) aspects are taken into account in order to find a suitable running example:

- **Widely known kind of language.** The reader should easily understand the purpose of the language. There should be as less explanations necessary as possible. Focus is not on the language but its language description. Thus, the reader should not be confronted with a completely unknown domain. As the target audience is independent of a particular modelling domain, only few kinds of languages are feasible. A viable language is for automatons, of course. Nevertheless, not all relevant issues can be illustrated.

- **Manageable in complexity.** The complexity of the given exemplary modelling language should not be too complex. Thus, as few concepts as possible should be defined within the language. In case the exemplary language is too complex, the detailed concepts will divert from essential aspects. This is the reason, why the present running example will do without data-types.

- **No trivial examples.** Although complexity should be manageable, trivial examples are not what the present thesis is aiming at. These examples are not reliable at all. Moreover, it will be impossible to show all issues within one running example in case of trivial examples. It will then also be not feasible to show the collaboration of different aspects of modelling a language.

- **Practical relevance.** The running example should be of practical relevance. If we are able to demonstrate the metamodelling concepts by a practically relevant example, the metamodelling concepts themselves become of practical relevance. Although this procedure cannot be seen as a profound proof for practical relevance, it does, however, provide an indication for it.

  In addition to this running example, the bootstrapping mechanism of *M2L* leads to the meta-metamodel of *M2L*. In contrast to the running example, this meta-metamodel defines a complete and ready-to-use language (namely the metamodelling language itself). And thus, it forms the proof-of-concept for *M2L* in the present work. For further details please refer to Chapter 9, *The overall specification of M2L*, p. 171 and Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291.

- **Ability of demonstrating relevant issues.** *M2L* offers a substantial set of extended metamodelling concepts. In order to demonstrate of how they are to be used, this running example should be utilised. All different metamodelling concepts should thus be able to be shown by one suitable running example. This is, of course, not always true. Although the metamodelling language *M2L*, for example, builds a complex and ready-to-use modelling language, not all concepts are necessary to describe its meta-metamodel.

When considering all those five aspects, the most suitable modelling language for being our running example is a language for modelling dataflow algorithms. This language will be introduced in the following sections.

3.2. Industrial project context

Designing software-intensive systems – e. g. in a vehicle – is subject to particular challenges. Such issues are described in [Broy, 2006] and [Grimm, 2005] in particular. First of all, an insight in these issues shall be given in Section 3.2.1, *Challenges in designing software-intensive systems*, p. 47. In Section 3.2.2, *The three architectural layers*, p. 47, the generic architectural approach as presented in [Broy et al., 2008] shall be introduced. Finally, the

concrete relevance of dataflow diagrams shall be discussed in Section 3.2.3, *Relevance of dataflow diagrams*, p. 48.

3.2.1. Challenges in designing software-intensive systems

A high percentage of competitive-relevant innovation in automotive industry is based on software. This brings along an increasing number of functions being distributed via a heterogeneous network of electronic control units (ECUs). The complexity of developing the functions is increased due to the intensive distribution of these functions. At the same time, it is harder to detect undesired dependencies of the functions among each other.

Moreover, cost pressure within the automotive industry remains very high. This results in the fact that cost-effective hardware is used, which does not only have a tight memory size but also a very poor processing speed. Some of the functions in the vehicle need to fulfil tough real-time requirements, wherein two contradictory requirement classes oppose each other which need to be met both, however.

Re-use and variability also play a pivotal role in automotive industry. On the one hand, there are basic functionalities such as controlling the power window lift, which can be found in all vehicles of an automotive manufacturer, and, on the other hand, there are variable functionalities, which can be found – depending on the equipment version selected – at various stages of design in individual vehicles, such as in the park distance control system. In both cases the level of re-use is intended to be kept as high as possible, e. g. to avoid that functionality will have to be entirely re-designed and altered respectively with enormous effort for each equipment version and each possible hardware platform.

The specifics mentioned are not only valid in automotive industry. Most challenges in designing software-intensive systems in a vehicle can be transferred to many other domains using embedded systems. Emphasis can be placed on manufacturers of mobile phones in this context. Although a mobile phone does not dispose of a heterogeneous electronic control unit network, its complexity is correspondingly high due to the enormous multi-functionality and thus the risk of undesired interactions between functions is substantial. Here as well the systems need to be highly efficient, e. g. small code size and compliance with real-time requirements, although cost-effective hardware resources represent a requirement in this domain as well.

3.2.2. The three architectural layers

Figure 3.1 presents an overview of the architectural model for software-intensive systems. It is basically subdivided into the following three architectural layers: *Functional Architecture*, *Logical Architecture*, and *Technical Architecture*. The level of abstraction decreases herein from top to bottom.

The layers were separated such that the special challenges in developing software for embedded systems and in particular those for the automotive industry can be met. Using these models, the aspects of individual challenges can be described in a particularly good manner.

- **Functional Architecture** offers models which allow for a formalisation of functional requirements, representing them in the form of hierarchical structures, additionally illustrating dependencies between these functional requirements. Functional architecture therefore provides the basis to detect undesired interactions between functions at an early stage of the development process. Thanks to this basis and its high level of

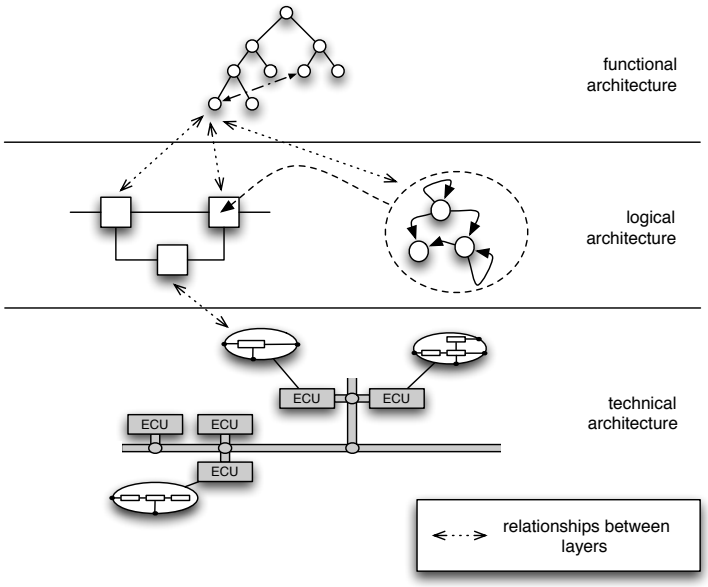


Figure 3.1.: The three architectural layers in software-intensive systems [Broy et al., 2008]

abstraction, functional architecture is also well suited for an extension by product line concepts.

- **Logical Architecture** offers models which allow for a structuring of functionality into domain-specific components. The functional requirements formalised in functional architecture are realised by a network of hierarchical components, which are, however, independent of the underlying hardware. The model of the system on the layer of logical architecture can be executed and simulated. It is thus amenable to an earlier architectural verification. Due to the modular design and this layer's independence of hardware, the complexity of the model is reduced and a high potential for re-use is created.
- **Technical Architecture** finally describes the realisation consisting of hardware and software in an abstract way. It offers suitable models which describe the behaviour of hardware and software uniformly and which allow to describe the influence of the hardware used on the behaviour of the entire system. Here, the level of abstraction is chosen such as to make it possible to conclude whether real-time requirements can be met and such that at the further transition from technical architecture to implementation only software-technical transformations will take place, but no modifications of the behaviour.

A more detailed picture including the link to a concrete tooling support can be found in [Broy et al., 2010].

3.2.3. Relevance of dataflow diagrams

For each of the architectural layers specific modelling techniques exist which are – at best – connected to each other by an integrated modelling language. Among the most essential language elements are dataflow diagrams. Strictly speaking, it must be distinguished be-

tween two types of dataflow: weak causal and strong causal ones. This fact is, however, of minor importance for our interests within the present work. A system in each of these architectural layers is basically hierarchically decomposed into subsystems for implementing the divide-and-conquer principle. Systems (and subsystems) in particular dispose of a syntactical interface which again consists of a set of input and output ports. As these ports are connected by channels, this results in (various types of) dataflow diagrams.

In [Sutherland, 1966] Sutherland created one of the first graphical dataflow programming frameworks. Nowadays, most of the modern system engineering languages use dataflow networks because of the reasons mentioned above. Formal foundations are Lustre [Caspi et al., 1987, Halbwachs et al., 1991], Signal [Houssais, 2002], and Focus [Broy and Stølen, 2001]. Scientific and industrial implementations are Matlab/Simulink [Mathworks, 2010], Scade [Esterel, 2010, Scade, 2010, Berry et al., 2000], AutoFOCUS [Hölzl, 2010], as well as the Component Language (COLA) [Kugele et al., 2007].

The running example within the present work depends on the Component Language (*COLA*). COLA is an integrated modelling language for embedded reactive systems providing specification techniques for requirements, system design, implementation, and test. The language has been developed for the automotive industry and its specification is entirely based on the theoretical foundation. Via *OOMEGA* [OOMEGA, 2010], this formal language is available as an eclipse-based [Eclipse, 2010b] tool including a common database back-end.

As COLA represents an extensive language, focus shall be on a very small and simplified part of COLA in our example: dataflow networks. For our purposes it is not necessary to go into details of COLA. Instead, a short and informal explanation of what our exemplary language does shall be provided. In order to get a clear understanding of the syntax and semantics of COLA, please refer to [Kugele et al., 2007].

3.3. Informal description of the modelling language

As already mentioned above, the exemplary language, which will be further used as running example, is suitable for modelling dataflow networks. According to *COLA*, a weak-causal semantics forms the basis. The processing units within such dataflow networks are called *components* and may again be networks. Due to that, a system can be hierarchically decomposed into sub-components (and thus subsystems). A component which should not be decomposed any further is called a *block*. Arithmetic operators, such as an addition or multiplication, are typically modelled as blocks.

Each component (i. e. both networks and blocks) comprises a syntactical interface which is called *signature*, which again consists of a set of input and output ports. For simplicity our exemplary language is – in contrast to *COLA* – untyped. Hence, a port is simply described by its name and does, for example, not include a data type.

Ports of the signature as well as those of its sub-components are connected via channels within a network. Each channel connects exactly one source port to at least one destination port. Herein, a source port could be an input port of the signature or an output port of a sub-component. Similarly, a destination port could be an output port of the signature or an input port of a sub-component. In order to prevent fix-point arithmetic, cyclic connections are not allowed unless a particular *pre*-block is located in between. This *pre*-block has one input and one output port. It delays input for one cycle.

In order to allow re-use of previously defined networks, an additional differentiation between the definition of a component (i. e. a network or a block) and its (possibly multiple) use within

other networks shall be provided. On the one hand, components can be defined within other components. Such inline-defined components can only be used inside these components. On the other hand, it shall be possible to define components within libraries. Whenever a component from a library should be used in another network this library must be imported. Libraries can be hierarchically structured. Thus, each library may contain sub-libraries. An import of a library should automatically include its sub-libraries.

3.4. A first, semi-formal abstract syntax

Within the present work, a formal metamodelling approach shall be generated to be able to define a precise metamodel for e. g. such a running example. Obviously, this formal approach is not yet available at this point. Anyway, in order to give the reader a more precise idea of the running example, an abstract syntax will be provided in the following by using commonly known UML class diagrams (the MOF subset in particular will be sufficient), as defined in [OMG, 2006a].

In Figure 3.2, the abstract syntax for the previously introduced dataflow modelling language will be defined. It shows all necessary concepts (known as classes in UML), including their relationships and attributes (which will be called properties from now on).

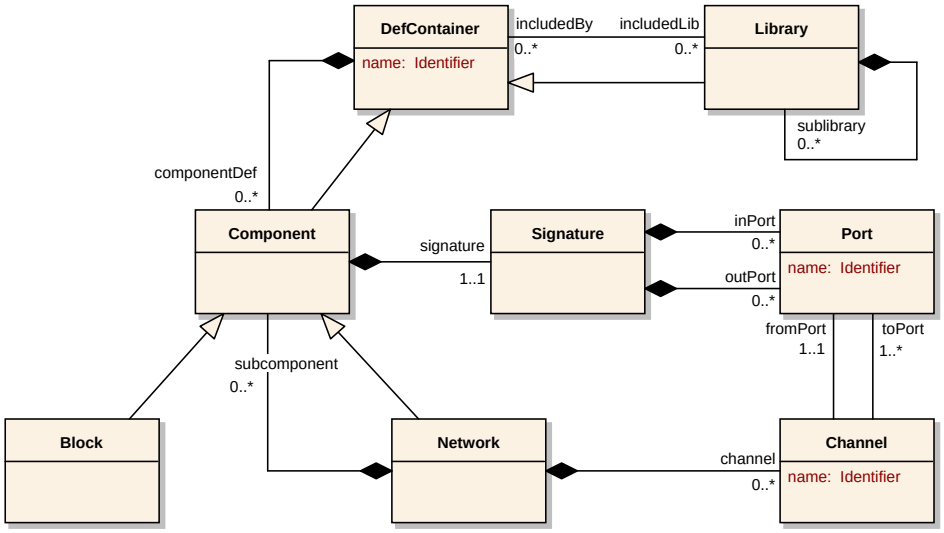


Figure 3.2.: A first semi-formal abstract syntax for the running example

In the following, each of the concepts shown shall be introduced including their properties. Note that here, focus will only be on abstract syntax. As defined above, a metamodel also consists of other elements, e. g. concrete syntax, which will be omitted herein.

As this definition can only be semi-formal at this point, a formal definition will be provided later on by using *M2L*. Section B.1, *Metamodel of the Running Example*, p. 307 provides an overall specification – including concrete syntax as well.

3.4.1. Concept *DefContainer*

DefContainer is a common concept for *Library* and *Component*. It mainly deals with the fact that both libraries and components are able to contain definitions of components. Moreover, it provides the include mechanism so as to be able to use components defined in other libraries within new dataflow networks.

The properties of the concept *DefContainer* are:

- **name:** The name of a *DefContainer* and thus of a library or a component, which may be any kind of a non-empty character sequence. Initially, these names should not be globally unique. Instead, there should be a kind of local uniqueness such that canonical names in the form of <name1>.<name2>.<name3> can be used as globally unique identifiers.
- **includedLib:** All libraries which should be included to be able to use the components defined within them for creating new dataflow networks. If a library is not included, its defined components cannot be used, unless the library is transitively included (i. e. the library is a sub-library of an included library).
- **componentDef:** The set of components which are defined within a library or component. Note that if a component is defined within a network (a network is a component and thus also a DefContainer) it is not automatically part of the network in terms of a sub-component. Please refer to the property *subcomponent* in the concept *Network*.

3.4.2. Concept *Library*

By specialising the concept *DefContainer*, a library is able to contain a set of defined components in order to be used in other component definitions. Libraries can be hierarchically structured and thus may contain sub-libraries. This sub-library relationship has major influence on the include mechanism: When a library is included, all sub-libraries thereof will be included as well. Note that this approach differs from e. g. *Java* [Microsystems, 2010]: If a package is included in Java, the sub-packages thereof will not be included automatically.

The properties of the concept *Library* are:

- **includedBy:** This property defines the opposite of the property *includedLib* in *DefContainer* and thus it contains all libraries and components which include this library. Note that cyclic includes should not be allowed.
- **sublibrary:** The sub-libraries of said library. A sub-library may again comprise sub-libraries.

3.4.3. Concept *Component*

A component forms the re-usable unit within our exemplary modelling language. In detail, a component may be a network of other components or just a block which cannot be decomposed any further. Thus, each component comprises a syntactical interface which is called *signature*.

The properties of the concept *Component* are:

- **signature:** The signature of the component. Each component must have exactly one signature.

3.4.4. Concept *Signature*

The *Signature* represents the syntactical interface of a *Component*. It consists of a set of input ports and a set of output ports.

The properties of the concept *Signature* are:

- **inPort:** The set of all input ports.
- **outPort:** The set of all output ports.

3.4.5. Concept *Port*

The concept *Port* represents a single input or output port. Note that there are no dedicated sub-concepts such as *InputPort* and *OutputPort* as this is modelled as a role of the port. Thus, the decision on whether a port is an input or an output port depends on whether it is referenced by the property *inPort* or *outPort* within the signature. As mentioned above, our exemplary language is untyped for simplicity. Hence, a port is simply described by its name and does, for example, not include a data type.

The properties of the concept *Port* are:

- **name:** The name of a port. This name must be unique within a signature over input *and* output ports.

3.4.6. Concept *Block*

The concept *Block* defines a component which can be seen as a primitive within the language. Arithmetic operators such as *add*, *sub*, *mult*, or *div* are normally modelled as blocks. The *pre*-operator is also modelled as a block. It delays its input for one cycle.

Blocks are also used for modelling sensors and actuators in the form of data sources and data sinks. Such blocks either have input ports (i. e. actuators/data sinks are modelled) or output ports (i. e. sensors/data sources are modelled). A prominent source block is dT. It returns the physical delta time (e. g. in milliseconds) between two cycles.

Additional properties to the inherited ones of the concept *Component* are not necessary.

3.4.7. Concept *Network*

The concept *Network* is the core of our exemplary modelling language. As the name suggests, it specifies the dataflow network itself. Ports of the signature as well as those of its sub-components are connected via channels within a network. In order to prevent fix-point arithmetic, cyclic connections are not allowed unless a particular *pre*-block is located in between.

The properties of the concept *Network* are:

- **subcomponent:** The sub-components of the present network. In order to implement re-use, sub-components must be already specified (see property *componentsDef* in concept *DefContainer*). Then, the property *subcomponent* must contain a clone of such a component definition to be able to reference their ports of the signature correctly. Please refer to Chapter 7, *Abstract Syntaxes in M2L*, p. 119 for further details.

- **channel:** The channels connecting the ports within this dataflow network.

3.4.8. Concept *Channel*

The concept *Channel* represents a channel within a dataflow network. Each channel connects exactly one source port to at least one destination port. Herein, a source port could be an input port of the signature or an output port of a sub-component. Similarly, a destination port could be an output port of the signature or an input port of a sub-component.

The properties of the concept *Channel* are:

- **name:** Name of the channel.
- **fromPort:** Source port of the channel.
- **toPort:** Destination ports of the channel.

3.5. Two exemplary dataflow diagrams

As the structural design of the language has been illustrated above by introducing abstract syntax, two concrete dataflow networks shall now be introduced. Most of the ongoing illustrations within the present work are based on these two examples.

In this section, the two desired textual concrete syntaxes for dataflow networks, named *Structural* and *Functional*, shall be introduced as well:

- The textual concrete syntax **Structural** focuses on the basic idea of dataflow networks: A network consists of sub-components which are connected via channels.
- In contrast to the first one, the textual concrete syntax **Functional** concentrates on the fact that each dataflow network can be seen as a mathematical function mapping input ports to output ports.

One or the other syntax will be more adequate depending on the circumstances. Although the decision on which one will be preferred is subjective, the following two examples will show that in case of the first example (see Section 3.5.1, *Integrator*, p. 53) functional syntax might be preferred in many situations, whereas in case of the second example (see Section 3.5.2, *Demonstration Vehicle*, p. 55) structural syntax might be more useful.

Although the present work focuses on textual concrete syntaxes, the examples will first of all be presented in a diagrammatic way – as defined for *COLA* – in order to allow for a faster understanding of what each of the models does.

3.5.1. Integrator

Our first example is a simple integrator over time. It consists of one input port x and one output port y. As our language is untyped, the type of each port is always denoted by *Any*. First of all, it multiplies the input value by the delta time of the last cycle. Afterwards, the result value is summed up with the result value of the last cycle (done via the *pre*-operator). Note that it is not possible to define an initial value for the *pre*-operator in our simplified modelling language. It is assumed to be *zero* for the first cycle.

Figure 3.3 shows the implementation of the integrator over time in a diagrammatic way: It consists of four sub-components: *dT*, *pre*, *mult* (denoted by $*$), and *add* (denoted by $+$). To

sum it up, five channels c_1 to c_5 connect all ports. Here, c_1 to c_4 are 1-to-1 channels, and c_5 is a 1-to-many channel.

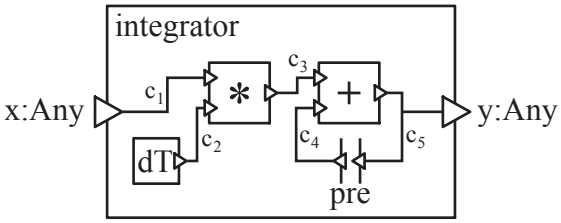


Figure 3.3.: Exemplary model: integrator network (diagrammatic syntax)

In the following, this dataflow network shall be encoded by the two textual concrete syntaxes, namely *Structural* and *Functional*.

Structural syntax. As mentioned above, the textual concrete syntax *Structural* focuses on the fact that a network consists of sub-components which are connected via channels. Listing 3.1 shows the encoding of our integrator example.

Listing 3.1: Exemplary model: integrator network (structural syntax)

```
1  network integrator (x:Any -> y:Any) {
2    include basic;
3
4    subcomponent dT[dT0];
5    subcomponent pre[pre0];
6    subcomponent mult[mult0];
7    subcomponent add[add0];
8
9    channel c1: x => mult0.x;
10   channel c2: dT0.out => mult0.y;
11   channel c3: mult0.result => add0.x;
12   channel c4: pre0.out => add0.y;
13   channel c5: add0.result => pre0.in, y;
14 }
```

In the beginning, a reserved word **network** indicates that a dataflow network is encoded, followed by the name of said network. Then, signature within parentheses is required: The input ports – generally separated by commas – will be encoded first. The output ports – again, generally separated by commas – follow after an arrow encoded by ->.

After the signature the actual definition of the network begins, which is embedded within curly brackets. It starts with an include statement indicating that the library *basic* is required. This library is not shown herein and defines the blocks necessary for *dT*, *pre*, *mult*, and *add*, as the sub-components are only references to their definitions.

After the include statement the sub-components are referenced or more precisely: instantiated. The first name (e. g. *dT*) is the name of the instantiated component's name. The second name within the square brackets defines an instance name. This is generally necessary in order to distinguish two instances of the same block. This is for example the case if two *add*-blocks are present within one network.

In the end, the channels are encoded: After the reserved word **channel**, the name is required. Then, the source port is encoded followed by an arrow encoded as =>. Afterwards, the destination ports – generally separated by commas – are encoded. If the referenced port belongs to a sub-component, the instance name is necessary in front and separated by a dot.

Functional syntax. As mentioned above, in contrast to the syntax *Structural*, the textual concrete syntax *Functional* concentrates on the fact that each dataflow network can be seen as a mathematical function mapping input ports to output ports. Listing 3.2 shows the encoding of our integrator example:

Listing 3.2: Exemplary model: integrator network (functional syntax)

```
network integrator (x:Any -> y:Any) {
  include basic;

  y := add(mult(x, dT()), pre(y)) ;
}
```

Here it can be seen that, although the header of both representations is the same, the functional representation is much shorter than the structural one: The output port y is assigned to the corresponding functional statement. In this special case no channel names will be required. Even the instance names are omitted herein.

This example shows that not every textual syntax has to represent all information as in our context a textual syntax is only a view to overall abstract syntax (which is e. g. stored in a database or XML file). Shortness is therefore not only based on the syntactical definition but also on the fact that part of the information (especially instance names and channel names) is not defined therein.

All in all, defining the functional syntax is much more tricky than defining the structural. The reason is that the given structure of abstract syntax is much closer to structural syntax which allows for a straight-forward definition. Not all details of functional syntax will be presented herein as it contains a lot of exceptional cases. At this point it is important to understand that different syntaxes may be useful for different situations. On the other hand, the requirement shall be formulated that it should also be possible to define a more complex textual syntax (as the syntax *Functional*).

3.5.2. Demonstration Vehicle

Our second example models a top-level view of a demonstration vehicle which was developed as a case-study for *COLA*. This example was selected to show the ubiquitous applicability of dataflow networks: They can be used for both extremely low-level and extremely high-level modelling. It will also be shown that for such a high-level example – in contrast to our fist example – the syntax *Structural* will be better suited than the syntax *Functional*.

Figure 3.4 shows the implementation of a top-level view of the demonstration vehicle in a diagrammatic way.

In total it consists of six sub-components: three data sources, namely *User Interface*, *Rotation Sensor*, and *Radar Sensor*; two data sinks, namely *Display*, and *Engine*; and one complex behavioural component, namely *Adaptive Cruise Control* which provides the actual functionality of the system.

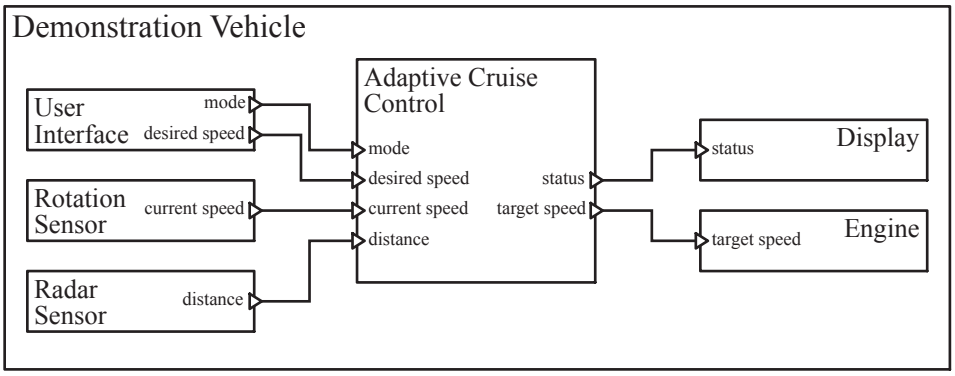


Figure 3.4.: Exemplary model: demonstration vehicle (diagrammatic syntax)

The six channels *mode*, *desired speed*, *current speed*, *distance*, *status*, and *target speed* connect all sub-components. In contrast to our first example, the present one represents a closed system and does therefore not have an external signature. Note that the channel names are not visible in the diagram. Only the port names which are chosen so as to be the same on both sides of the channel can be seen. The channel itself is also named the same way.

In our context the detailed meaning of the sub-components and ports is not that important. Thus, a detailed explanation will be skipped. As it was done in the first example, this dataflow network shall be encoded by the two textual concrete syntaxes, namely *Structural* and *Functional* in the following.

Structural syntax. Listing 3.3 shows the encoding of our demonstration vehicle example by using the textual concrete syntax *Structural*. As most details have already been introduced during our fist example (see Section 3.5.1, *Integrator*, p. 53), only the deltas will be dealt with.

Listing 3.3: Exemplary model: demonstration vehicle (structural syntax)

```
network "Demonstration Vehicle" ( -> ) {

  network "Adaptive Cruise Control" (mode:Any, "desired speed":Any,
    "current speed":Any, distance:Any -> status:Any,
    "target speed":Any) {
    ...
  }
  block Display (status:Any -> ) {}
  block Engine ("target speed":Any -> ) {}
  block "Radar Sensor" ( -> distance:Any) {}
  block "Rotation Sensor" ( -> "current speed":Any) {}
  block "User Interface" ( -> mode:Any, "desired speed":Any) {}

  subcomponent "Adaptive Cruise Control"
    ["Adaptive Cruise Control0"];
  subcomponent Display[Display0];
  subcomponent Engine[Engine0];
  subcomponent "Radar Sensor"["Radar Sensor0"];
  subcomponent "Rotation Sensor"["Rotation Sensor0"];
```

```
20   subcomponent "User Interface" ["User Interface0"];
21
22   channel "current speed": "Rotation Sensor0"."current speed"
23     => "Adaptive Cruise Control0"."current speed";
24   channel "desired speed": "User Interface0"."desired speed"
25     => "Adaptive Cruise Control0"."desired speed";
26   channel distance: "Radar Sensor0".distance
27     => "Adaptive Cruise Control0".distance;
28   channel mode: "User Interface0".mode
29     => "Adaptive Cruise Control0".mode;
30   channel status: "Adaptive Cruise Control0".status
31     => Display0.status;
32   channel "target speed": "Adaptive Cruise Control0"."target speed"
33     => Engine0."target speed";
34 }
```

First of all, the empty signature (−>) attracts attention which results from the fact that this system is a closed one.

Secondly, instead of an *include*-statement, a set of inline-defined networks and blocks will be found: As mentioned above, a network or block must be defined before it can be instantiated. Components have been used in the first example, which were defined in the library *basic*, as the required components were standard components. Herein, the required components are specifically tailored to the vehicle to be modelled. The components are therefore defined inline.

Note that an implementation of the network *Adaptive Cruise Control* is obvious. To simplify the example, this implementation will be omitted herein. The second example also shows that identifiers may contain arbitrary characters. In such situations the identifiers are put into quotes.

Functional syntax. Although functional syntax is not that intuitive in case of the present example, it is, however, possible to encode it this way as well. Listing 3.4 shows the encoding of our demonstration vehicle example by using the textual concrete syntax *Functional*:

Listing 3.4: Exemplary model: demonstration vehicle (functional syntax)

```
1  network "Demonstration Vehicle" ( -> ) {
2
3    network "Adaptive Cruise Control" (mode:Any, "desired speed":Any,
4      "current speed":Any, distance:Any -> status:Any,
5      "target speed":Any) {
6      ...
7    }
8    block Display (status:Any -> ) {}
9    block Engine ("target speed":Any -> ) {}
10   block "Radar Sensor" ( -> distance:Any) {}
11   block "Rotation Sensor" ( -> "current speed":Any) {}
12   block "User Interface" ( -> mode:Any, "desired speed":Any) {}
13
14   (mode, "desired speed") := "User Interface"() ;
15   (status, "target speed") := "Adaptive Cruise Control"(
16     mode, "desired speed", "Rotation Sensor"(), "Radar Sensor"()) ;
```

```
17    Display(status) ;
18    Engine("target speed") ;
19  }
```

The first part including the definitions of networks and blocks is quite similar. Then, there are four assignments whereupon the last two will be degenerated as the sub-components *Display* and *Engine* are assigned to nothing. Note that the order of the four assignments is arbitrary.

For coding reasons, some of the channels are seen explicitly (in particular *mode*, *desired speed*, *status*, and *target speed*) in this example. The channels *current speed* and *distance* are still not necessary.

3.6. Issues to be expressed by a formalised metamodel

Having introduced all three abstract syntaxes, concrete syntaxes, and two exemplary models, the issues in the context of our exemplary modelling language shall be listed so as to be solved by a formalised metamodel, such as *M2L*, as will be described in the following chapters. The issues will be classified into three types: those affecting abstract syntax, those affecting concrete syntax, and finally those affecting the inspection of the model.

3.6.1. Issues affecting abstract syntax

Up to now, abstract syntax has only been defined in a semi-formal way. Besides that it only defines the basic structure. Many detailed consistency conditions are not expressed at all. All in all, the following major issues regarding our exemplary language affecting abstract syntax should be solved:

- **No cyclic re-use of components.** When a network is defined, a set of other sub-components is used. It should be guaranteed that there is no cyclic definition – even when sub-components are decomposed recursively.
- **No cycle without a *pre*-component.** A directed cycle is not allowed within a network unless at least one *pre*-block is located inbetween.
- **Each output port must be connected.** All output ports within a network must be connected via a channel in order to prevent undefined values.
- **Local uniqueness of identifiers.** The names of identifiers for ports and sub-components must be locally unique such that there is a canonical name which is unique.
- **Channels always define valid connections.** According to the abstract syntax defined, a channel may connect arbitrary ports to each other. At first it must be ensured that only ports of that network to which also the channel belongs, are connected to each other. Secondly, only correct source and destination ports should be connected. It should, for example, be prevented that two output ports are connected to each other.
- **Valid instantiation of components.** As described above, instantiation is realised by cloning the original component. Here, it must be ensured that the original component is exactly the same as the cloned component.

- **Correct include mechanism.** As defined above, components can only be referenced if they are part of a library that is included. This mechanism should be defined in a formal way.

3.6.2. Issues affecting concrete syntax

Besides those regarding abstract syntax, there are also issues concerning concrete syntax. The following major issues affecting concrete syntax should be solved by a formalised metamodelling approach:

- **No impact on abstract syntax by concrete syntax.** When a concrete syntax is defined, reserved words normally restrict possible values for identifiers. This should not be true for this language. If, on the other hand, a new concrete language is defined for an abstract syntax, the abstract syntax itself is altered.
- **Local references to ports and components.** The canonical name is much too long when a port or a component are referenced. Due to that, a formal definition is required of how to achieve unique shortcuts.
- **Identifiers including arbitrary characters.** In order to allow arbitrary characters within identifiers, a well-defined way of quoting identifiers becomes necessary.

3.6.3. Issues affecting model inspection

Finally, there are a lot of necessary model inspections which are required. Focus will be on two major issues regarding our exemplary language:

- **Correct execution order of components.** When a dataflow network is evaluated, a partial execution order must be followed. It should be possible to calculate said execution order.
- **All components involved in a network definition.** It is necessary to know all components that will recursively be used for defining the network when generating a code.

In the following chapters it shall be illustrated how all these requirements can be addressed by the metamodelling language *M2L*.

Chapter 4

Pomsets in the context of metamodelling

Pomsets themselves have been known for many years and have originally been invented for process modelling in particular. Pratt introduced them in [Pratt, 1985]. In [Grumbach and Milo, 1995] an algebra of pomsets is defined in order to create a generalisation of the traditional algebras for (nested) sets, bags and lists.

Nevertheless, they have not been used in the domain of metamodelling up to now, although sets, bags as well as lists have to be combined within one (meta-)model. To support this combination in a sound way, it will be shown that pomsets are the right concept in the metamodelling domain, too. In addition, pomsets will allow a much more powerful model query: E. g. in the context of weak-causal dataflow networks – as introduced in Chapter 3, *Running example: Modelling dataflow algorithms*, p. 45 – the calculation of the execution order would result in a partially ordered set. This can easily be handled in the case of pomsets, of course. (See Section 4.5, *Running Example*, p. 80 for details.)

This chapter will provide a short overview of pomsets and will introduce the operators defined in [Grumbach and Milo, 1995] complemented by a set of additional operators which are particularly important for the present work and for the metamodelling domain in general.

Contents

4.1. Relationship between different types of sets

As mentioned in Section 1.3, *Approach*, p. 8, MOF introduces two modifiers required for sets: {*ordered*} and {*bag*}. According to that, it is possible to describe the four types, namely *set*, *totally ordered set (toset)*, *bag*, and *list*. In order to formulate an appropriate mathematical foundation, a common set type generalising all four types is necessary. Unfortunately, none of the four is mostly general. In particular, *Bag* generalises *Set* (to be written $Set \subset Bag$)

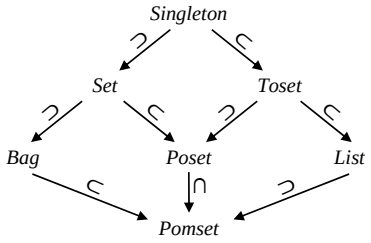


Figure 4.1.: Inclusion relationships between the different types of sets

and *List* generalises *Toset* (to be written $Toset \subset List$), but besides these two there is no further generalisation relationship.

In some approaches, lists are seen as the most general set type: In case of a (normal) set, the order is simply ignored. When having a closer look at it, it will be realised that an additional boolean attribute has to be taken into account implicitly, indicating whether the list should be seen as a set or not. This leads to many problems: Two different equation operators – one ignoring the order, whereas the other does not – will therefore become necessary.

To get an exact picture in a mathematical sound way, the *partially ordered multi-set (pomset)* and its specialisation *partially ordered set (poset)* have to be introduced. As can be seen in Figure 4.1, *pomset* is a common generalisation for all set types mentioned.

Singleton will also be illustrated to show that single valued *sets* and single valued *lists* are equal in the sense of pomsets. This corresponds to the fact that for single-valued properties in MOF the modifiers $\{ordered\}$ and $\{bag\}$ are not allowed.

On the other hand, there are many programming environments where single-valued lists and sets are not equal. This is for example true in the Java Collection API. Besides that, there is a third way of describing a single value by omitting the set type around the value. Again, this complicates the expressions as we have to take explicit care of both concepts. This leads to many case differentiations especially in generic modelling environments.

Thus, in our approach there is no distinction between a single-valued set and a single-valued list. In addition to that, single values themselves will be omitted entirely. They are always represented as a single-valued pomset. For this reason, a single value can also be given whenever a pomset is expected, so it can be written $a = \{a\}$.

4.2. Definition of pomsets

As could be seen in the previous sections, pomsets can provide a unified view of sets, lists as well as single values. Although pomsets are already known, they shall be introduced herein again, concentrating on a different focus, however, namely the metamodelling domain. Informally spoken, a pomset is a multi-set in which the elements are additionally partially ordered. If the partial order is *total* a so-called *totally ordered multi-set (tomset)* will be obtained which is also known as *list*. A formal definition for a pomset is given as follows:

Definition 1 (Pomset)**.** *A* partially ordered multiset (pomset) *over a set E is a quadruple $\langle E, V, \lambda, \prec \rangle$ where E is a basic set of elements, V is a set of vertices, $\lambda : V \to E$ is a total function labelling each vertex with an element from the basic set, and $\prec \subseteq V \times V$ is a strict (irreflexive) partial order over V.*

Two pomsets $A = \langle E_A, V_A, \lambda_A, \prec_A \rangle$ *and* $B = \langle E_B, V_B, \lambda_B, \prec_B \rangle$ *are equal (denoted by* $A = B$*) if, and only if, a bijection* $\varphi : V_A \rightarrow_{bij} V_B$ *exists, such that* $\forall v \in V_A : \lambda_A(v) = \lambda_B(\varphi(v))$ *and* $\forall u, v \in V_A : u \prec_A v \Leftrightarrow \varphi(u) \prec_B \varphi(v)$*. The set of all partially ordered multisets over a basic set* E *is denoted by* $\mathcal{P}_{pomset}(E)$*.*

Due to the given equivalence relation, the concrete set of vertices is irrelevant. Thus, disjoint sets of vertices can be assumed for any two pomsets if needed.

Note that an equation over pomsets is deliberately defined instead of an isomorphism although pomsets can be seen as directed graphs, as the concrete set of vertices has absolutely no influence on the pomset. This equation operator also corresponds to those over sets, bags, lists or even singletons.

Now, different types of sets can be defined based on this pomset definition. It is easy to verify that the relationships given in Figure 4.1 result from these definitions.

- $\mathcal{P}_{bag}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid \prec = \emptyset\}$

  A pomset is a bag if, and only if, the order $\prec$ over the pomset vertices V is empty. Thus, the elements are completely unordered but may contain duplicates.

- $\mathcal{P}_{list}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid \prec \textit{ is total}\}$

  A pomset is a list if, and only if, the order $\prec$ over the pomset vertices V is total. Thus, the elements are completely ordered but may contain duplicates.

- $\mathcal{P}_{poset}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid \lambda \textit{ is injective}\}$

  A pomset is a partially ordered set if, and only if, the labelling function λ is injective. Thus, there do not exist two vertices which are mapped to the same element from E. Hence, the pomset does not contain duplicates.

- $\mathcal{P}_{toset}(E) = \mathcal{P}_{list}(E) \cap \mathcal{P}_{poset}(E)$

  A pomset is a totally ordered set if, and only if, the pomset is both a list and a partially ordered set. Hence, the pomset is totally ordered but does not contain duplicates.

- $\mathcal{P}_{set}(E) = \mathcal{P}_{bag}(E) \cap \mathcal{P}_{poset}(E)$

  A pomset is a set if, and only if, the pomset is both a bag and a partially ordered set. Hence, the pomset is unordered but does not contain duplicates.

- $\mathcal{P}_{singleton}(E) = \{\langle E, V, \lambda, \prec \rangle \in \mathcal{P}_{pomset}(E) \mid |V| \leq 1\} = \mathcal{P}_{set}(E) \cap \mathcal{P}_{toset}(E)$

  A pomset is a singleton if, and only if, the set of vertices V is empty or contains exactly one element. Another definition is: A pomset is a singleton if, and only if, the pomset is both a set and a totally ordered set.

4.3. Notations for pomsets

As previously defined, pomsets are a generalisation of sets and lists. Nevertheless, the internal structure is much more complex. Thus, it is important to have suitable notations for pomsets. Adequate notations for pomsets, which will be used in the following sections and chapters, will now be introduced. At first, the mathematical, straight-forward notation will be illustrated as a quadruple. Then, the graphical notation will be defined based on transitively reduced graphs. Finally, a linear textual notation, which is more convenient than the quadruple, will be provided.

4.3.1. Pomsets as quadruples

Using Definition 1, a pomset can be specified explicitly by a quadruple, as in the following example (4.1): At first, the set of elements $\{a, b, c\}$ will be defined. Then, an arbitrary set of vertices $\{v_1, v_2, v_3, v_4, v_5\}$ will be given. In order to identify the vertices easily, v_i will always be used from now on. Next, the labelling function $\{(v_1 \mapsto a), (v_2 \mapsto a), (v_3 \mapsto b), (v_4 \mapsto b), (v_5 \mapsto c)\}$ will be defined in the given notation. And in the end, the partial order over the vertices $\{(v_1, v_2), (v_2, v_3), (v_2, v_5), (v_1, v_3), (v_1, v_5)\}$ will be given by a set of tuples.

$$\begin{aligned} A = \langle &\{a, b, c\}, \{v_1, v_2, v_3, v_4, v_5\}, \\ &\{(v_1 \mapsto a), (v_2 \mapsto a), (v_3 \mapsto b), (v_4 \mapsto b), (v_5 \mapsto c)\}, \\ &\{(v_1, v_2), (v_2, v_3), (v_2, v_5), (v_1, v_3), (v_1, v_5)\}\rangle \end{aligned} \tag{4.1}$$

As the $\prec$-relation has to be transitive, tuples resulting from transitivity can be omitted. Hence, orders can be defined by *transitively reduced relations*. They are defined as the smallest subset of the relation wherein its transitive closure results in the original relation. In our example (4.1) it is now possible to skip the last two tuples, namely (v_1, v_3) and (v_1, v_5). Our example (4.1) will now be simplified to (4.2):

$$\begin{aligned} A = \langle &\{a, b, c\}, \{v_1, v_2, v_3, v_4, v_5\}, \\ &\{(v_1 \mapsto a), (v_2 \mapsto a), (v_3 \mapsto b), (v_4 \mapsto b), (v_5 \mapsto c)\}, \\ &\{(v_1, v_2), (v_2, v_3), (v_2, v_5)\}\rangle \end{aligned} \tag{4.2}$$

4.3.2. Pomsets as transitively reduced graphs.

A more intuitive notation for pomsets based on a graph representation and which is closer related to the basic idea of pomsets shall now be introduced: By way of this notation, a pomset is represented by a directed graph. Hereby the vertices are labelled with the pomset elements. If the pomset contains duplicates, two vertices are labelled by the same element. The partial order of the pomset is represented by the directed edges of the graph: If a directed edge exists from one element to another $(e_1 \rightarrow e_2)$, it is indicated that $e_1 \prec e_2$. In accordance to transitively reduced relations the transitively reduced graph will be shown for clarity. In order to indicate that the graph represents a pomset, it is placed into curly brackets. Our example (4.2) can now be specified by (4.3):

$$A = \left\{ \begin{array}{ccc} \mathrm{b} & & \\ \mathrm{a} \rightarrow \mathrm{a} & \nearrow & \mathrm{b} \\ & \searrow & \mathrm{c} \end{array} \right\} \tag{4.3}$$

4.3.3. Pomsets as linear text.

Although the pomset notation by transitively reduced graphs is the most intuitive and convenient one, in some situations – in programming environments or textual concrete syntaxes in general – a representation as linear text will be necessary. The notation as a quadruple is also linear, of course. The focus concerning notation herein is less mathematical but more intuitive. It should also be a little shorter than the quadruple.

The set of elements will therefore be enriched by adding the mapped identifiers of the vertices within square brackets behind each element. v_i with ascending indices will again be used for these identifiers. If a pomset element comes up within this pomset more than once, only the first and the last indexed identifier – separated by ".." – will be denoted. In the present example, the two vertices v_1 and v_2 are mapped to the element a, resulting in $a[v_1..v_2]$. After all elements have been enumerated, separated by a "|", the transitively reduced order is denoted by simply stringing together the relationships such as $v_1 \prec v_2$. If the pomset is unordered – i. e. no relationship does exist – the separating "|" has to be skipped. Our example (4.3) can now be described by (4.4):

$$A = \{\, a[v_1..v_2], b[v_3..v_4], c[v_5] \mid v_1 \prec v_2 \quad v_2 \prec v_3 \quad v_2 \prec v_5 \} \tag{4.4}$$

When depending on ASCII characters, the notation is simplified by skipping the indexed notation and replacing "$\prec$" by "$<$", as illustrated in (4.5):

$$A = \{\, a[v1..v2], b[v3..v4], c[v5] \mid v1 < v2 \quad v2 < v3 \quad v2 < v5 \} \tag{4.5}$$

4.3.4. Special notations for bags and lists

If a pomset is totally ordered or unordered it can be described in a much easier way. For bags it is sufficient to list the elements separated by commas. If one element comes up in the pomset more than once, it will accordingly come up more than once in the listing, too:

$$\{a, a, b, b, c\} =_{def} \{a[v_1..v_2], b[v_3..v_4], c[v_5]\} \tag{4.6}$$

Lists can be encoded quite similar to bags: The elements are listed again – separated by commas. In order to distinguish lists from bags, the curly brackets are replaced by angle brackets. In contrast to bags, the order of the listed elements is relevant:

$$\langle a, b, a, c, b \rangle =_{def} \{\, a[v_1..v_2], b[v_3..v_4], c[v_5] \mid v_1 \prec v_3 \quad v_3 \prec v_2 \quad v_2 \prec v_5 \quad v_5 \prec v_4 \} \tag{4.7}$$

4.3.5. Bags specified by a multiplicity function

Finally, a common notation shall be extended by defining sets in order to define bags. When specifying a set S, the following notation $S = \{e \in E \mid p(e)\}$ will often be used, whereas $p(e)$ is a predicate over E. The meaning of this notation is that the resulting set S contains all those elements from E for which the predicate $p(e)$ is true. Based on this definition, this notation can be extended to bags:

$$B = \{e \in E \mid m(e)\} \tag{4.8}$$

Instead of defining a predicate $p(e)$, a multiplicity function $m(e)$ is defined, specifying the multiplicity of each element out of E for the bag. Thus, the function $m : E \to \mathbb{N}$ is exactly the multiplicity function of the specified bag B. An example might be:

$$\{e \in \{1, 2, 3\} \mid e - 1\} = \{2, 3, 3\} \tag{4.9}$$

4.4. Operators on pomsets

Having introduced the pomset itself as a generalisation of sets and lists, the pomset operators shall now be defined in the following section. According to the complexity of the structure, much more operators will be required than for sets, for example. Although some of these operators have already been defined in [Grumbach and Milo, 1995], those operators which turn out to be useful in the metamodelling domain shall be quoted and combined with some new ones. Table 4.1 gives you an overview of the operators.

operator	notation
CC-Extraction	κA
Expansion	$A \chi f$
Order Expansion	$A \overrightarrow{\chi} B$
Additive Union	$A \uplus B$
Concatenation	$A \oplus B$
Projection	$A \downarrow B$
Complement Projection	$A \downarrow^* B$
Consists Of	$A \Subset B$
Union	$A \cup B$
Intersection	$A \cap B$
Difference	$A \setminus B$
Subset	$A \subseteq B$
Cardinality	$\lvert A \rvert$
Depth	$\lVert A \rVert$
Multiplicity	$a \in_m A$
Path	πA
Totalise	τA
Sub-Pomset	$[i; j]A$
First	$1A$
Order Inverse	$\gtrless A$
Order Destroy	μA
Duplicate Destroy	ϵA
Duplicate Destroy Over CCs	$\epsilon_{\subset} A$
Is Empty	empty? A
Is Singleton	singleton? A
Is Set	set? A
Is Bag	bag? A
Is List	list? A
Is Toset	toset? A
Is Poset	poset? A

Table 4.1.: Overview of pomset operators

For each operator, a short motivation shall be provided of why and in which context this operator is necessary (or at least helpful). After the formal definition, an example will be provided to illustrate the operator. For some very intuitive operators a formal definition will be skipped due to complexity.

For all following definitions we define $A = \langle E_A, V_A, \lambda_A, \prec_A \rangle$ and $B = \langle E_B, V_B, \lambda_B, \prec_B \rangle$ as being two pomsets. It is assumed without loss of generality that the two sets of vertices V_A and V_B are disjoint.

4.4.1. CC-Extraction, κ

As pomsets allow for a partial order, there are pairs of elements related to each other, and there are pairs of elements unrelated to each other. This allows to represent a set (or even bag) of lists in one pomset, for example. An example may be:

$$A = \left\{ \begin{array}{c} \mathrm{b} \to \mathrm{b} \to \mathrm{d} \\ \mathrm{a} \to \mathrm{c} \to \mathrm{c} \to \mathrm{a} \end{array} \right\} \tag{4.10}$$

The pomset A can be seen as a bag containing the two lists, namely $\langle b, b, d \rangle$ and $\langle a, c, c, a \rangle$. In some situations an explicit representation of this set of lists will now be desired, such that:

$$B = \left\{ \begin{array}{c} \left\{ \mathrm{b} \to \mathrm{b} \to \mathrm{d} \right\} \\ \left\{ \mathrm{a} \to \mathrm{c} \to \mathrm{c} \to \mathrm{a} \right\} \end{array} \right\} \tag{4.11}$$

The CC-Extraction operator κ will therefore be used, such that:

$$B = \kappa A$$

In order to provide a general definition for arbitrary pomsets, *connected components* (abbreviated by *CC*) will have to be introduced, as in [Grumbach and Milo, 1995]. The easiest way to understand connected components is to look at the transitively reduced graph which represents a pomset:

Definition 2 (Connected Component, CC)**.** *The connected components of a pomset are exactly those connected sub-graphs of the transitively reduced graph which represent the pomset.*

Based on the definitions of connected components, it will now be possible to define the CC-Extraction operator:

Definition 3 (CC-Extraction, κ)**.** *The CC-Extraction operator* $\kappa : \mathcal{P}_{pomset}(E) \to \mathcal{P}_{bag}(\mathcal{P}_{pomset}(E))$ *returns a bag of pomsets, each containing one connected component of the original pomset. It is written as* κA*.*

Example

A more generalised example than given in (4.10) and (4.11) is:

$$\kappa \left\{ \begin{array}{c} \mathrm{a} \nearrow \mathrm{b} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \quad \begin{array}{c} \mathrm{a} \nearrow \mathrm{b} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \quad \mathrm{a} \to \mathrm{c} \right\} = \left\{ \left\{ \begin{array}{c} \mathrm{a} \nearrow \mathrm{b} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \quad \left\{ \begin{array}{c} \mathrm{a} \nearrow \mathrm{b} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \quad \left\{ \mathrm{a} \to \mathrm{c} \right\} \right\}$$

Note that in the present example the result is a bag and not a set as one pomset comes up twice.

This operator will also help to extend the operators on the sets *union*, *intersection*, *subset*, and *difference* to pomsets. For further details please refer to Section 4.4.9, *Union,* ∪, p. 73 and the following.

4.4.2. Expansion, χ

Let $A = \langle a, c, c, a \rangle$ be a list of elements. If each of the elements shall be replaced by others, e. g. a should be replaced by a, b and c should be replaced by c, d, the result will be $B = \langle a, b, c, d, c, d, a, b \rangle$.

Formally, a replacement function $f : E \to \mathcal{P}_{pomset}(E)$ will be required. For our example this replacement function will be $f = \{(a \mapsto \langle a, b \rangle), (b \mapsto \langle c, d \rangle)\}$.

If we would deal with sets, the formal definition of such a replacement is easy to formulate:

$$B = \bigcup_{e \in A} f(e)$$

In the context of lists (or even pomsets in general) replacing elements is more complicated. Nevertheless, it could be defined by a replacement function f:

Definition 4 (Expansion, χ)**.** *Let f be a function $f : E_A \to \mathcal{P}_{pomset}(E_f)$. $f(e) = \langle E_{f(e)}, V_{f(e)}, \lambda_{f(e)}, \prec_{f(e)} \rangle$ denotes the pomset quadruple that results from applying f to $e \in E_A$. Without loss of generality, disjoint sets of vertices are assumed for all these pomsets. $A \chi f$ is a pomset where all elements $e \in A$ are replaced by the pomset $f(e)$, preserving the order:*

$$\begin{aligned} & A \chi f =_{def} \langle E', V', \lambda', \prec' \rangle \\ & \textit{where} \\ & E' = \bigcup_{e \in E_A} E_{f(e)} \subseteq E_f \\ & V' = \{ (v, w) \mid v \in V_A \wedge w \in V_{f(\lambda_A(v))} \} \\ & \lambda' : V' \to E', (v, w) \mapsto \lambda_{f(\lambda_A(v))}(w) \\ & \prec' = \{ ((v, w), (v', w')) \mid (v \prec_A v') \vee (v = v' \wedge w \prec_{f(\lambda_A(v))} w') \} \end{aligned} \tag{4.12}$$

Thus, the expansion operator can be seen as a graph transformation wherein f defines the replacement rules.

Example

$$\left\{ \begin{array}{c} \mathrm{a} \nearrow \mathrm{b} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \quad \mathrm{a} \to \mathrm{c} \right\} \chi \left\{ \begin{array}{c} (a \mapsto \{a, b\}), \\ (b \mapsto \langle b, c \rangle), \\ (c \mapsto \varnothing) \end{array} \right\} = \left\{ \begin{array}{ccc} \mathrm{a} \longrightarrow \mathrm{b} \longrightarrow \mathrm{c} & & \\ \quad \searrow \nearrow & \mathrm{a} & \mathrm{b} \\ \mathrm{b} \longrightarrow \mathrm{b} \longrightarrow \mathrm{c} & & \end{array} \right\}$$

In particular, the expansion operator will be used for defining the navigation operator of the Edge Algebra. For further details please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95.

A special situation occurs if the replacement function is an identity function over pomsets f_{id}. In this case the expansion operator does not – as might be assumed – result in an

equal pomset. Instead, this only makes sense if the elements of the pomsets are pomsets themselves (and thus also the identity function is defined over pomsets). Then, the expansion operator removes internal pomsets. The expansion operator in particular will, together with the identity function, invert the CC-Extraction operator:

$$\left\{ \left\{ \begin{array}{c} a \nearrow b \\ a \searrow b \end{array} \right\} \quad \left\{ \begin{array}{c} a \nearrow b \\ a \searrow b \end{array} \right\} \quad \left\{ a \to c \right\} \right\} \chi\, f_{id} = \left\{ \begin{array}{c} a \nearrow b \\ a \searrow b \end{array} \quad \begin{array}{c} a \nearrow b \\ a \searrow b \end{array} \quad a \to c \right\}$$

4.4.3. Order Expansion

The previously defined expansion operator allows for a replacement of a pomset's elements. In an analogous way, the *Order Expansion* operator replaces the edges of the transitively reduced graph representation of a pomset.

This operator will be used in particular when infixes will be defined within a textual syntax. (For further details please refer to Chapter 8, *Textual Concrete Syntaxes in M2L*, p. 153.) For a given sequence of symbols $\langle a, b, c \rangle$, for example, it will then be possible to insert an infix element x very easily:

$$\langle a, b, c \rangle \overrightarrow{\chi} \{x\} = \langle a, x, b, x, c \rangle$$

Definition 5 (Order Expansion, $\overrightarrow{\chi}$). *$A \overrightarrow{\chi} B$ is a pomset containing all elements out of A preserving the order, and additionally adds the pomset B for each transitively reduced relationship between two elements $e_1 \prec e_2$, such that $\forall b \in B : e_1 \prec b \prec e_2$:*

$$\begin{aligned}
&A \overrightarrow{\chi} B =_{def} \langle E', V', \lambda', \prec' \rangle \\
&\textit{where} \\
&E' = E_A \cup E_B \\
&V' = V_A \cup (tr(\prec_A) \times V_B) \\
&\lambda' : V' \to E', v \mapsto \begin{cases} \lambda_A(v) & \textit{if } v \in V_A \\ \lambda_B(v_B) & \textit{if } v = \langle \cdot, v_B \rangle \in (tr(\prec_A) \times V_B) \end{cases} \\
&\prec' = \prec_A \cup \\
&\quad \left\{ \left\langle v_A, \left\langle \left\langle v_A^1, \cdot \right\rangle, \cdot \right\rangle \right\rangle \mid v_A \preceq_A v_A^1 \right\} \cup \\
&\quad \left\{ \left\langle \left\langle \left\langle \cdot, v_A^2 \right\rangle, \cdot \right\rangle, v_A \right\rangle \mid v_A^2 \preceq_A v_A \right\} \cup \\
&\quad \left\{ \left\langle \left\langle \left\langle \cdot, v_A^2 \right\rangle, \cdot \right\rangle, \left\langle \left\langle w_A^1, \cdot \right\rangle, \cdot \right\rangle \right\rangle \mid v_A^2 \preceq_A w_A^1 \right\} \cup \\
&\quad \left\{ \left\langle \left\langle \left\langle v_A^1, v_A^2 \right\rangle, v_B \right\rangle, \left\langle \left\langle w_A^1, w_A^2 \right\rangle, w_B \right\rangle \right\rangle \mid \left\langle v_A^1, v_A^2 \right\rangle = \left\langle w_A^1, w_A^2 \right\rangle \wedge v_B \prec_B w_B \right\}
\end{aligned} \tag{4.13}$$

$tr(\prec_A)$ is the transitively reduced relationship of $\prec_A$

Example

$$\left\{ a \to b \quad c \right\} \overrightarrow{\chi} \left\{ \begin{array}{c} d \nearrow e \\ d \searrow e \end{array} \right\} = \left\{ a \to d \begin{array}{c} \nearrow e \\ \searrow e \end{array} \begin{array}{c} \searrow b \\ \nearrow b \end{array} \quad c \right\}$$

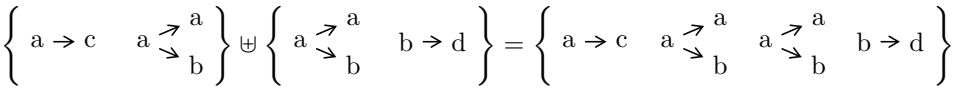
4.4.4. Additive Union, $\uplus$

This binary operator is already known from bags. In contrast to the ordinary union operator, multiplicities are added instead of building the maximum. Thus, each single element of both pomsets – even if there are duplicates – occurs separately in the resulting pomset. This allows to preserve the internal order of the given pomsets. In contrast to the concatenation (see Section 4.4.5, *Concatenation,* $\oplus$, p. 70) no additional ordering is defined.

Definition 6 (Additive Union, $\uplus$)**.** *$A \uplus B$ is a pomset containing all elements in A and B, which preserve the order of those elements and do not add any additional order. Thus, all elements in A are unrelated to those in B.*

$$A \uplus B =_{def} \langle E_A \cup E_B, V_A \cup V_B, \lambda_A \cup \lambda_B, \prec_A \cup \prec_B \rangle \tag{4.14}$$

Example



For bags, the additive union operator works as usual:

$$\{a, a, b\} \uplus \{a, b, c\} = \{a, a, a, b, b, c\}$$

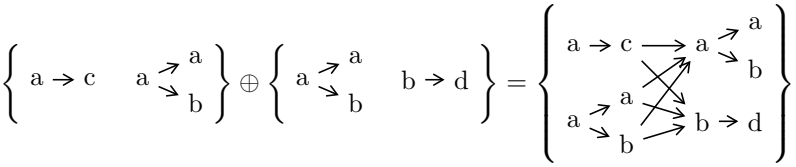
4.4.5. Concatenation, $\oplus$

Whereas additive union is known from bags, this operator is already known from lists (and sequences). As pomsets build a unifying concept for both bags and lists, both operators are sensitive to pomsets. This binary operator therefore represents the second type for unifying two pomsets. The behaviour is quite similar to additive union: It also sums up the multiplicity of elements and the internal order is also preserved. In contrast to additive union, concatenation also preserves the order of the operands as known from concatenating sequences.

Definition 7 (Concatenation, $\oplus$)**.** *$A \oplus B$ is a pomset containing all elements in A and B, which preserve the order of those elements and additionally make up all elements in A smaller than those in B.*

$$A \oplus B =_{def} \langle E_A \cup E_B, V_A \cup V_B, \lambda_A \cup \lambda_B, \prec_A \cup \prec_B \cup (V_A \times V_B) \rangle \tag{4.15}$$

Example



For lists, the concatenation operator works as usual:

$$\langle a, a, b \rangle \oplus \langle a, b, c \rangle = \langle a, a, b, a, b, c \rangle$$

4.4.6. Projection, $\downarrow$

Later on, it will be seen that a symmetric intersection operator for pomsets cannot be defined element by element but has to be defined on connected components. The projection operator should be an adequate substitute for this lack.

In many situations it is required for the elements of an arbitrary resulting pomset to be restricted to a defined set of elements. This is done by the projection operator:

Definition 8 (Projection, $\downarrow$)**.** *$A \downarrow B$ is the pomset A in which all elements are removed that are not in B – independent of their exact multiplicity in B. Order and multiplicities within A are preserved.*

$$
\begin{aligned}
& A \downarrow B =_{def} A \chi r \\
& where \\
& r : E_A \to \mathcal{P}_{pomset}\left(E_A \cap E_B\right), e \mapsto \begin{cases} \{e\} & if e \in B \\ \emptyset & if e \notin B \end{cases} \\
& e \in B \Leftrightarrow \exists v \in V_B : \lambda_B(v) = e
\end{aligned}
\tag{4.16}
$$

Note that the projection operator is not symmetric: While order and multiplicity of the left operand are preserved in the end, order and multiplicity are irrelevant for the second operand. It is therefore sufficient for the right operator to be defined as a set. Nevertheless the operator allows pomsets on the right side but ignores additional information.

Example

$$
\left\{ \begin{array}{ll} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \downarrow \left\{ \begin{array}{ll} \mathrm{b} \nearrow \mathrm{a} \\ \mathrm{b} \searrow \mathrm{d} \end{array} \right\} = \left\{ \begin{array}{ll} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \downarrow \{a, b, d\} = \left\{ \begin{array}{ll} \mathrm{a} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\}
$$

The projection operator does the same for sets as the intersection:

$$
\{a, b, c\} \downarrow \{a, b, d\} = \{a, b, c\} \cap \{a, b, d\} = \{a, b\}
$$

Note that such an equation is not correct for bags!

4.4.7. Complement Projection, $\downarrow^*$

The pomset operator for Complement Projection is defined in an opposite way to projection: Whereas the projection operator removes those elements *not* contained by the right operand, the difference operator removes those elements that *are* contained by the right operand. Apart from that, the two operators are quite similar: Both operators preserve order and multiplicities of the left operand and both operators ignore order and multiplicities of the right operand.

Definition 9 (Complement Projection, $\downarrow^*$). *$A \downarrow^* B$ is the pomset A in which all elements are removed that are in B – independent of its exact multiplicity in B. Order and multiplicities within A are preserved.*

$$A \downarrow^* B =_{def} A \chi r$$

where

$$r : E_A \to \mathcal{P}_{pomset}(E_A \cap E_B), e \mapsto \begin{cases} \{e\} & if\, e \notin B \\ \emptyset & if\, e \in B \end{cases} \tag{4.17}$$

$$e \in B \Leftrightarrow \exists v \in V_B : \lambda_B(v) = e$$

Example

$$\left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \downarrow^* \left\{ \begin{array}{cc} \mathrm{c} \nearrow \mathrm{c} \\ \mathrm{c} \searrow \mathrm{d} \end{array} \right\} = \left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \downarrow^* \{c, d\} = \left\{ \begin{array}{cc} \mathrm{a} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\}$$

4.4.8. Consists Of, $\Subset$

It will be seen that whenever a domain will be specified for a property to be defined (e. g. a property *name* should be a list of strings) that a pomset only consists of elements out of a given set of elements. For sets that can be easily realised by the subset predicate. The corresponding generalisation of this subset predicate for pomsets as defined in Section 4.4.12, *Subset, $\subseteq$*, p. 74 does, however, not even fulfil these requirements for lists:

$$\{a, b\} \subseteq \{a, b, c\}$$

But:

$$\langle a, b \rangle \not\subseteq \{a, b, c\}$$

Because of these facts, the consists-of operator will be defined:

Definition 10 (Consists Of, $\Subset$). *$A \Subset B$ if, and only if, all elements occurring in A also occur in B, independent of order and multiplicity of the elements.*

$$A \Subset B \Leftrightarrow_{def} A \downarrow B = A \tag{4.18}$$

Example

$$\left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \Subset \left\{ \begin{array}{ccc} \mathrm{a} \nearrow \mathrm{b} & & \mathrm{d} \\ \mathrm{a} \searrow \mathrm{c} & & \end{array} \right\}$$

In many situations, the right operand is specified as a set:

$$\left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \Subset \{a, b, c, d\}$$

But:

$$\left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \notin \{a, b\}$$

4.4.9. Union, $\cup$

As pomsets are a generalisation of sets and bags, even the fundamental operators/predicates *Union* $\cup$, *Intersection* $\cap$, and *Subset-Of* $\subseteq$ shall be generalised. This should be done such that the common meaning of those operands for sets and bags is kept unchanged. In addition, the following rules should also be true over pomsets:

$$\begin{gathered} A = B \Leftrightarrow (A \subseteq B \wedge B \subseteq A) \\ (A \cap B) \subseteq A \\ A \subseteq (A \cup B) \end{gathered} \tag{4.19}$$

In order to achieve this, these operators are reduced to those of bags in case of pomsets, whereas the elements of bags are the connected components of the pomsets.

Definition 11 (Union, $\cup$)**.** *$A \cup B$ is a pomset containing the connected components of A and B such that the quantity of each connected component represents the maximum of quantities in each pomset A and B.*

$$A \cup B =_{def} (\kappa A \cup_{bag} \kappa B) \ \chi \ f_{id} \tag{4.20}$$

Example

$$\left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\} \cup \left\{ \begin{array}{ccc} \mathrm{a} \nearrow \mathrm{a} & \mathrm{a} \nearrow \mathrm{a} & \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} & \mathrm{a} \searrow \mathrm{b} & \end{array} \right\} = \left\{ \begin{array}{cccc} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} & \mathrm{a} \nearrow \mathrm{a} & \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} & \mathrm{a} \searrow \mathrm{b} & \end{array} \right\}$$

For bags and sets, the union operator works as usual:

$$\{a, a, b\} \cup \{a, b, c\} = \{a, a, b, c\}$$

4.4.10. Intersection, $\cap$

For the same reason as described for unions in Section 4.4.9, *Union, $\cup$*, p. 73 intersections are defined over pomsets as follows:

Definition 12 (Intersection, $\cap$)**.** *$A \cap B$ is a pomset containing the connected components of A and B such that the quantity of each connected component represents the minimum of quantities in each pomset A and B.*

$$A \cap B =_{def} (\kappa A \cap_{bag} \kappa B) \ \chi \ f_{id} \tag{4.21}$$

Example

$$\left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \end{array} \right\} \cap \left\{ \begin{array}{ccc} \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} & \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} & \mathrm{a} \end{array} \right\} = \left\{ \begin{array}{c} \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \end{array} \right\}$$

For bags and sets, the union operator works as usual:

$$\{a, a, b\} \cap \{a, b, c\} = \{a, b\}$$

4.4.11. Difference, $\setminus$

For the same reason as described for unions in Section 4.4.9, *Union, $\cup$*, p. 73 the difference over pomsets is defined as follows:

Definition 13 (Difference, $\setminus$)**.** *$A \setminus B$ is a pomset containing the connected components of A and B such that the quantity of each connected component represents the subtraction of quantities in each pomset A and B (or zero if the result is negative).*

$$A \setminus B =_{def} (\kappa A \setminus_{bag} \kappa B) \; \chi \; f_{id} \tag{4.22}$$

Example

$$\left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \end{array} \right\} \setminus \left\{ \begin{array}{ccc} \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} & \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} & \mathrm{a} \end{array} \right\} = \left\{ \mathrm{a} \to \mathrm{c} \right\}$$

For bags and sets, the union operator works as usual:

$$\{a, a, b\} \setminus \{a, c\} = \{a, b\}$$

4.4.12. Subset, $\subseteq$

For the same reason as described for unions in Section 4.4.9, *Union, $\cup$*, p. 73 the difference over pomsets is defined as follows:

Definition 14 (Subset, $\subseteq$)**.** *$A \subseteq B$ is true if, and only if, the quantity of each connected component in pomset A is greater than in pomset B.*

$$A \subseteq B \Leftrightarrow_{def} \kappa A \subseteq_{bag} \kappa B \tag{4.23}$$

Example

$$\left\{ \mathrm{a} \to \mathrm{c} \right\} \subseteq \left\{ \begin{array}{cc} \mathrm{a} \to \mathrm{c} & \begin{array}{c} \mathrm{a} \nearrow \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} \end{array} \end{array} \right\}$$

But:

$$\{a\} \not\subseteq \left\{ \begin{array}{ll} \mathrm{a} \to \mathrm{c} & \mathrm{a} \nearrow \mathrm{a} \\ & \mathrm{a} \searrow \mathrm{b} \end{array} \right\}$$

For bags and sets, the subset operator works as usual:

$$\{a, b\} \subseteq \{a, a, b\}$$

4.4.13. Cardinality, $|\cdot|$

In order to get the size of a set or the length of a list, the cardinality operator is defined.

Definition 15 (Cardinality, $|\cdot|$)**.** *$|A|$ returns the total number of elements within pomset A taking duplicates into account.*

$$|A| =_{def} |V_A| \tag{4.24}$$

Example

$$\left| \left\{ \begin{array}{ll} \mathrm{a} \nearrow \mathrm{b} \to \mathrm{b} & \mathrm{c} \\ \mathrm{a} \searrow & \end{array} \right\} \right| = 5$$

4.4.14. Depth, $\|\cdot\|$

The depth operator is related to the cardinality operator which has previously been introduced. In contrast to cardinality, the depth operator takes the order of the elements into account: The depth of a pomset equals the length of the longest path of the transitively reduced graph representing the pomset. For a formal definition, the *depth of a vertex* will be defined first:

Definition 16 (Depth of a vertex v_i, $\|v_i\|$)**.** *For a given pomset A, the depth of a vertex v_i, denoted by $\|v_i\|$, returns the length of the longest path from the root vertex of the corresponding connected component to the vertex v_i. Hereby, the length of a path meets the number of nodes therein. The depth of root vertices themselves therefore equals 1.*

Now, the *depth of a pomset* can be defined:

Definition 17 (Depth, $\|\cdot\|$)**.** *$\|A\|$ returns the maximum depth over all vertices $v_i \in V_A$. For empty pomsets, $\|\varnothing\| = 0$ holds.*

$$\|A\| =_{def} \begin{cases} \max_{v_i \in V_A} \|v_i\| & if A \neq \varnothing \\ 0 & if A = \varnothing \end{cases} \tag{4.25}$$

Example

$$\left\| \left\{ \begin{array}{ll} \mathrm{a} \nearrow \mathrm{b} \to \mathrm{b} & \mathrm{c} \\ \mathrm{a} \searrow & \end{array} \right\} \right\| = 3$$

4.4.15. Multiplicity, $\in_m$

The multiplicity operator represents the generalisation of the *element-of* predicate of the set theory. While the *element-of* predicate only indicates whether an element is part of a pomset or not, the multiplicity operator returns the number of occurrences. If the multiplicity is zero, the element is not an element of the pomset.

Definition 18 (Multiplicity, $\in_m$)**.** *$a \in_m A$ returns the number of occurrences of a in the pomset A.*

$$a \in_m A =_{def} |A \downarrow \{a\}| \tag{4.26}$$

Example

$$b \in_m \left\{ \begin{array}{ccc} & \text{b} \rightarrow \text{b} & \\ \text{a} \nearrow & & \text{c} \\ \text{a} \searrow & & \end{array} \right\} = 2$$

4.4.16. Path, π

The partial order of a pomset can be interpreted in different ways. One interpretation states that if two elements of a pomset are unordered to each other, these two elements are seen as alternatives.

In case an encoding shall be defined such that an "a" or "b" stands within round brackets, it can be defined by the following pomset:

$$\left\{ \begin{array}{ccccc} & & \text{"a"} & & \\ \text{"("} & \nearrow & & \searrow & \text{")"} \\ & \searrow & & \nearrow & \\ & & \text{"b"} & & \end{array} \right\} \tag{4.27}$$

Again, the interpretation of this pomset is that "a" and "b" are encoded alternatively. Thus the explicit codings are $\langle$"(", "*a*", ")"$\rangle$ and $\langle$"(", "*b*", ")"$\rangle$. The *path*-operator – denoted as a π – should return this interpretation explicitly:

$$\pi \left\{ \begin{array}{ccccc} & & \text{"a"} & & \\ \text{"("} & \nearrow & & \searrow & \text{")"} \\ & \searrow & & \nearrow & \\ & & \text{"b"} & & \end{array} \right\} = \left\{ \begin{array}{c} \text{"("} \rightarrow \text{"a"} \rightarrow \text{")"} \\ \text{"("} \rightarrow \text{"b"} \rightarrow \text{")"} \end{array} \right\}$$

The *path* operator therefore returns a pomset of which each connected component is totally ordered and represents one possible encoding.

Definition 19 (Path, π)**.** *πA is a pomset of which each (totally ordered) connected component represents one path by the transitively reduced graph representing the pomset A. Equal connected components are preserved as duplicates within the resulting pomset.*

Example

$$\pi \left\{ \begin{array}{ccc} & \nearrow \mathrm{a} \searrow & \\ \mathrm{a} & & \mathrm{b} \\ & \searrow \mathrm{b} \nearrow & \end{array} \quad \begin{array}{ccc} & \nearrow \mathrm{d} & \\ \mathrm{c} & & \\ & \searrow \mathrm{d} & \end{array} \right\} = \left\{ \begin{array}{c} \mathrm{a} \to \mathrm{a} \to \mathrm{b} \\ \mathrm{a} \to \mathrm{b} \to \mathrm{b} \\ \mathrm{c} \to \mathrm{d} \\ \mathrm{c} \to \mathrm{d} \end{array} \right\}$$

4.4.17. Totalise, τ

In the previous section (Section 4.4.16, *Path, π*, p. 76), the partial order was interpreted as alternatives. Another, more straight-forward interpretation states that if two elements of a pomset are unordered to each other, these two elements are not seen as alternatives, but must both occur in e. g. an encoding and only the order thereof is not fixed.

In case an encoding shall be defined such that an "a" and a "b" stands within round brackets in an arbitrary order, this can be defined by the same pomset (4.27) from the last section.

This time the interpretation states that the explicit encodings are $\langle$"(", "*a*", "*b*", ")"$\rangle$ and $\langle$"(", "*b*", "*a*", ")"$\rangle$. The *totalise*-operator – denoted as a τ – should return this interpretation explicitly:

$$\tau \left\{ \begin{array}{ccc} & \nearrow \text{"a"} \searrow & \\ \text{"("} & & \text{")"} \\ & \searrow \text{"b"} \nearrow & \end{array} \right\} = \left\{ \begin{array}{c} \text{"("} \to \text{"a"} \to \text{"b"} \to \text{")"} \\ \text{"("} \to \text{"b"} \to \text{"a"} \to \text{")"} \end{array} \right\}$$

The *totalise* operator therefore returns a pomset of which each connected component is totally ordered and represents one possible encoding.

Definition 20 (Totalise, τ)**.** *τA is a pomset of which each (totally ordered) connected component represents one possible totalisation of the ordering of the pomset A. Equal connected components do not occur.*

Example

$$\tau \left\{ \begin{array}{ccc} & \nearrow \mathrm{a} \searrow & \\ \mathrm{a} & & \mathrm{c} \\ & \searrow \mathrm{b} \nearrow & \end{array} \right\} = \left\{ \begin{array}{c} \mathrm{a} \to \mathrm{a} \to \mathrm{b} \to \mathrm{c} \\ \mathrm{a} \to \mathrm{b} \to \mathrm{a} \to \mathrm{c} \end{array} \right\}$$

4.4.18. Sub-Pomset, $[i; j]\cdot$

The *Sub-Pomset* operator is a generalisation of a common operator on sequences/lists: namely an operator to get a sub-sequence beginning and ending at a dedicated index. For a given sequence $\langle$"(", "*a*", "*b*", "*c*", ")"$\rangle$, for example, the three inner characters shall be obtained. The *Sub-Pomset* operator should allow such a use-case:

$$[2; 4] \langle \text{"("}, \text{"}a\text{"}, \text{"}b\text{"}, \text{"}c\text{"}, \text{")"} \rangle = \langle \text{"}a\text{"}, \text{"}b\text{"}, \text{"}c\text{"} \rangle$$

The generalisation of the operator is based on the depth of a vertex as defined in Definition 16:

Definition 21 (Sub-Pomset, $[i;j]\cdot$)**.** *$[i;j]A$ is a pomset that is restricted to the vertices v of A, such that $i \leq \|v\| \leq j$. If $j = *$, no upper limit is defined.*

$$\begin{aligned} [i;j]A &=_{def} \langle E_A, \{v \in V_A \mid i \leq \|v\| \leq j\}, \lambda_A, \prec_A \rangle \\ [i;*]A &=_{def} [i;\|A\|]A \end{aligned} \tag{4.28}$$

Example

$$[2;4]\left\{\begin{array}{l} \mathrm{a} \nearrow \\ \mathrm{a} \searrow \end{array} \begin{array}{l} \mathrm{b} \to \mathrm{c} \to \mathrm{c} \searrow \\ \mathrm{b} \end{array} \mathrm{d} \quad \mathrm{e} \to \mathrm{f} \right\} = \left\{\begin{array}{l} \mathrm{b} \to \mathrm{c} \to \mathrm{c} \\ \mathrm{b} \\ \mathrm{f} \end{array}\right\}$$

4.4.19. First, 1

The *First* operator is a special case of the *Sub-pomset* operator. It returns the first element for sequences. The general definition for a pomset is defined as follows:

Definition 22 (First, 1)**.** *$1A$ is a set of those elements out of A which have no predecessor according to the partial order in A.*

$$1A =_{def} [1;1]A \tag{4.29}$$

Example

$$1\left\{\begin{array}{l} \mathrm{a} \nearrow \\ \mathrm{a} \searrow \end{array} \begin{array}{l} \mathrm{b} \to \mathrm{c} \to \mathrm{c} \searrow \\ \mathrm{b} \end{array} \mathrm{d} \quad \mathrm{e} \to \mathrm{f} \right\} = \{a, e\}$$

4.4.20. Order Inverse, $\gtrless$

Definition 23 (Order Inverse, $\gtrless$)**.** *$\gtrless A$ is the same pomset as A except that the order is inversed. If A is completely unordered, this operator has no effect.*

$$\gtrless A =_{def} \langle E_A, V_A, \lambda_A, \{(v,w) \mid w \prec_A v\} \rangle \tag{4.30}$$

Example

$$\gtrless \left\{\begin{array}{l} \mathrm{a} \nearrow \\ \mathrm{a} \searrow \end{array} \mathrm{c} \quad \begin{array}{l} \mathrm{a} \nearrow \\ \mathrm{a} \searrow \end{array} \mathrm{c} \quad \mathrm{c} \right\} = \left\{\begin{array}{l} \mathrm{c} \searrow \\ \mathrm{b} \nearrow \end{array} \mathrm{a} \quad \begin{array}{l} \mathrm{c} \searrow \\ \mathrm{b} \nearrow \end{array} \mathrm{a} \quad \mathrm{c} \right\}$$

4.4.21. Order Destroy, μ

Definition 24 (Order Destroy, μ)**.** *μA deletes the order of the pomset A. Thus, the result is a bag.*

$$\mu A =_{def} \langle E_A, V_A, \lambda_A, \emptyset \rangle \tag{4.31}$$

Example

$$\mu \left\{ \begin{array}{ccc} \mathrm{a} \nearrow \mathrm{c} & & \mathrm{a} \nearrow \mathrm{c} \\ \mathrm{a} \searrow \mathrm{b} & & \mathrm{a} \searrow \mathrm{b} \end{array} \quad \mathrm{c} \right\} = \{a, a, b, b, c, c, c\}$$

4.4.22. Duplicate Destroy, ϵ

Definition 25 (Duplicate Destroy, ϵ)**.** *ϵA is a pomset wherein all vertices $v \in V_A$ being mapped to the same label by λ_A will collapse to form one unique vertex with that label, and wherein the order on these new objects is the order consistent with that of the sources of the elements. Thus, the result is a poset.*

$$\begin{array}{l} \epsilon A =_{def} \langle E_A, E_A, f_{id}, \prec' \rangle \\ \textit{where} \\ \quad f_{id} \textit{ is the identity on } E_A \\ \quad \prec' = \left\{ (e, e') \;\middle|\; \begin{array}{ll} & (\exists v, v' : (\lambda_A(v) = e) \wedge (\lambda_A(v') = e') \wedge (v \prec_A v')) \\ \wedge & (\forall v, v' : (\lambda_A(v) = e \wedge \lambda_A(v') = e') \Rightarrow \neg(v' \prec_A v)) \end{array} \right\} \end{array} \tag{4.32}$$

Example

$$\epsilon \left\{ \begin{array}{ccc} \mathrm{a} \nearrow \mathrm{c} & & \mathrm{a} \nearrow \mathrm{c} \\ \mathrm{a} \searrow \mathrm{b} & & \mathrm{a} \searrow \mathrm{b} \end{array} \quad \mathrm{c} \to \mathrm{a} \right\} = \left\{ \mathrm{a} \to \mathrm{b} \quad \mathrm{c} \right\}$$

4.4.23. Duplicate Destroy Over CCs, $\epsilon_\subset$

Definition 26 (Duplicate Destroy Over CCs, $\epsilon_\subset$)**.** *$\epsilon_\subset A$ is pomset wherein duplicate connected components are removed.*

$$\epsilon_\subset A =_{def} (\epsilon \kappa A) \, \chi \, f_{id} \tag{4.33}$$

Example

$$\epsilon_\subset \left\{ \begin{array}{ccc} \mathrm{a} \nearrow \mathrm{c} & & \mathrm{a} \nearrow \mathrm{c} \\ \mathrm{a} \searrow \mathrm{b} & & \mathrm{a} \searrow \mathrm{b} \end{array} \quad \mathrm{c} \to \mathrm{a} \right\} = \left\{ \begin{array}{cc} \mathrm{a} \nearrow \mathrm{c} & \mathrm{c} \to \mathrm{a} \\ \mathrm{a} \searrow \mathrm{b} & \end{array} \right\}$$

4.4.24. Predicates for pomset types

As defined in Section 4.1, *Relationship between different types of sets*, p. 61, a set of different pomset types exists. In order to be able to check easily whether a pomset is of a special type, the following predicates will be defined:

Definition 27 (Predicates for pomset types)**.**

$$\begin{aligned} \text{empty?}\ A &\Leftrightarrow_{def} |A| = 0 \\ \text{singleton?}\ A &\Leftrightarrow_{def} |A| \leq 1 \\ \text{set?}\ A &\Leftrightarrow_{def} \mu\epsilon A = A \\ \text{bag?}\ A &\Leftrightarrow_{def} \mu A = A \\ \text{list?}\ A &\Leftrightarrow_{def} \tau A = A \\ \text{toset?}\ A &\Leftrightarrow_{def} (\tau A = A) \wedge (\epsilon A = A) \\ \text{poset?}\ A &\Leftrightarrow_{def} \epsilon A = A \end{aligned} \tag{4.34}$$

Example

$$\begin{aligned} \text{toset?}\ \langle a, b, c \rangle &= \top \\ \text{bag?}\ \langle a, b, c \rangle &= \bot \end{aligned}$$

4.5. Running Example

As described in Section 3.3, *Informal description of the modelling language*, p. 49, the present exemplary modelling language describes weak-causal dataflow networks. Due to the weak causality thereof upon an evaluation of such a network, the order of said evaluation is crucial. As has also been described, cycles are not allowed within a network except a *pre*-Block is located inbetween.

Such an evaluation order is, in general, not total all the time as some sub-components do not depend on each other whereas others do. The result is a partial order over all sub-components. As already mentioned, the *pre*-Blocks play a special role: As it has been possible to create cycles by using that block, these cycles need to be broken up.

By using pomsets, the evaluation order can easily be described by the following pomset:

$$evalOrder = \left\{ \begin{array}{ccc} \text{dT0} & \longrightarrow & \text{mult0} \searrow \\ & & \text{add0} \\ & & \text{pre0} \nearrow \end{array} \right\} \tag{4.35}$$

The pomset states that $dT0$ must be evaluated before the evaluation of $mult0$, $add0$ cannot be evaluated before the evaluation of $mult0$ and $pre0$. The evaluation order of $dT0$ and $pre0$ is left open, for example.

The following chapters will show, how to calculate this pomset formally by inspecting the model in terms of abstract syntax.

Chapter 5

Models as Abstract Words

In order to formalise metamodelling languages e. g. for comparing two different types thereof, models will have to be introduced first. Models are often defined as *instances of metamodels*. Independent of the concrete meaning of what an instance is, this definition leads to the undesired strong coupling between model and metamodel. The idea of formal languages will be followed instead: The definition of the principal superstructure of words (a sequence of terminal symbols) is completely independent of the way of defining the language. Influenced by that idea, a general superstructure for models will be defined, independent of a metamodelling language. A model defined within such a superstructure will be called an *abstract word*. Whereas words, as defined in formal languages, represent a model in a concrete textual way (which will be called *textual concrete words* for a better distinction), abstract words concentrate on the underlying *concepts* including their *properties* which describe the links inbetween. In the following section, a formal definition of these abstract words will be provided.

Contents

5.1. M-graphs (Model-graphs)

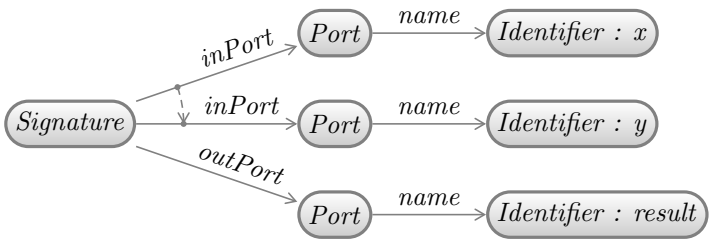
While textual concrete words are defined as a sequence of symbols out of an alphabet, an abstract word will be defined by a special type of directed graph where the vertices are labelled by *concepts* and the directed edges are labelled by *properties*. Thus, the basic elements for abstract words are not characters but a set of concepts and properties. According to that, an *abstract alphabet* is defined as follows:

Definition 28 (Abstract alphabet)**.** *An* abstract alphabet Σ *is a pair* $\langle C, P \rangle$ *where* C *is a partially ordered set of* concepts *and* P *is a set of* properties*. Furthermore, the two sets* C *and* P *must be disjoint. The partial order over concepts represents the refinement hierarchy and thus reflects the specialisation/generalisation principle. When a concept* A *is a specialisation of the concept* B *we denote* $A \prec B$*.*

In the running example there e. g. exists the refinement relationship $Network \prec Component$ which means that every network is a component. For a better readability concepts will always be written by starting with a capital letter, whereas properties begin with a small letter. Before providing the complete formal definition of an abstract word, a short example (Figure 5.1) shall be introduced describing the signature for e. g. an *add*-block out of our running example as described in Section B.2, *Exemplary Model: Basic Library*, p. 312 by both a textual concrete as well as an abstract word which are closely related to graphs wherein the edges are partially ordered (which is denoted by dashed arrows).

```
(x:Any, y:Any -> result:Any)
```

(a) textual concrete word



(b) abstract word

Figure 5.1.: Example: abstract and textual concrete word of a signature

In particular when having a look at the alphabets, the difference becomes transparent: For the textual concrete word the alphabet of terminal symbols in the form of tokens is $\Sigma = \{$"(", ")", ",", ":", "->", "Any", "x", "y", "result"$\}$, whereas the abstract alphabet for the abstract word is $\Sigma = \langle \{Signature, Port, Identifier\}, \{inPort, outPort, name\} \rangle$. While the textual concrete word is just a collection of characters and strings, the abstract word represents the structure of the language in an explicit way. Even a taxonomy can be derived. The concepts *Signature*, *Port*, and *Identifier* label the vertices, whereas the properties *inPort*, *outPort*, and *name* label the edges. The type *Any*, denoted in the textual concrete word, is skipped in the abstract word, because the present running example is untyped in general. Thus, the reserved word *Any* does not contain any additional information and is therefore skipped in abstract syntax.

Figure 5.1(b) illustrates already that abstract words are closely related to traditional graphs with their common definitions for e. g. *vertex* or *label*. As usual for graphs, a set of vertices is necessary in addition to the set of concepts and properties. In some situations it is helpful to explicitly write down the element out of the set of vertices in the graphical representation. The vertex element is denoted after the concept name, separated by a colon. In the present example, this is done for the three identifier nodes. This procedure could, of course, be performed for all vertices as shown in Figure 5.2. In case of identifiers, the vertex element is named as the identifier itself. As described in Section 5.5.3, *Attributes*, p. 89, this notation is only a shortcut for modelling attributes such as strings – as in this case.

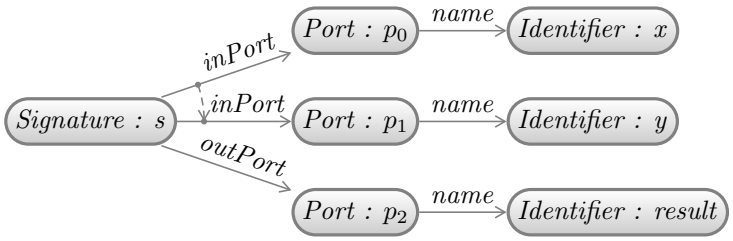


Figure 5.2.: Example: abstract word including names for each vertex

Note that the implicit meaning of the concepts and properties is not defined in a formal way herein. Thus, a replacement of the abstract alphabet by e. g. $\Sigma = \langle \{A, B, C\}, \{a, b, c\} \rangle$ including the replacement of the vertex names by c_0, c_1, c_2 will render the abstract word (as shown in Figure 5.3) meaningless although the structure is preserved. Such a replacement can be interpreted as an obfuscation:

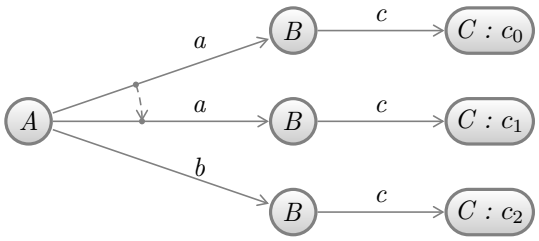


Figure 5.3.: Example: obfuscated abstract word of a signature

As the input ports should be defined in an ordered way, the order in the present example (Figure 5.1(b)) is reflected by a dashed arrow between the two edges. Abstract words should generally allow a partial order between edges wherein both of which have the same source vertex and are labelled by the same property. Thus, an order between edges that are not labelled by the same property is not allowed. In the present example, an arrow (thus an ordering) between one of the input ports and the output port, for example, is not allowed. Similar to the notation for pomsets only the transitively reduced arrows are shown.

According to these requirements, the set of edges labelled by the same property by a function mapping each vertex to a pomset of vertices, can be specified from a mathematical point of view. This function is called an *edge function*:

Definition 29 (Edge function)**.** *An edge function over a set of vertices V is a total function mapping each vertex to a pomset of target vertices. The set of all edge functions is denoted by $\mathcal{E}_V = V \to \mathcal{P}_{pomset}(V)$.*

According to the present example, e. g. for the edge function which describes the property *inPort* can be defined by:

$$\left\{ \begin{array}{c} (s \mapsto \{p_0 \to p_1\}), (p_0 \mapsto \varnothing), (p_1 \mapsto \varnothing), (p_2 \mapsto \varnothing), \\ (x \mapsto \varnothing), (y \mapsto \varnothing), (result \mapsto \varnothing) \end{array} \right\}$$

Thus, for the vertex s the given edge-function evaluates to the two input ports. For all other vertices it evaluates to an empty set, indicating that there is no outgoing edge labelled by *inPort* for these vertices.

Thanks to the edge functions, a formal definition of the superstructure for graphs such as given in Figure 5.1(b) becomes possible:

Definition 30 (M-graph)**.** *A* model graph (M-graph) *is a* Σ*-labelled, directed, partially ordered multi-graph which is described as a quadruple* $\langle \Sigma, V, type, edge \rangle$ *wherein*

- $\Sigma = \langle C, P \rangle$ *is an abstract alphabet,*
- V *is a set of vertices which is disjoint to both sets* C *and* P*,*
- $type : V \to C$ *is a total function which labels each node with a concept out of the alphabet, and*
- $edge : P \to \mathcal{E}_V$ *is a total function which assigns an edge function out of* $\mathcal{E}_V$ *to each property.*

For a given M-graph a directed multi-graph $\langle V, E \rangle$ *with the identical set of vertices can be derived by defining the bag* E *as follows:*

$$E =_{def} \left\{ (v_1, v_2) \in V \times V \;\middle|\; \sum_{p \in P} v_2 \in_m edge(p)(v_1) \right\} \tag{5.1}$$

Note that the notation used is defined in (4.8). The mapping to directed multi-graphs allows for an adoption of terms and definitions – such as *cycle* and *path* – from graph-theory to M-graphs.

In some situations, an inverse *type*-function is useful. It is defined as:

$$type^{-1} : C \to V,\ c \mapsto \{v \in V \mid type(v) = c\} \tag{5.2}$$

According to Definition 30, the abstract word shown in Figure 5.1(b) can now be written as an M-graph $\omega = \langle \Sigma, V, type, edge \rangle$ wherein $\Sigma = \langle C, P \rangle$ and

- $C = \{Port, Signature, Identifier\}$
- $P = \{inPort, outPort, name\}$
- $V = \{s, p_0, p_1, p_2, x, y, result\}$
- $type = \left\{ \begin{array}{c} (s \mapsto Signature), (p_0 \mapsto Port), (p_1 \mapsto Port), (p_2 \mapsto Port), \\ (x \mapsto Identifier), (y \mapsto Identifier), (result \mapsto Identifier) \end{array} \right\}$
- $edge = \left\{ \begin{array}{l} \left(inPort \mapsto \left\{ \begin{array}{c} (s \mapsto \{p_0 \to p_1\}), (p_0 \mapsto \varnothing), (p_1 \mapsto \varnothing), (p_2 \mapsto \varnothing), \\ (x \mapsto \varnothing), (y \mapsto \varnothing), (result \mapsto \varnothing) \end{array} \right\} \right), \\ \left(outPort \mapsto \left\{ \begin{array}{c} (s \mapsto \{p_2\}), (p_0 \mapsto \varnothing), (p_1 \mapsto \varnothing), (p_2 \mapsto \varnothing), \\ (x \mapsto \varnothing), (y \mapsto \varnothing), (result \mapsto \varnothing) \end{array} \right\} \right), \\ \left(name \mapsto \left\{ \begin{array}{c} (s \mapsto \varnothing), (p_0 \mapsto \{x\}), (p_1 \mapsto \{y\}), (p_2 \mapsto \{z\}), \\ (x \mapsto \varnothing), (y \mapsto \varnothing), (result \mapsto \varnothing) \end{array} \right\} \right) \end{array} \right\}$

Note that the partially ordered set C in this example is unordered. As is usual in the context of graph theory, the concrete set of vertices is irrelevant. An equivalence over M-graphs will therefore be defined in Definition 31:

Definition 31 (M-graph equivalence)**.** *Two M-graphs* $\omega_1 = \langle \Sigma_1, V_1, type_1, edge_1 \rangle$ *and* $\omega_2 = \langle \Sigma_2, V_2, type_2, edge_2 \rangle$ *are equivalent (denoted by* $\omega_1 \cong \omega_2$*) if, and only if, a bijection* $\varphi : V_1 \rightarrow_{bij} V_2$ *exists, such that*

- $\forall v \in V_1 : type_1(v) = type_2(\varphi(v))$,
- $\forall p \in P_1 \cap P_2, v \in V_1 : edge_1(p)(v) \,\chi\, \varphi = edge_2(p)(\varphi(v))$,
- $\forall p \in P_1 \setminus P_2, v \in V_1 : edge_1(p)(v) = \varnothing$, *and*
- $\forall p \in P_2 \setminus P_1, v \in V_2 : edge_2(p)(v) = \varnothing$.

Note that φ can be used as a replacement function for the pomset expansion operator, as single values and singleton pomsets, as introduced in Section 4.1, *Relationship between different types of sets*, p. 61, are treated in a unique way.

According to this definition, unused concepts and properties can be added to the alphabet without influencing the equivalence relation. This characteristic of the abstract alphabet is similar to that of formal languages: A word does not change when adding additional (unused) symbols to the alphabet. Having introduced M-graphs, a formal definition of the term *Abstract word* will now be possible:

Definition 32 (Abstract word)**.** *An* abstract word *is an M-graph.*

5.2. Defining M-graphs without using pomsets

In order to illustrate the relationship to traditional graph theory, we will show in this section how to define M-graphs without using pomsets. Instead of using pomsets as done in Definition 30, the following definition should only use the concepts out of the traditional graph theory. This can be easily achieved by defining an explicit set of edges E combined with two functions α and β with maps these edges to the according source and target vertices. As done in the original definition, the function *type* defines the labels for vertices out of the set of concepts C. In a similar way the edges are labelled by the function *prop* with a property out of the set of properties P. Finally, the ordering is modelled explicitly by a partial order over the set of edges E. Since the partial ordering is only allowed between edges which have the identical source vertex and the same property label, an additional constraint is defined for the order.

Definition 33 (M-graph without using pomsets)**.** *A* model graph (M-graph) *is a* Σ*-labelled, directed, partially ordered multi-graph which is described as a 7-tuple* $\langle \Sigma, V, E, \alpha, \beta, type, prop, \sqsubseteq \rangle$ *wherein*

- $\Sigma = \langle C, P \rangle$ *is an abstract alphabet,*
- V *is a set of vertices,*
- E *is a set of nodes,*
- $\alpha : E \rightarrow V$ *is a total function which defines the source vertex for each edge,*
- $\beta : E \rightarrow V$ *is a total function which defines the target vertex for each edge,*
- $type : V \rightarrow C$ *is a total function which labels each node with a concept out of the alphabet,*

- $prop : V \to P$ *is a total function which labels each edge with a property out of the alphabet,*
- $\sqsubseteq \subseteq \{\langle e_1, e_2\rangle \in E \times E \,|\, \alpha(e_1) = \alpha(e_2) \wedge prop(e_1) = prop(e_2)\}$ *is the partial order over edges which have the identical source vertex and the same property label.*

Based on this definition we now want to compare both ways of describing M-graphs by a common use-case in the metamodelling domain: For a given vertex $v \in V$ and a given property $p \in P$ we want to get all target vertices including the correct ordering. By example, we want to find out all input ports for a given signature in the right order. For an abstract word as defined in Figure 5.2 the given vertex v is the signature s, and the given property $p = inPort$. The expected result would be $\{p_0 \to p_1\}$. When using Definition 30 the requested function is defined as follows:

$$\begin{array}{lcl} V \times P & \to & \mathcal{P}_{pomset}(V) \\ \langle v, p\rangle & \mapsto & edge(p)(v) \end{array} \tag{5.3}$$

When using Definition 33 the requested function can not return a set of vertices since both duplicates and ordering should be preserved. Instead, we have to go back to a set of edges which are unique on the one hand and are ordered on the other.

$$\begin{array}{lcl} V \times P & \to & \mathcal{P}_{set}(E) \\ \langle v, p\rangle & \mapsto & \{e \in E \,|\, \alpha(e) = v \wedge prop(e) = p\} \end{array} \tag{5.4}$$

Let $F \subseteq \mathcal{P}_{set}(E)$ be the resulting set of the function. In order to get the full information additionally the set of vertices V, the mapping function α, and the partial order $\sqsubseteq$ is necessary. Finally, the quadruple $\langle V, F, \alpha, \sqsubseteq\rangle$ builds the wanted result. When comparing this quadruple with the definition for pomsets (Definition 1), it shapes up exactly as a pomset while V meets the basic set of elements, F meets the set of vertices of the pomset, α meets the pomset labelling function, and finally $\sqsubseteq$ meets the partial ordering of the pomset.

5.3. Graph-like notation for Abstract Words

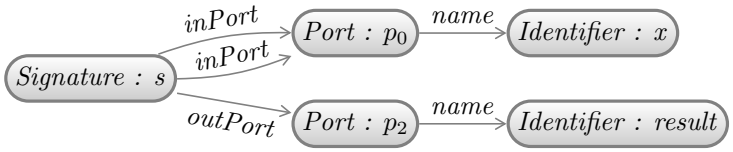
As can be seen in Figure 5.1(b), a traditional graph notation is used for visualising abstract words. The edges are labelled by properties. The label of each node always shows the assigned concept. If the vertex out of V is of interest for a node, it can optionally be shown after the concept, separated by a colon. The node labelled by $String : x$, for example, implies that the concept is $String$ and the vertex element is x.

Strictly speaking, the vertex element x has nothing to do with the fact that this element should represent a string "x". In fact, the two nodes $String : x$ and $String : y$, for example, out of the example are equal as the concrete vertex element is irrelevant. This notation is basically a shortcut to avoid an explicit modelling of the strings "x", "y" or "result". Section 5.5.3, *Attributes*, p. 89 shows the realisation thereof.

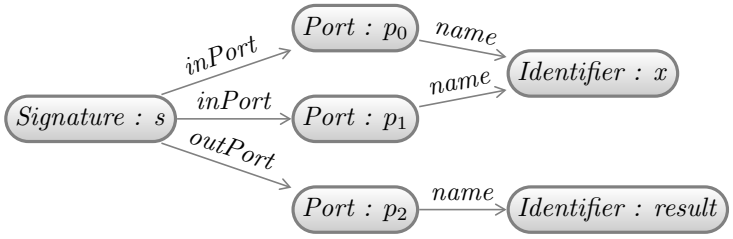
5.4. Node equivalence

According to Definition 30, a property may contain duplicates, because the co-domain of edge functions is $\mathcal{P}_{pomset}(V)$. As edges are defined over the set of vertices, duplicates in the

resulting pomset of an edge-function occur if, and only if, multiple edges from one node to another labelled by the same property exist.



(a) Signature with duplicate port p_0



(b) Signature with equal ports p_0, p_1

Figure 5.4.: Example: Signature modeled with duplicate or same port

In Figure 5.4(a) a signature with a duplicate port is shown. In Figure 5.4(b) the same signature is modelled but without having multiple edges from the signature to one port. For simplicity, the order of the input ports is skipped herein.

Although the signature in Figure 5.4(b) has duplicate ports, it is not possible to detect them by just inspecting the two vertices p_0 and p_1: Informally spoken, two ports can be seen as equal if the "internal structure" is the same. Formally, the internal structure consists of all vertices that can be reached by a respective vertex. In order to achieve a formal definition, *reachable sub-graphs* will be introduced.

The idea behind this definition is that two nodes are equal if the sub-graph, which can be reached by a respective vertex, is equal. Herein a vertex can be reached if it can be navigated thereto transitively. In Figure 5.4, for example, the signature can reach every vertex while a port cannot reach the signature or the other ports. As the sub-graphs, which can be reached by the two ports p_0 and p_1 in Figure 5.4(b), are exactly the same, the two ports should be seen as being equal. Formally, it is defined as follows:

Definition 34 (Reachable sub-graph)**.** *For a given M-graph* $\omega = \langle \Sigma, V, type, edge \rangle$ *and a node* $v \in V$ *the reachable sub-graph is defined for* v *in* ω *as* $\mathbf{rsub}_\omega(v) =_{def} \langle \Sigma, V_R, type \cap (V_R \to C), edge \cap (P \to \mathcal{E}_{V_R}) \rangle$*, wherein* V_R *is the set of all reachable nodes starting from* v*, thus* $V_R = \{ v' \in V \mid \text{it exists a path } v \xrightarrow{*} v' \text{ in } \omega \}$.

Based on this definition, the node equivalence can be defined:

Definition 35 (Node equivalence)**.** *Given a M-graph* $\omega = \langle \Sigma, V, type, edge \rangle$*, two vertices* $v_1, v_2 \in V$ *are equal (denoted by* $v_1 \simeq v_2$*), if, and only if, a bijection* φ *exists, representing a M-graph equivalence between the two reachable sub-graphs, such that* $\mathbf{rsub}_\omega(v_1) \cong \mathbf{rsub}_\omega(v_2)$*, and the bijection* φ *for this M-graph equivalence maps* v_1 *to* v_2*, thus* $\varphi(v_1) = v_2$.

Note that according to these definitions, two equal nodes may also see each other, such as

in Figure 5.5.

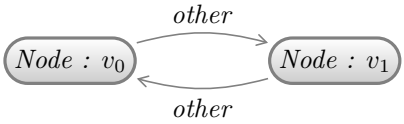


Figure 5.5.: Example: Equal nodes seeing each other

Please keep in mind that due to this definition for equality of nodes said equality of nodes will be influenced by each and every outgoing edge which is added to a model. This will particularly help taking a decision about which direction an association should have and whether an association should be bidirectional or not. Please refer also to Section 5.5.1, *Bidirectional associations*, p. 88.

5.5. Mapping established metamodelling concepts to abstract words

In the following section, it shall be outlined of how to model usual constructs of the metamodelling domain by abstract words. It will be shown that the definition for M-graphs in Definition 30 is powerful enough to model all desired constructs, it shall, however, also be shown what the restrictions are. In particular they are *Bidirectional Associations*, *Compositions*, and *Attributes*.

Note that at this point, it is not metamodels that are discussed: We are still talking about models. So when it's, for example, about compositions, it will not be discussed of how to model them within a metamodel, but within a model.

In this situation the question arises whether a construct is explicitly represented in a model or not. If not, the construct is only represented within the metamodel. In many situations it is a combination of both. In the case of multiplicities, for example, the concrete cardinality of a pomset can, on the one hand, be read out of the abstract word. On the other hand, the restriction of what valid cardinalities for a given property are, is only represented in the metamodel.

5.5.1. Bidirectional associations

Bidirectional associations in metamodelling represent links between nodes that are navigable in both directions. Additional properties *sourceChannel* and *destChannel* representing the other direction of the association can be assumed, for example, for the $fromPort$- and *toPort*-property of our exemplary model. As edges in M-graphs are always directed, a bidirectional link is represented by two separate edges. Figure 5.6 illustrates a bidirectional link between channels and ports: The source port p_0 is connected to the destination port p_1 via a channel *ch*.

Note that two properties like these are not marked as being the opposite of each other. Additionally, the condition that for each property an opposite one exists is not guaranteed at the level of abstract words. Therefore an adequate construct within the metamodel will be necessary. As mentioned in Section 5.4, *Node equivalence*, p. 86 above, modelling a link in a bidirectional way has an influence on the node equivalence as defined in Definition 35.

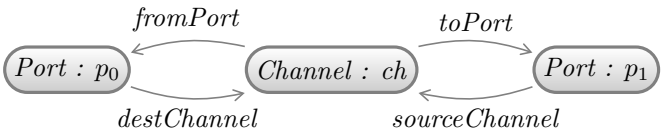


Figure 5.6.: Example: Bidirectional link between ports and channels

5.5.2. Compositions

A *Composition* is a special kind of association wherein one node is seen as *a part of* another one. A port can be seen as a part of a signature, for example. In order to express this fact in an abstract word explicitly, a special property *composite* is introduced: If a node is part of another one, an additional edge labelled by this *composite* property points from the containing node to the contained node. Figure 5.7 illustrates a signature containing two ports.

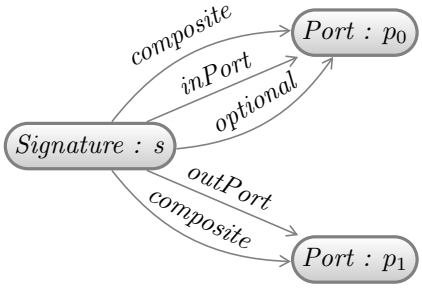


Figure 5.7.: Example: Compositional links

By this approach, the compositional link becomes an explicit part of abstract words. As for bidirectional associations, the specific characteristics of compositional links (e. g. that a node must have at least one parent) are, again, not specified at the level of abstract words. Thus, characteristics such as the lack of cycles within compositional relationship, have not yet been described.

Note that this way of modelling compositional links does not allow to bind the composition to a specific property: Imagine a signature in which some ports may be optional. A way to model this fact is to add an edge labelled by e. g. a property *optional* that points to each optional port. In Figure 5.7 the port p_0 is marked as such an optional port. In such a scenario, it can not be determined whether the property *inPort* or the property *optional* is the compositional property. In fact, a property is not compositional at all. Instead, the nodes are in a compositional relationship independent of any other property. In particular, no additional relationship is necessary at all.

5.5.3. Attributes

Some attributes, such as identifiers modelled as character strings, have already been used in the present examples. Another example relevant in practice are natural numbers. Up to now, only the shortcut notation for attributes has been introduced. Most metamodelling approaches introduce a set of primitive types in order to solve this requirement. In order to prevent the need of additional theories, it shall be shown in this section that attributes

themselves can be described by the constructs that have already been introduced without introducing any primitive type. Thus, this approach also allows a flexible treatment of the set of primitive types: In different environments different primitive types can be introduced.

Attributes with a finite domain: Enumerations

In principle, types wherein the domain of which is a finite set, can be modelled by adding a concept for each value of the type. In general, enumerations as used in metamodelling are of that kind. A relevant example is the type *Boolean*: An individual concept is added to the abstract alphabet for each logical value: $True$ and $False$. Thus, an optional port out of our example in Figure 5.7 can be modelled in a different way, such as demonstrated in Figure 5.8.

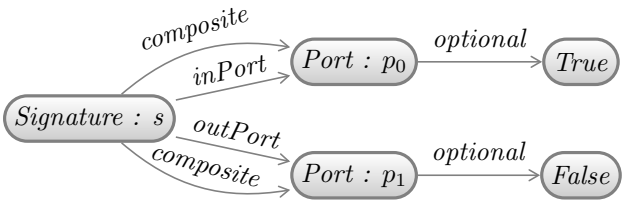


Figure 5.8.: Example: Boolean values in abstract words

Instead of having an edge from the signature node to those port nodes which are marked as optional, an additional outgoing edge labelled again by the property *optional* exists for each port, but targets to a node labelled by one of the two concepts $True$ or $False$.

In analogy to the type *Boolean* arbitrary other enumeration types can be modelled. An example may be a type $SignalLightColors$ with the elements $\{Red, Yellow, Green\}$.

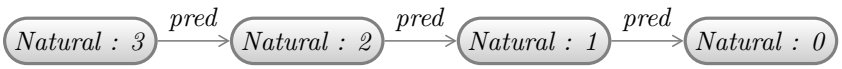
Attributes with an infinite domain

For attributes wherein the domain of which is an infinite set, such as for natural numbers or character strings, the way of modelling shown above is not sufficient. As most data structures can be modelled by natural numbers, it will first be shown of how to model this special kind of attribute. Basically it must be noted that there are arbitrary ways to model natural numbers by an M-graph. In this context, two of those shall be presented:

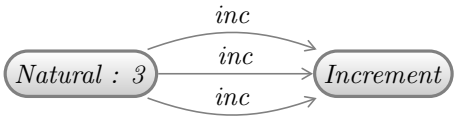
The first one follows the mathematical definition which defines natural numbers by the predecessor function. In terms of an M-graph, a node representing a natural number simply points to its predecessor. The corresponding edge is labelled by the property *pred*. Zero is marked by having no predecessor. Figure 5.9(a) models the natural number 3 by using the predecessor construct.

The second way of modelling natural numbers simply counts the increments which are modelled by multiple edges from the node representing the natural number to a node labelled by the concept *Increment*. The corresponding edge is labelled by a property *inc*. Figure 5.9(b) models the natural number 3 by using the increment construct.

Based on natural numbers, elements of arbitrary data structures can be modelled now. An important example will be the character strings: Character strings can be modelled as a list of characters. A character itself is modelled as a natural number being the Unicode for the character.



(a) Natural number modelled by predecessor



(b) Natural number modelled by increments

Figure 5.9.: Example: Two ways of modelling natural numbers

This encoding shall be shown for the character string "result" which is used in our example in Figure 5.1(b) as an identifier for a port. The abstract word representing the character string "result" is modelled as shown in Figure 5.10.

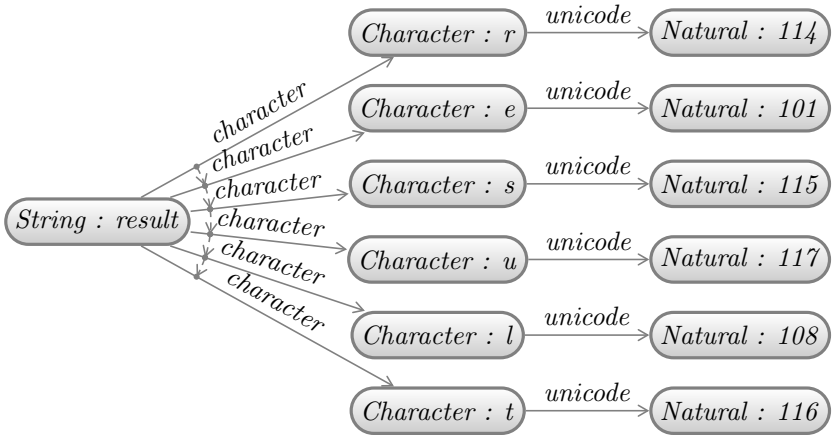


Figure 5.10.: Example: Modelling the String "result" as an abstract word

The node representing the character string is labelled by the concept *String*. It contains a list of character nodes which are labelled by the concept *Character*. The edges are labelled by *character* and are totally ordered.

Attributes as a specialisation of compositions

The composition construct has been introduced in Section 5.5.2, *Compositions*, p. 89. It will be shown that an attribute is treated as a specialisation of compositions in this approach. Thus, the *composite*-edge will have to be added in the previous diagrams in parallel to e. g. the *character*-edges or *unicode*-edges, as the characters modelled are *part of* the string. Also the natural number representing the Unicode for a character is seen as being a composition of the character. Figure 5.11 shows an M-graph of a string including the *composite*-edges.

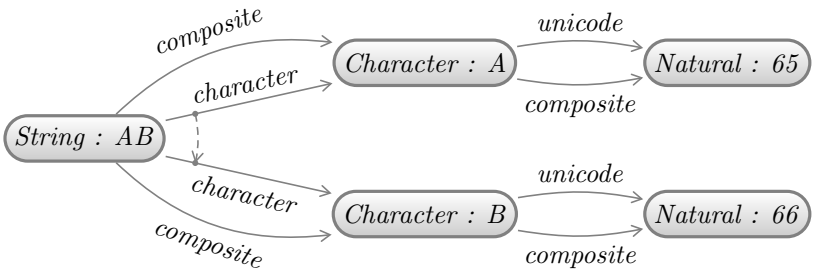


Figure 5.11.: Example: Modelling the String "ab" including the *composite*-edges

Note that up to now there has been no difference between compositions and attributes. Nevertheless, it will be shown in Chapter 7, *Abstract Syntaxes in M2L*, p. 119 that a differentiation will be necessary.

5.6. Running Example

Having introduced M-graphs for representing abstracts words in a mathematical way, a more complex abstract word shall now be shown to get a better idea of how a model is represented by M-graphs. The integrator network as shown in Figure 3.3 has been introduced in Chapter 3, *Running example: Modelling dataflow algorithms*, p. 45. For convenience, it will be illustrated here again. The M-graph representing this network is shown in Figure 5.13.

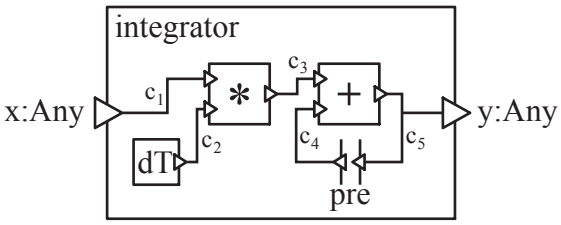


Figure 5.12.: Exemplary model: integrator network (copy of Figure 3.3)

As can be seen, the size of M-graphs increases rapidly. Larger M-graphs become more and more confusing because of the huge number of edges. Thus, even in the given example, a number of edges and nodes will be omitted for clarity:

- Except for the node representing the total integrator network n_{int} and the two ports p_1 and p_2 thereof, the explicit identifier nodes (labelled by the concept $Identifier$ and reachable via an edge labelled by the property *name*) are omitted.
- Even for the three modelled identifiers "x", "y", and "integrator" the detailed modelling of the character strings as presented in Section 5.5.3, *Attributes*, p. 89 is omitted.
- All composite edges are omitted.
- For the two lower *Channel*-nodes ch_2 and ch_4 the incoming property *channel* from n_{int} is omitted.
- For the two lower *Block*-nodes b_{dT} and b_{pre} the incoming property *subcomponent* from n_{int} is omitted.

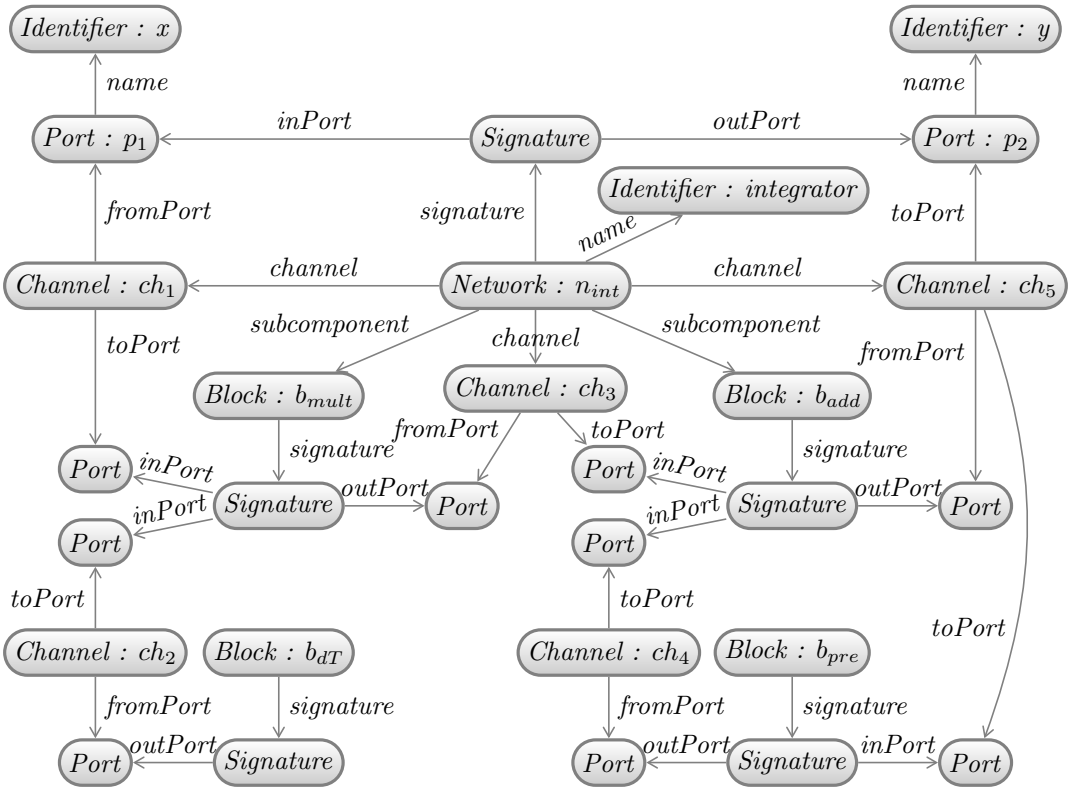


Figure 5.13.: Example: Modelling the integrator network as an abstract word

Due to the complexity of this example, a second, more trivial model will be introduced which will be used in the following chapter for demonstrating edge algebra (see Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95). This simplified example is the easiest network that can be imagined: The identity simply connecting its single input port directly to its single output port. Figure 5.14 will show such a network in diagrammatic syntax. The M-graph representing this identity network is shown in Figure 5.15.

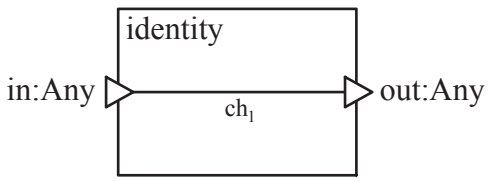


Figure 5.14.: Exemplary model: identity network (diagrammatic syntax)

In contrast to the first example, the composite edges are modelled explicitly in this second example.

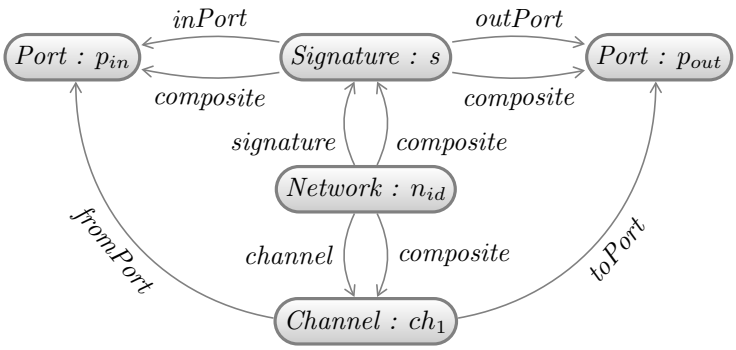


Figure 5.15.: Exemplary model: identity network as an abstract word

5.7. Defining *M2L* – Step 1: *M2L* Meta-Metamodel in terms of an Abstract Word

Arbitrary abstract words can now be defined in terms of M-graphs. This brings the reader to the first step of defining the metamodelling language *M2L*. As described in Section 2.4, *Procedure specifying the (self-describing) metamodelling language* M2L , p. 42, the difficulty of defining the metamodelling language itself is that it is defined by its own constructs. This results in a chicken-and-egg problem. Nevertheless it will now be possible to initialise the definition as illustrated in Figure 5.16.

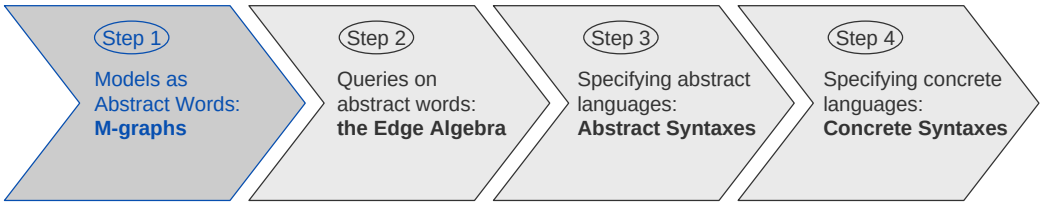


Figure 5.16.: Overview - First of the four steps of specifying M2L.

As every language, also the metamodelling language itself is described by a metamodel. In this special case, it will be called meta-metamodel. As the term already suggests, a metamodel (and thus the meta-metamodel as well) is also a (special kind of) model. Hence, the meta-metamodel can formally be written down as a M-graph. This can be done for the entire and final meta-metamodel of *M2L* as described in Chapter 9, *The overall specification of M2L*, p. 171.

The present work will skip the explicit representation of this M-graph which describes the meta-metamodel of *M2L* as it would be extremely large and thus very hard to read.

Note that although the meta-metamodel can be written down, nothing can be stated about the meaning of this model. This is the reason why the further steps in the following chapters will be required.

Chapter 6

Queries on abstract words - the Edge Algebra

In the previous section it has been shown how it is possible to describe models as abstract words in the form of M-graphs, which has solely been introduced for that purpose. In model-based engineering it is now necessary to be able to infer new properties from the basis of a given model or to check consistency conditions in many situations. In analogy to relational data bases, a query and constraint language will be essential in both cases. Whereas the well-known Relational Algebra [Codd, 1970] offers a formal basis for SQL [ISO, 2008], a corresponding suitable approach for the metamodelling domain is still missing. The Edge Algebra presented herein is supposed to bridge this gap.

The Edge Algebra will be introduced in two steps: At first, focus shall be on fundamental Edge Algebra which concentrates exclusively on navigation across edges. Secondly, the Edge Algebra will be extended in order to be able to deal with predicates which allows to describe consistency conditions.

Contents

6.1. Fundamental Edge Algebra

In our examples from Figure 5.12 and Figure 5.13 the question might come up, which of the sub-components do directly depend on a given one. The answer would come in the form of a newly, calculated edge function. The result is intuitionally easy to indicate: E. g. b_{add} depends on b_{pre} and b_{mult}, meaning that $dependsOn(b_{add}) = \{b_{pre}, b_{mult}\}$. In the graphical notation it will be possible to present the intuitive result in Figure 6.1. For clarity, all nodes except the *Block*-nodes will be omitted.

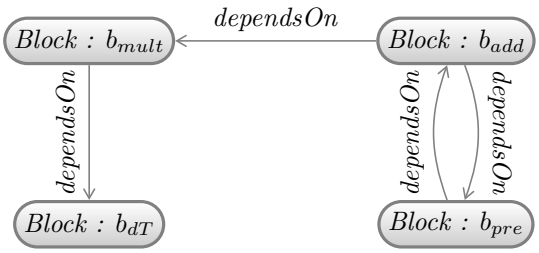


Figure 6.1.: Example: *dependsOn*-edge for integrator network (reduced M-graph)

The question might also be what the correct execution orders of the sub-components in the network stand for, thus leading to the causal order of the sub-components. It will be shown that, as it is possible to deal with partial orders, such computations considering orders and duplicates can also be expressed in an elegant way.

6.1.1. Carrier set

As already the name reveals, edge algebra forms an algebra across edges as defined in Chapter 5, *Models as Abstract Words*, p. 81 for abstract words. More precisely, the algebra is defined across edge functions $V \to \mathcal{P}_{pomset}(V)$ to a given set of vertices V. The carrier set is therefore defined as follows:

Definition 36 (Carrier set for the fundamental Edge Algebra)**.** *The carrier set of the fundamental edge algebra is the set of all edge functions* $\mathcal{E}_V$*.*

From our example in Figure 5.13 all edges labelled by one property such as *subcomponent* or $fromPort$ would correspond to exactly one element of this carrier set. Edge Algebra now defines operators to deduce new edge functions (such as *dependsOn*) from given edge functions (such as *subcomponent* or $fromPort$).

One example of a simple but important operator is the edge inverse. (Details will be described in Section 6.1.4, *The fundamental edge operators: Empty Edge, Reflexive, Equality, Inverse, Closure, and Navigation*, p. 100.) The edge inverse to the property *composite*, for example, corresponds to exactly that edge function wherein all directions are inverted. Thus the inverse edge to the property *composite* always directs towards the parent node. Figure 6.2 illustrates the resulting edge by using the example introduced in Figure 5.15.

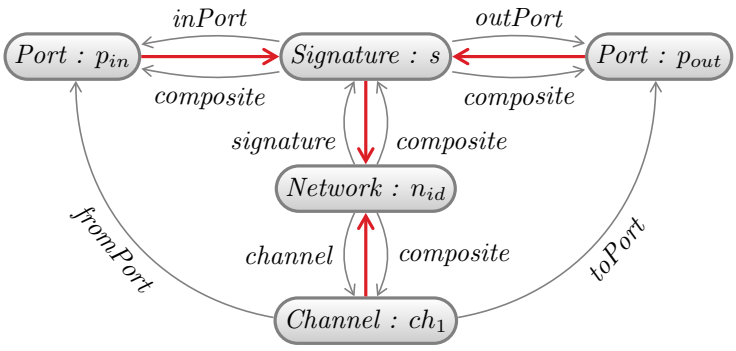


Figure 6.2.: Illustration of the inverse edge to the property *composite* (in red)

6.1.2. From abstract words to edge-functions: The Edging Operator

The alert reader will have noticed that the properties *subcomponent* or $fromPort$ do, however, correspond to one edge function respectively, but according to the definition of abstract words they are at first only an identifier for elements from the set of properties P. Moreover, individual vertices from V and concepts from C respectively cannot be used directly in the edge algebra as they do not dispose of the form of an edge function $V \to \mathcal{P}_{pomset}(V)$. In the following, concepts, properties as well as vertices are supposed to be transferred into edge functions for edge algebra. Therefore, the edging operators will be introduced:

Definition 37 (Edging Operators)**.** *For a given M-graph $\omega = \langle \Sigma, V, type, edge \rangle$ wherein $\Sigma = \langle C, P \rangle$ and the set of vertices V is disjoint to both C and P, the edging operators are defined as follows:*

$$
\begin{aligned}
P_\omega : x &=_{def} \begin{cases} edge(x) & \text{if } x \in P \\ V \to \mathcal{P}_{pomset}(V), v \mapsto \varnothing & \text{otherwise} \end{cases} \\
C_\omega : x &=_{def} \begin{cases} V \to \mathcal{P}_{pomset}(V), v \mapsto \{w \in V \mid type(w) \preceq x\} & \text{if } x \in C \\ V \to \mathcal{P}_{pomset}(V), v \mapsto \varnothing & \text{otherwise} \end{cases} \\
T_\omega : x &=_{def} \begin{cases} V \to \mathcal{P}_{pomset}(V), v \mapsto \{w \in V \mid type(w) = x\} & \text{if } x \in C \\ V \to \mathcal{P}_{pomset}(V), v \mapsto \varnothing & \text{otherwise} \end{cases} \\
V_\omega : x &=_{def} \begin{cases} V \to \mathcal{P}_{pomset}(V), v \mapsto \{x\} & \text{if } x \in V \\ V \to \mathcal{P}_{pomset}(V), v \mapsto \varnothing & \text{otherwise} \end{cases}
\end{aligned}
\tag{6.1}
$$

The edging operator thus maps the identifiers for concepts, properties and vertices in the context of a given abstract word to edge functions. It is hereby differentiated between four cases:

1. As has already been indicated, *properties* are mapped to edge functions one-to-one. If our exemplary model is given by ω, $P_\omega : subcomponent$ and $P_\omega : fromPort$ would consequently be the corresponding edge functions. Figure 6.3 illustrates the edge function $P_\omega : fromPort$ with the help of our exemplary model.

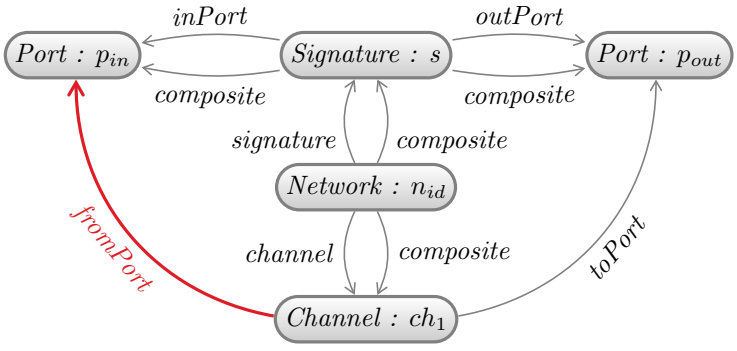


Figure 6.3.: Property edging: illustration of the edge function $P_\omega : fromPort$

2. *Concepts* are mapped to edge functions that return the set of all vertices that are labelled by the given concept for each vertex. In detail there are two different edging

operators for concepts. T_ω : returns all nodes that are exactly labelled by the given concept. In contrast C_ω :includes also those nodes that are labelled by refining concepts according to the partial order of the concepts. An example from the abstract word is $C_\omega : Port$. The resulting edge function directs from every vertex to those vertices that are labelled with the concept $Port$. Figure 6.4 illustrates the edge function $C_\omega : Port$ with the help of our exemplary model.

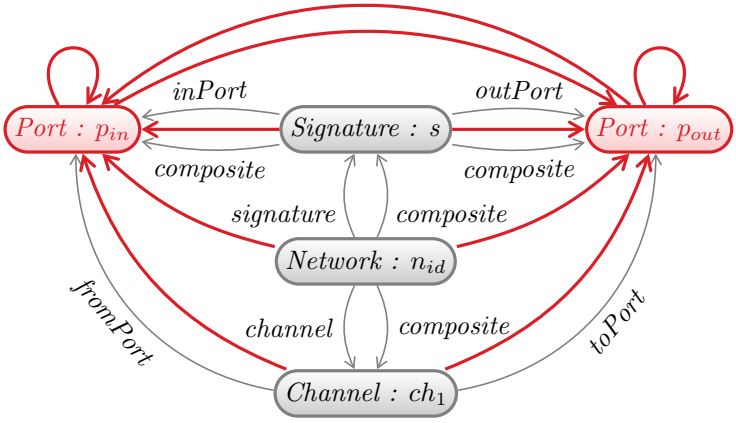


Figure 6.4.: Concept edging: illustration of the edge function $C_\omega : Port$

3. *Vertices* are mapped to constant edge functions as well. The result will be a single-valued set with the respective vertex for each vertex. Note that the set of vertices becomes relevant for the edging operator. Thus, although the concrete set of vertices is irrelevant concerning the equality of M-graphs, it must be agreed upon a concrete set of vertices when using the edging operator over a vertex. An example is $V_\omega : p_{out}$ which directs from every vertex to the vertex p_{out}. Figure 6.5 illustrates the edge function $V_\omega : p_{out}$ with the help of our exemplary model.

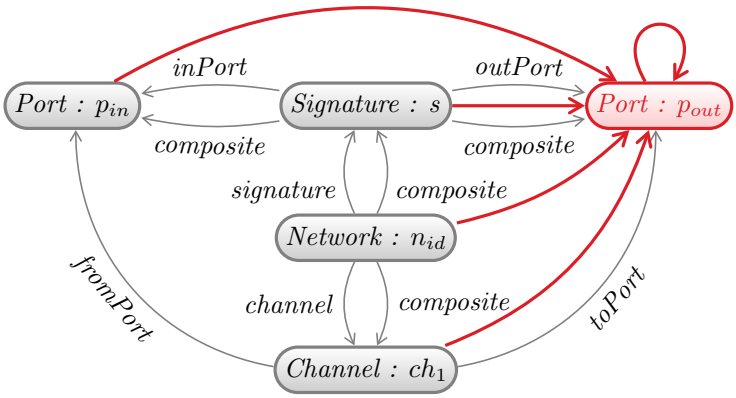


Figure 6.5.: Vertex edging: illustration of the edge function $V_\omega : p_{out}$

4. In all other cases (the identifier is not contained in the respective set C, P, or V) the resulting edge function assigns the empty pomset to every vertex. Thus, no vertices are connected via this edge function. It is thus possible to apply any identifier at the edging operator without demanding them explicitly in one of the sets C, P, and V.

If it can be seen from the context that an edge function is expected and which abstract word it refers to, the operators $C:$, $P:$, and $V:$ can be used without denoting the subscript identifier for the abstract word.

Finally, the Star Edging Operator will be introduced, which is defined as the additive union of all properties of the abstract alphabet. Here as well the subscript ω may be omitted in case it already becomes obvious from the context:

Definition 38 (Star Edging Operator)**.** *For a given M-graph $\omega = \langle \Sigma, V, type, edge \rangle$ the star edging operator is defined as follows:*

$$*_\omega =_{def} V \to \mathcal{P}_{pomset}(V), v \mapsto \biguplus_{p \in P} edge(p)(v) \tag{6.2}$$

Figure 6.6 illustrates the edge function $*_\omega$ with the help of our exemplary model.

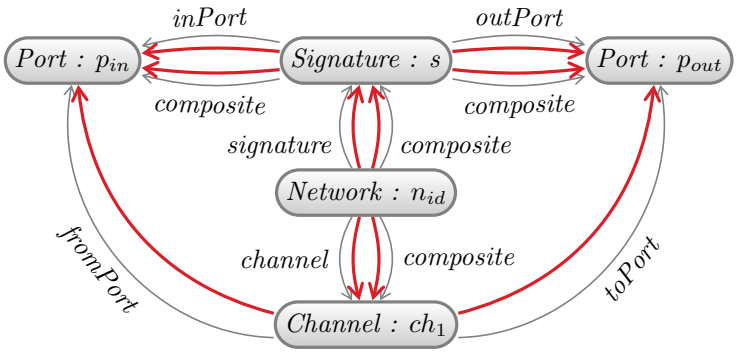


Figure 6.6.: Star edging: illustration of the edge function $*_\omega$

6.1.3. Derived pomset operators

Having finished the description of how a given abstract word is formalised as a set of edge functions, the operators of the algebra will be addressed now. Most of the operators can be canonically deduced from pomset operators. This is achieved by applying the pomset operator as defined in Section 4.4, *Operators on pomsets*, p. 66 for each vertex to the resulting pomsets of the given edge functions. Due to the carrier set of the Edge Algebra, only those pomset operators are relevant of which both the domain and co-domain are purely pomsets. Table 6.1 provides an overview of the operators. The formal definition is given by Definition 39.

Definition 39 (Derived pomset operators)**.**

$$\begin{aligned} op: \quad & {\mathcal{E}_V}^n & \to \quad & \mathcal{E}_V \\ & e_1, \ldots, e_n & \mapsto \quad & op(e_1, \ldots, e_n) =_{def} \begin{pmatrix} V \to \mathcal{P}_{pomset}(V), \\ v \mapsto op(e_1(v), \ldots, e_n(v)) \end{pmatrix} \end{aligned} \tag{6.3}$$

where

$op \in \{\uplus, \oplus, \downarrow, \downdownarrows, \cup, \cap, \setminus, \pi, \tau, 1, \gtrdot, \mu, \epsilon, \epsilon_\subset\}$

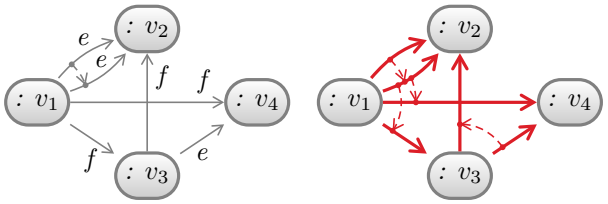
n is the arity of the operator op

operator	notation
Additive Union	$e \uplus f$
Concatenation	$e \oplus f$
Projection	$e \downarrow f$
Complement Projection	$e \downarrow^{*} f$
Union	$e \cup f$
Intersection	$e \cap f$
Difference	$e \setminus f$
Path	πe
Totalise	τe
First	$1e$
Order Inverse	$\gtrless e$
Order Destroy	μe
Duplicate Destroy	ϵe
Duplicate Destroy Over CCs	$\epsilon_{\subset} e$

Table 6.1.: Overview of edge operators derived from pomset operators

It must be taken into consideration that the elements of the carrier set $\mathcal{E}_V = V \to \mathcal{P}_{pomset}(V)$ are functions themselves. This results primarily in the somehow unfamiliar definition of operators which is to be explained in short by means of additive union: For two edge functions $e, f \in \mathcal{E}_V$ the resulting edge function given by $e \uplus f$ maps each vertex $v \in V$ to $e(v) \uplus f(v)$.

Thus, the respective pomset operator is applied to the result pomset of the edge function of each node. In case of the additive union of two edge functions, the outgoing edges of the first edge function are added to the outgoing edges of the second edge function for each node by additively uniting the two respective pomsets of target nodes. The abstract example provided in Figure 6.7 shall illustrate this procedure.



$$(e \oplus f)(v_1) = \left\{ v_2 \to v_2 \nearrow^{v_3}_{v_4} \right\}$$

$$(e \oplus f)(v_2) = \varnothing$$

$$(e \oplus f)(v_3) = \{v_4 \to v_2\}$$

$$(e \oplus f)(v_4) = \varnothing$$

Figure 6.7.: Illustration of a derived pomset operator using the example of $e \oplus f$

6.1.4. The fundamental edge operators: Empty Edge, Reflexive, Equality, Inverse, Closure, and Navigation

Besides the canonically defined edge operators fundamental Edge Algebra comprises five core operators which will be defined in the following section. Table 6.2 provides an overview of these operators.

operator	notation
Empty Edge	$\varnothing$
Reflexive	$\circlearrowleft$
Equality Edge	$\hookrightarrow$
Edge Inverse	$\leftrightarrows e$
Navigation	$e.f$
Closure	$^\wedge e$

Table 6.2.: Overview of the fundamental edge operators

For the following definitions it will be assumed that $e, f \in \mathcal{E}_V$ are two edge functions. The four operators will then be defined as follows:

Empty Edge, $\varnothing$

Definition 40 (Empty Edge, $\varnothing$)**.** *$\varnothing$ is a constant edge operator wherein the resulting edge function thereof assigns each node to an empty pomset.*

$$\begin{aligned} \varnothing \ : \quad & \to \ \mathcal{E}_V \\ & \mapsto \ \varnothing \ =_{def} \left(V \to \mathcal{P}_{pomset}\left(V\right), v \mapsto \varnothing\right) \end{aligned} \tag{6.4}$$

Reflexive, $\circlearrowleft$

The reflexive assigns each node exactly to itself. In Edge Algebra, the reflexive therefore roughly corresponds to the construct which is often called *self* or *this* in object-oriented languages.

Definition 41 (Reflexive, $\circlearrowleft$)**.** *$\circlearrowleft$ is a constant edge operator wherein the resulting edge function thereof directs to itself for each vertex.*

$$\begin{aligned} \circlearrowleft \ : \quad & \to \ \mathcal{E}_V \\ & \mapsto \ \circlearrowleft \ =_{def} \left(V \to \mathcal{P}_{pomset}\left(V\right), v \mapsto \{v\}\right) \end{aligned} \tag{6.5}$$

Figure 6.8 illustrates the edge operator $\circlearrowleft$ with the help of our exemplary model.

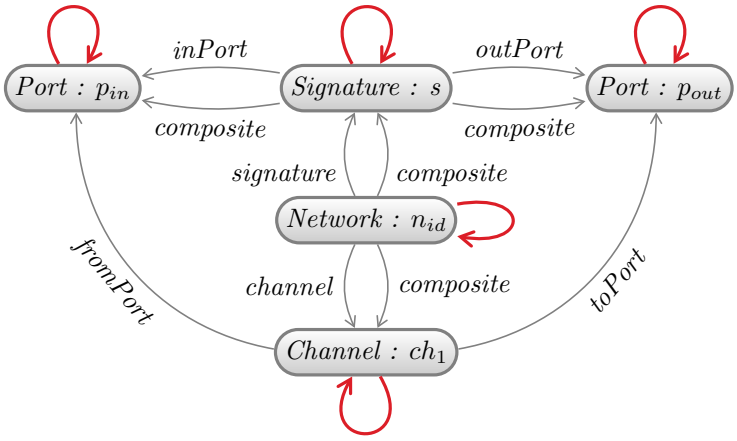


Figure 6.8.: Illustration of the reflexive edge operator $\circlearrowleft$

Please note that the Reflexive operator is a function of arity zero and is thus constant in terms of Edge Algebra. Nevertheless, this operator results to an edge function that is *not* constant at all.

Equality Edge, $\hookrightarrow$

The equality edge assigns each node to all equal nodes according the node equivalence relation. Please refer to Section 5.4, *Node equivalence*, p. 86 for a detailed definition of this relation. Note that this relation is reflexive in particular.

Definition 42 (Equality Edge, $\hookrightarrow$)**.** $\hookrightarrow$ *is a constant edge operator wherein the resulting edge function thereof directs to all equal nodes.*

$$\begin{array}{lllll} \hookrightarrow : & & \to & \mathcal{E}_V & \\ & & \mapsto & \hookrightarrow =_{def} (V \to \mathcal{P}_{pomset}(V),\, v \mapsto \{w \in V \mid w \simeq v\}) \end{array} \tag{6.6}$$

As already said for the reflexive operator, the equality edge is also a function of arity zero and is thus constant in terms of Edge Algebra. Nevertheless, this operator results to an edge function that is *not* constant at all.

Edge Inverse, $\leftrightarrows$

In many situations, a graph shall be traversed against the direction of the edges. In order to fulfil that task, the *Edge Inverse*-operator will be introduced. Many metamodelling frameworks offer no possibility to traverse against the edge direction. Thus, bidirectional associations are always necessary. Due to the *Edge Inverse*-operator, bidirectional associations will not be required in such a situation in the present approach. The formal definition is given as follows:

Definition 43 (Edge Inverse, $\leftrightarrows$)**.** *The inverse of an edge function inverts the directions of all edges. Hereby the order of pomsets is lost as only the outgoing (and not the incoming) edges of one vertex can be ordered. Duplicates, however, will remain.*

$$\begin{array}{lllll} \leftrightarrows : & \mathcal{E}_V & \to & \mathcal{E}_V & \\ & e & \mapsto & \leftrightarrows e =_{def} (V \to \mathcal{P}_{pomset}(V),\, v \mapsto \{w \in V \mid v \in_m e(w)\}) \end{array} \tag{6.7}$$

Note that a double inversion will therefore result in the initial edge function, without order, however. Figure 6.9 illustrates the *Edge Inverse*-operator with the help of our exemplary model.

Navigation, .

Intuitively, the navigation $e.f$ forms that edge function that results if a first navigation is performed along e and afterwards along f, i. e. if the edges described by e are composed of those of f. Due to this procedure, the resulting edge function will contain duplicates if multiple paths exist from one vertex to another via e followed by f. Formally, the navigation is defined by the expansion operator:

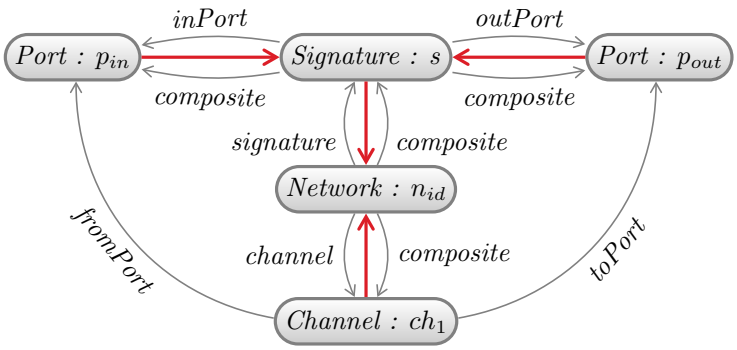


Figure 6.9.: Illustration of the *Edge Inverse*-operator $\leftrightarrows P_\omega : composite$

Definition 44 (Navigation, .)**.** *The resulting edge $e.f$ maps each node $v \in V$ to the pomset $e(v)$ wherein the elements $w \in V$ of this pomset are replaced by the pomset $f(w)$.*

$$\begin{array}{rccl} . : & \mathcal{E}_V \times \mathcal{E}_V & \to & \mathcal{E}_V \\ & e, f & \mapsto & e.f =_{def} \left(V \to \mathcal{P}_{pomset}\left(V\right), v \mapsto e(v) \,\chi\, f\right) \end{array} \tag{6.8}$$

Figure 6.10 illustrates the *Navigation*-operator with the help of our exemplary model.

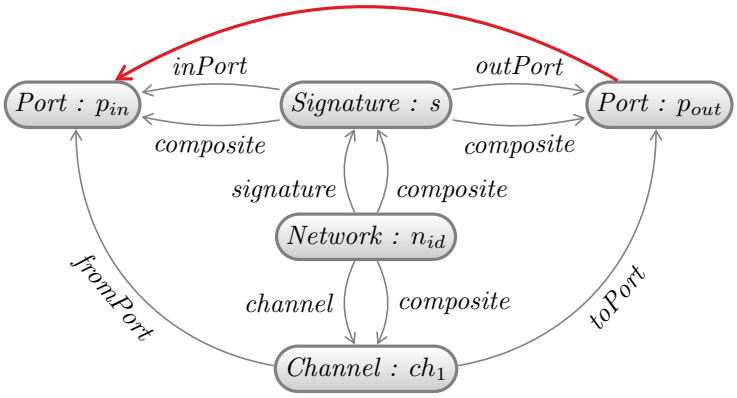
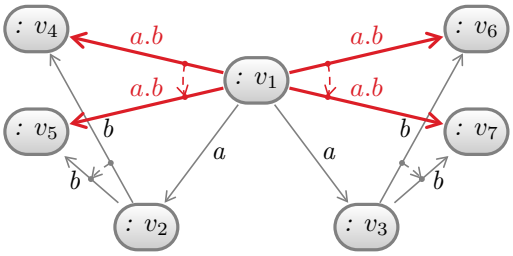


Figure 6.10.: Illustration of the *Navigation*-operator $(\leftrightarrows P_\omega : toPort).(P_\omega : fromPort)$

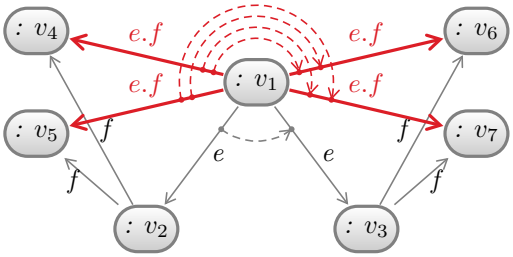
According to the definition of the navigation operator both order and duplicate information is preserved. Due to pomsets, the preserving of the order is also possible if the navigation is over two edges wherein of which is ordered and the other one is unordered. The result will be a pomset which is neither a set nor a list. This fact renders the navigation operator much more powerful than similar operators in other formalisms such as OCL [OMG, 2006b]. Figure 6.11 illustrates the *Navigation*-operator by two abstract exemplary models.

In both cases, $a.b$ and $e.f$ respectively result in an edge that maps v_1 to a real pomset of vertices:

$$(a.b)(v_1) = \left\{ \begin{array}{c} v_4 \to v_5 \\ v_6 \to v_7 \end{array} \right\} \qquad (e.f)(v_1) = \left\{ \begin{array}{c} v_4 \to v_6 \\ \diagdown\!\!\diagup \\ v_5 \to v_7 \end{array} \right\}$$



(a) $a \in (V \to \mathcal{P}_{set}(V)), b \in (V \to \mathcal{P}_{list}(V))$



(b) $e \in (V \to \mathcal{P}_{list}(V)), f \in (V \to \mathcal{P}_{set}(V))$

Figure 6.11.: Illustration of the *Navigation*-operator preserving the order

Closure, $^\wedge$

As usual definitions, the closure $^\wedge e$ for a given edge function e maps each vertex to all vertices that are transitively reachable by a vertex including itself. Additionally, the resulting set of vertices may be partially ordered: First, the order of e is preserved. Secondly, the vertices will be arranged in an ascending order via the length of the paths to reach these vertices. If the order of two vertices is contradictory or if a directed cycle exists, the two vertices are unordered. Formally, the closure is inductively defined:

Definition 45 (Closure, $^\wedge$)**.**

$$
\begin{aligned}
^\wedge : \quad & \mathcal{E}_V \quad \to \quad \mathcal{E}_V \\
& e \quad \mapsto \quad ^\wedge e =_{def} \lim_{n\to\infty} {}^\wedge_n e \\
\textit{where} & \\
& {}^\wedge_0 e =_{def} \circlearrowleft \\
& {}^\wedge_n e =_{def} \epsilon\left(\left({}^\wedge_{n-1} e\right) . \left(\circlearrowleft \oplus e\right)\right)
\end{aligned}
\tag{6.9}
$$

The duplicate-destroy operator ϵ will be required as otherwise, potential cycles would cause infinite multiplicities of nodes.

The following three examples shall illustrate how the reflexive-transitive closure works. The respective first representation shows the original edge function of e and the second one the reflexive-transitive closure $^\wedge e$. In order to be able to imagine it better with regard to the resulting pomsets, $^\wedge e$ will additionally be applied explicitly to each node.

Example 1: nodes that are reached several times (Figure 6.12)

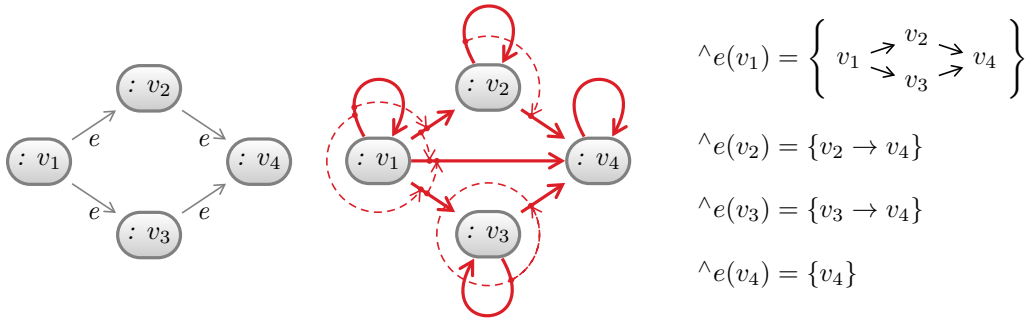


Figure 6.12.: Illustration of the *Closure*-operator $^{\wedge}e$ (example 1)

Node v_4 can be reached by node v_1 both via v_2 and via v_3. This results in the orders $\{v_1 \to v_2 \to v_4\}$ and $\{v_1 \to v_3 \to v_4\}$. As multiplicity is destroyed by the reflexive-transitive closure, v_4 will come up only once in the end.

This example also shows the similarity between the original graph and the resulting pomsets. Nonetheless, the differences should be taken into consideration: The first representation shows an edge function (resulting from a property) across all nodes. The pomset results upon insertion of a particular node into the reflexive-transitive closure of the original edge function and only reflects all reachable nodes.

Example 2: nodes conncected by multiple edges (Figure 6.13)

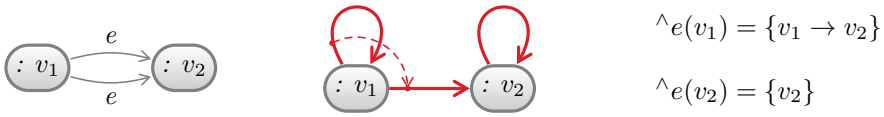


Figure 6.13.: Illustration of the *Closure*-operator $^{\wedge}e$ (example 2)

This second example shows the elimination of multiplicities due to closure formation. The two edges from vertex v_1 to vertex v_2 result in a single edge when the closure operator is applied to edge e.

Example 3: cyclical edge function (Figure 6.14)

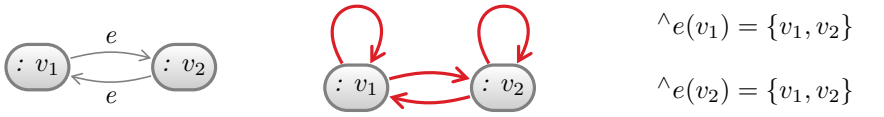


Figure 6.14.: Illustration of the *Closure*-operator $^{\wedge}e$ (example 3)

This third example shows how closure formation works in the case of cyclical edge functions. As the order $\{v_1 \rightarrow v_2\}$ as well as the order $\{v_1 \rightarrow v_2 \rightarrow v_1\}$ etc. should be contained due to the cycle, the duplicate-destroy operator ϵ will destroy the order here as well as they are contradictory.

Finally, Figure 6.15 illustrates the *Closure*-operator with the help of our exemplary model. Note that the ordering information will be omitted in this figure.

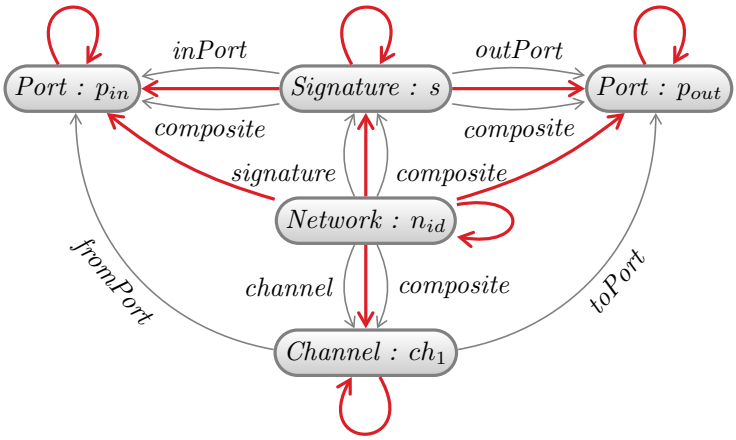


Figure 6.15.: Illustration of the *Closure*-operator $^\wedge P_\omega : composite$

6.2. Propositional Edge Algebra

As shown in the examples mentioned above, it will be possible to deduce new edge functions from a given set of edge functions and will thus be able to calculate new properties. Another important aspect is the definition of consistency conditions on abstract words. Such consistency conditions in the scope of our exemplary model from Figure 5.12 and Figure 5.13 were for example:

- A network must have a signature.
- Ports of a signature must have a name.
- All ports within a network must be connected correctly.

Edge algebra will be extended in the following to be able to formulate and evaluate such predicates.

6.2.1. Extended carrier set

The carrier set of the algebra will therefore be extended by so-called node predicates and node valuations in a first step. Propositional Edge Algebra will therefore from now on comprise the following three carrier sets in total:

$$\begin{aligned} \mathcal{E}_V &=_{def} V \to \mathcal{P}_{pomset}(V) \\ \mathcal{B}_V &=_{def} V \to \mathbb{B} \\ \mathcal{N}_V &=_{def} V \to \mathbb{N} \end{aligned} \tag{6.10}$$

Accordingly, node predicates ($\mathcal{B}_V$) assign a logical value *true* or $false$ to each vertex, whereas node valuations ($\mathcal{N}_V$) assign a natural number (including zero) to each vertex.

Node predicates and node valuations will be visualised in the present graph notation by including the assigned value in square brackets in the node marking. Figure 6.16 will provide a simple example wherein nodes will be numbered with the help of a node valuation. In addition, a node predicate will be defined in Figure 6.17, evaluating to *true* if the node valuation is even; for odd node valuations the node predicate evaluates to $false$:

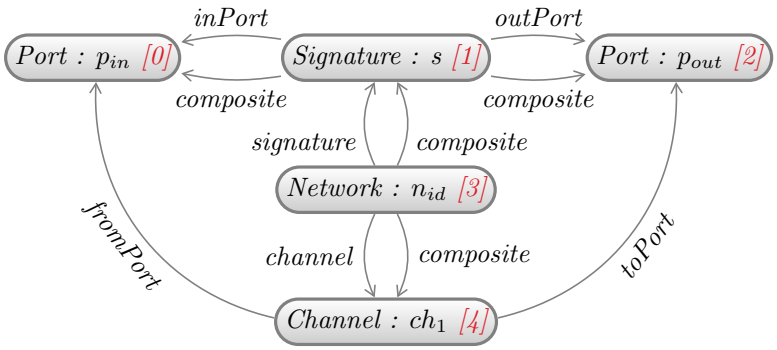


Figure 6.16.: Exemplary model: node valuation

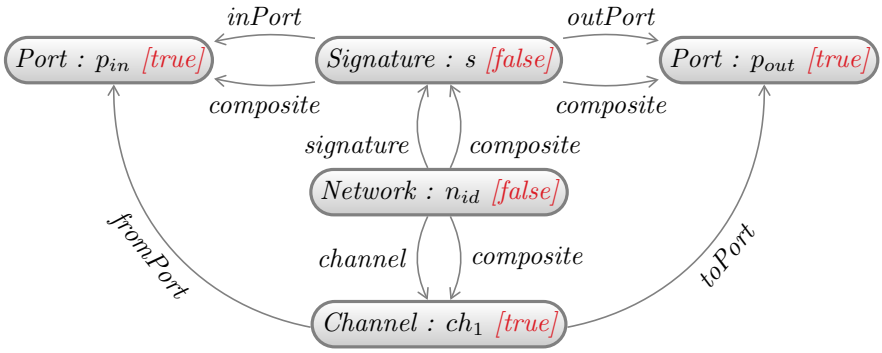


Figure 6.17.: Exemplary model: node predicate

6.2.2. Derived boolean and numeric operators

Now that the carrier set has been extended, the corresponding additional operators will be introduced in the following. Similar to pomset operators, most of the operators can be canonically derived here as well. The basic principle is – just as in the case of pomset operators – that operators are applied to the evaluations of each vertex.

Definition 46 (Derived propositional edge operators)**.**

$$\begin{array}{lrcl} op: & {\mathcal{X}_V}^n & \to & \mathcal{Y}_V \\ & x_1,\ldots,x_n & \mapsto & op(x_1,\ldots,x_n) =_{def} \begin{pmatrix} \mathcal{Y}_V, \\ v \mapsto op(x_1(v),\ldots,x_n(v)) \end{pmatrix} \end{array} \tag{6.11}$$

where

$op \in \{=, \Subset, \subseteq, |\cdot|, \|\cdot\|, [\cdot;\cdot]\cdot\} \cup \{=, \leq, +, -, \cdot, \div, \%, \min, \max\} \cup \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$

n *is the arity of the operator* op

$\mathcal{X}_V, \mathcal{Y}_V \in \{\mathcal{E}_V, \mathcal{B}_V, \mathcal{N}_V\}$ *corresponds to the domain and co-domain of* op

The set of operators is divided into three subsets which should indicate the three types of operators: pomset, numerical, and boolean operators. Note that the equality operator (=) can be applied to both pomsets and numericals. As node valuations operate on (non-negative) naturals, the subtraction operator will result in zero if the left operator is smaller than the right operator.

As an example, Definition 46 will be applied to the logical conjunction:

$$\begin{array}{lrcl} \wedge: & {\mathcal{B}_V}^2 & \to & \mathcal{B}_V \\ & p_1, p_2 & \mapsto & p_1 \wedge p_2 =_{def} \begin{pmatrix} V \to \mathbb{B}, \\ v \mapsto p_1(v) \wedge p_2(v) \end{pmatrix} \end{array}$$

6.2.3. Universally Quantified Edging

Fundamental edging has been introduced in Section 6.1.2, *From abstract words to edge-functions: The Edging Operator*, p. 97: The according edge is returned for a given concept, property or vertex. As it is now possible to talk about node properties, it is also desired to express that a node property holds for *each* property and *each* concept respectively. This is the reason why two special universal quantifiers will be introduced.

Definition 47 (Universally Quantified Edging)**.** *Let* $\omega = \langle \Sigma, V, type, edge \rangle$ *be an M-graph wherein* $\Sigma = \langle C, P \rangle$ *and the set of vertices* V *is disjoint to both* C *and* P*.*

Let $Q \subseteq P$ *be a set of excluded properties, and* $\mathrm{pred}_P : P \to \mathcal{B}_V$ *be a node predicate depending on a property, the quantified edging over properties will be defined as follows:*

$$\begin{array}{lrcl} \forall P: & \mathcal{P}_{set}(P) \times (P \to \mathcal{B}_V) & \to & \mathcal{B}_V \\ & \langle Q, \mathrm{pred}_P \rangle & \mapsto & \forall P_\omega : p \setminus Q : \mathrm{pred}_P(p) \\ & & & =_{def} \begin{pmatrix} V \to \mathbb{B}, \\ w \mapsto (\forall q \in P \setminus Q : \mathrm{pred}_P(q)(w)) \end{pmatrix} \end{array} \tag{6.12}$$

Let $D \subseteq C$ *be a set of excluded concepts, and* $\mathrm{pred}_C : C \to \mathcal{B}_V$ *be a node predicate depending on a concept, the quantified edging over concepts will be defined as follows:*

$$\begin{array}{lrcl} \forall C: & \mathcal{P}_{set}(C) \times (C \to \mathcal{B}_V) & \to & \mathcal{B}_V \\ & \langle D, \mathrm{pred}_C \rangle & \mapsto & \forall C_\omega : c \setminus D : \mathrm{pred}_C(c) \\ & & & =_{def} \begin{pmatrix} V \to \mathbb{B}, \\ w \mapsto (\forall d \in C \setminus D : \mathrm{pred}_C(d)(w)) \end{pmatrix} \end{array} \tag{6.13}$$

Figure 6.18 illustrates the *Quantified Edging*-operator with the help of our exemplary model: The node predicate holds if each property of a node only directs towards composed nodes according to the property *composite*.

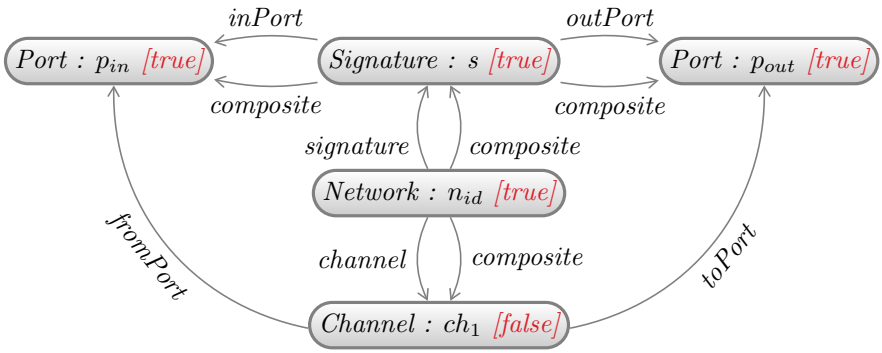


Figure 6.18.: Quantified edging $\forall P_\omega : p \setminus \{composite\} : (P_\omega : p \Subset P_\omega : composite)$

6.2.4. Core propositional operator: Selection

Up to now, only the derived operators (see Section 6.2.2, *Derived boolean and numeric operators*, p. 107) and the Quantified Edging operators (see Section 6.2.3, *Universally Quantified Edging*, p. 108) have been defined for propositional Edge Algebra. The final operator will be defined in the following, which will again be more specific concerning Edge Algebra.

The *Selection*-operator strengthens the correlation between node predicates, node valuations and the previously introduced edge functions (see Definition 29).

The basic idea of this selection operator is to *select* a set of vertices by a given predicate. An example would be to find out all root vertices of a model in terms of the *composite* property: Thus, all vertices which do not have an incoming *composite* edge should be returned. The corresponding predicate would be $(\leftrightarrows P : composite = \varnothing)$.

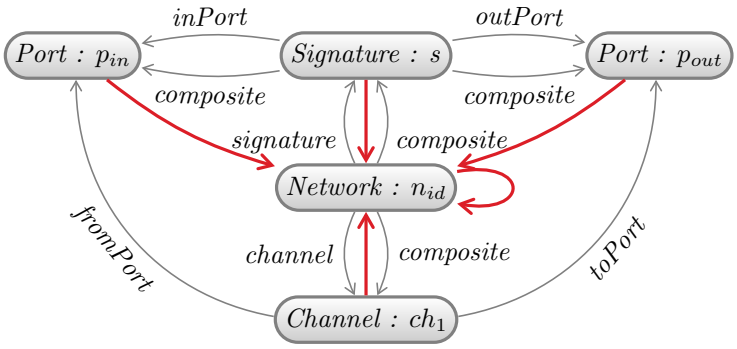


Figure 6.19.: Illustration of the *Selection*-operator $\sigma(\leftrightarrows P : composite = \varnothing)$

This would result in a set of vertices. As this operator should be part of the Edge Algebra, the result must be an edge function. Thus, in our example the resulting edge function would return a constant edge function returning the same set of vertices for which the given predicate holds for each vertex of the M-graph. The resulting edge function for our example, which shall be named *root*, should be denoted as follows:

$$root =_{def} \sigma(\leftrightarrows P{:}composite = \varnothing)$$

Figure 6.19 illustrates the given example with the help of our exemplary model. As the only vertex for which the given predicate holds is n_{id}, it is the only one that does not have an incoming *composite* edge. Thus, an outgoing edge to this single node n_{id} exists for each node. Formally, a more generalised selection operator is defined as follows:

Definition 48 (Selection (bounded and unbounded))**.** *Let $f : (V \to \mathcal{B}_V) \cup (V \to \mathcal{N}_V)$ be either a function mapping a vertex to a node predicate $\mathcal{B}_V$ or a function mapping a vertex to a node valuation $\mathcal{N}_V$.*

In case the co-domain of f consists of node predicates, $\sigma v : f(v)$ will result in a edge function mapping a vertex v to a set of those nodes for which the node predicate $\mathrm{f}(v)$ holds.

In case the co-domain of f consists of node valuations, $\sigma v : f(v)$ will result in a edge function mapping a vertex v to a bag in which the multiplicities of the vertices correspond to the corresponding node valuation.

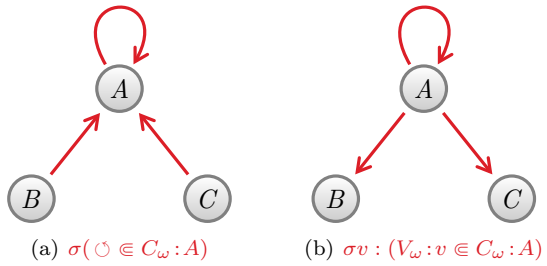
$$\begin{array}{lll} \sigma : & ((V \to \mathcal{B}_V) \cup (V \to \mathcal{N}_V)) & \to \quad \mathcal{E}_V \\ & f & \mapsto \quad \sigma v : f(v) \\ & & \quad =_{def} \begin{pmatrix} V \to \mathcal{P}_{pomset}(V), \\ w \mapsto \{u \in V \mid f(w)(u)\} \end{pmatrix} \end{array} \tag{6.14}$$

If f is constant and does therefore not depend on a given vertex v, the unbounded version of the selection operator can be used:

$$\begin{array}{lll} \sigma : & (\mathcal{B}_V \cup \mathcal{N}_V) & \to \quad \mathcal{E}_V \\ & f & \mapsto \quad \sigma f =_{def} \begin{pmatrix} V \to \mathcal{P}_{pomset}(V), \\ w \mapsto \{u \in V \mid f(u)\} \end{pmatrix} \end{array} \tag{6.15}$$

Based on the introductory example, the selection operator of the Edge Algebra, defined in Definition 48, is generalised in two ways:

- First, a selection may not only operate on node predicates but also on node valuations. In this second case, normally a bag is returned instead of a set: The multiplicity of each vertex is defined by the corresponding node valuation.
- Secondly, a selection may also depend on the vertex for which the node predicate/valuation is evaluated. In the previous example, the resulting edge function is constant such that the resulting set would be the same for each vertex. In the generalised form of the selection operator, the node predicate (and thus the resulting set as well) may differ according to the source node. Thus, the resulting edge function is generally not a constant one. This shall be denoted by a bounded vertex variable. The difference shall be illustrated by a simple example in Figure 6.20. At first Figure 6.20(a) demonstrates a unbounded usage of the selection operator: $f = (\circlearrowleft \Subset C_\omega : A)$. At second Figure 6.20(b) demonstrates a bounded usage of the selection operator: $f(v) = (V_\omega : v \Subset C_\omega : A)$.



(a) $\sigma(\circlearrowleft \in C_\omega : A)$ (b) $\sigma v : (V_\omega : v \in C_\omega : A)$

Figure 6.20.: Difference between bounded and unbounded *Selection*-operator

Finally, Figure 6.21 illustrates the selection operator σ with the help of our exemplary model. $\sigma v : (V_\omega : v \in P_\omega : composite.\hat{}P_\omega : composite)$ describes an edge function mapping a vertex n to those vertices that transitively contain vertex n in terms of the property *composite*.

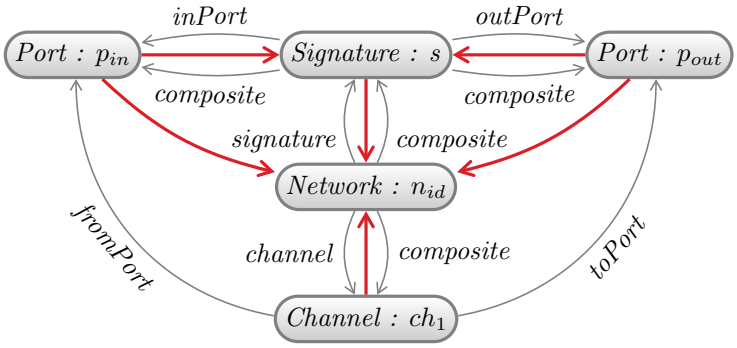


Figure 6.21.: Illustration of the selection $\sigma v : (V_\omega : v \in P_\omega : composite.\hat{}P_\omega : composite)$

6.3. Defining abstract languages using Edge Algebra

Abstract alphabets (see Definition 28) and *abstract words* have been introduced in the previous chapter based on M-graphs (see Section 5.1, *M-graphs (Model-graphs)*, p. 81). This section shall finally provide a definition of abstract languages according to the term *language* in formal languages and the use of Edge Algebra for describing such abstract languages.

Definition 49 (Abstract language)**.** *The set of all abstract words over an abstract alphabet $\Sigma = \langle C, P \rangle$ is denoted by $\Sigma^{\circledast}$. An abstract language $\mathcal{L}$ over an abstract alphabet $\langle C, P \rangle$ is defined as a possibly infinite set of valid abstract words over Σ, thus $\mathcal{L} \subseteq \Sigma^{\circledast}$.*

Note that the encircled star remembers to the Kleene star but of course it is not. It should only demonstrate the relationship to formal languages of constructing the set of all possible words based on the alphabet. As the set of valid abstract words is infinite in most cases, an explicit enumeration of all valid abstract words is impossible. Instead, abstract syntaxes are used to specify abstract languages. Generally, there are many ways of specifying an abstract language. A very fundamental one is using the Edge Algebra.

The basic idea of specifying abstract languages using the Edge Algebra is to define constraints: An abstract word belongs to an abstract language if, and only if, all constraints hold. It is possible to define node predicates over abstract words due to the propositional Edge Algebra introduced in Section 6.1, *Fundamental Edge Algebra*, p. 95 and Section 6.2, *Propositional Edge Algebra*, p. 106. A special universal quantifier over all nodes will be introduced for defining such invariants:

Definition 50 (Generality)**.** *Let $p \in \mathcal{B}_V$ be a node predicate over the set of vertices V. $\mathcal{G}^\omega p$ holds if, and only if, the given node predicate holds for all vertices within the abstract word ω.*

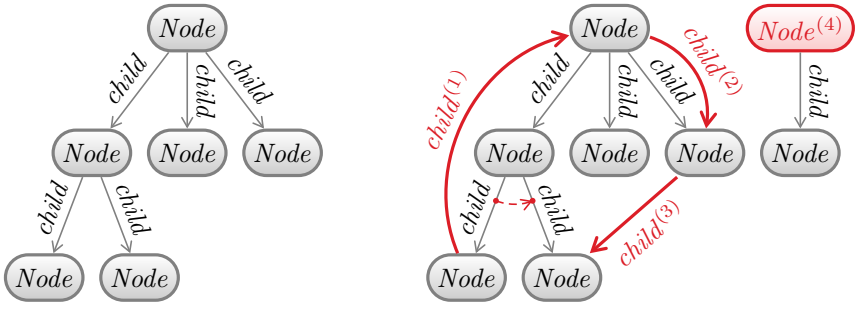
$$\begin{array}{rccl} \mathcal{G}: & \mathcal{B}_V & \to & \mathbb{B} \\ & p & \mapsto & \mathcal{G}^\omega p =_{def} (\forall v \in V : p(v)) \end{array} \tag{6.16}$$

Based on this *Generality*-operator it is now easy to formulate abstract syntaxes based on the Edge Algebra. Formally, it is defined as follows:

Definition 51 (Abstract syntax based on the Edge Algebra)**.** *An abstract syntax $\mathcal{S}$ is a tuple $\langle \Sigma, inv \rangle$ wherein Σ is an abstract alphabet and $inv \in \mathcal{B}_V$ is a node predicate. The specified abstract language will then result in $\mathcal{L}(\mathcal{S}) =_{def} \{\omega \in \Sigma^{\circledast} \mid \mathcal{G}^\omega inv\}$.*

This way of specifying an abstract language shall be illustrated by defining an exemplary abstract language of trees: Nodes are always labelled by the concept *Node*; edges are always labelled by the property *child*. Figure 6.22(a) provides an exemplary M-graph of such a tree. Figure 6.22(b) shows the tree including some additional, coloured edges and nodes which must be forbidden by the invariants. Formally, the abstract syntax $\mathcal{S}_{tree} = \langle \Sigma_{tree}, inv_{tree} \rangle$ while $\Sigma_{tree} = \langle C_{tree}, P_{tree} \rangle$ is defined by the following specification:

$$\begin{array}{rcl} C_{tree} & = & \{Node\} \\ P_{tree} & = & \{child\} \\ inv_{tree} & = & (\circlearrowleft \notin P{:}child.^\wedge P{:}child) \\ & & \wedge\, (\text{set?}\ P{:}child) \\ & & \wedge\, (|\leftrightarrows P{:}child| \leq 1) \\ & & \wedge\, (|\sigma\, (|\leftrightarrows P{:}child| = 0)| = 1) \end{array}$$



(a) Exemplary M-graph being a tree

(b) Exemplary M-graph not being a tree due to the coloured edges and nodes

Figure 6.22.: Valid and invalid abstract words for the exemplary tree language

The node predicate defining the invariant is made up of the following partial node predicates:

- $(\circlearrowleft \notin P{:}child.\hat{}P{:}child)$ ensures that there won't be any directed cycles within a tree. Thus, the invalid edge (1) will be prevented by this node predicate in Figure 6.22(b).
- (set? $P{:}child$) firstly forbids an ordering between children and secondly it prevents multiple edges to a child. Thus, the invalid edge (2) and the red coloured ordering will be prevented by this node predicate in Figure 6.22(b).
- $(|\leftrightarrows P{:}child| \leq 1)$ states that there is no node having more than one parent. Thus, the invalid edge (3) will be prevented by this node predicate in Figure 6.22(b).
- $(|\sigma\,(|\leftrightarrows P{:}child| = 0)| = 1)$ states that one root node must have exactly one tree. Thus, the invalid node (4) will be prevented by this node predicate in Figure 6.22(b).

Later on, a quite similar definition will be used to specify the property *composite* in a formal way. Please refer to Chapter 7, *Abstract Syntaxes in M2L*, p. 119 for further details. This definition will also be used in the running example (see Section 6.4, *Running Example*, p. 113).

Definition 51 allows a specification of abstract languages in a formal way *without* introducing a special metamodelling language. In order to formalise or compare various metamodelling approaches, a mapping can be defined that translates dedicated metamodels into this formalism. In the following Chapter 7, *Abstract Syntaxes in M2L*, p. 119, this approach shall be followed strictly, so as to define a formal but even mighty metamodelling language.

6.4. Running Example

As Edge Algebra has now been completely introduced, a specification of many aspects according to our running example will become possible. All in all, three aspects of metamodelling shall be illustrated. First, the specification of an abstract syntax in a formal way shall be shown by using the Edge Algebra. Secondly, it shall be emphasised that additional definitions of specific constraints are possible. Thirdly, it will be shown how to formulate the inferred edge functions *dependsOn* and *evalOrder* that have previously been introduced.

6.4.1. Abstract Syntax using the Edge Algebra

Abstract languages and the definition thereof based on the Edge Algebra has been introduced in Section 6.3, *Defining abstract languages using Edge Algebra*, p. 111. This section shall now illustrate the formulation of abstract syntax in such a formal way by using the running example as shown in Section 3.4, *A first, semi-formal abstract syntax*, p. 50.

Formally, the abstract syntax $\mathcal{S}_{net} = \langle \Sigma_{net}, inv_{net} \rangle$ while $\Sigma_{net} = \langle C_{net}, P_{net} \rangle$ is defined by the following specification:

$$C_{net} = \left\{ \begin{array}{l} DefContainer, Library, Component, \\ Signature, Port, Block, Network, Channel \end{array} \right\}$$

$$P_{net} = \left\{ \begin{array}{l} composite, name, includedLib, componentDef, includedBy, sublibrary, \\ signature, inPort, outPort, subcomponent, channel, fromPort, toPort \end{array} \right\}$$

$$
\begin{aligned}
inv_{net} \;=\; & (\\
& \quad (\circlearrowleft \notin P{:}composite.{}^{\wedge}P{:}composite)\\
& \quad \wedge (\text{set? } P{:}composite)\\
& \quad \wedge (\sigma n : ((\circlearrowleft \neq n) \Rightarrow (({}^{\wedge}P{:}composite \downarrow (n.{}^{\wedge}P{:}composite)) = \varnothing)))\\
&) \wedge\\
& (\circlearrowleft \Subset C{:}DefContainer \Rightarrow (\\
& \quad \circlearrowleft \notin T{:}DefContainer\\
& \quad \wedge P{:}name \Subset C{:}Identifier \wedge\ P{:}name \Subset P{:}composite\\
& \quad \wedge |P{:}name| = 1\\
& \quad \wedge P{:}includedLib \Subset C{:}Library\\
& \quad \wedge \text{set? } P{:}includedLib\\
& \quad \wedge \mu P{:}includedLib = \leftrightarrows P{:}includedBy\\
& \quad \wedge P{:}componentDef \Subset C{:}Component \wedge\ P{:}componentDef \Subset P{:}composite\\
& \quad \wedge \text{set? } P{:}componentDef\\
&)) \wedge\\
& (\circlearrowleft \Subset C{:}Library \Rightarrow (\\
& \quad \circlearrowleft \Subset C{:}DefContainer\\
& \quad \wedge P{:}includedBy \Subset C{:}DefContainer\\
& \quad \wedge \text{set? } P{:}includedBy\\
& \quad \wedge \mu P{:}includedBy = \leftrightarrows P{:}includedLib\\
& \quad \wedge P{:}sublibrary \Subset C{:}Library \wedge\ P{:}sublibrary \Subset P{:}composite\\
& \quad \wedge \text{set? } P{:}sublibrary\\
&)) \wedge\\
& (\circlearrowleft \Subset C{:}Component \Rightarrow (\\
& \quad \circlearrowleft \notin T{:}Component\\
& \quad \wedge\ \circlearrowleft \Subset C{:}DefContainer\\
& \quad \wedge P{:}signature \Subset C{:}Signature \wedge\ P{:}signature \Subset P{:}composite\\
& \quad \wedge |P{:}signature| = 1\\
&)) \wedge\\
& (\circlearrowleft \Subset C{:}Signature \Rightarrow (\\
& \quad P{:}inPort \Subset C{:}Port \wedge\ P{:}inPort \Subset P{:}composite\\
& \quad \wedge \text{toset? } P{:}inPort\\
& \quad \wedge P{:}outPort \Subset C{:}Port \wedge\ P{:}outPort \Subset P{:}composite\\
& \quad \wedge \text{toset? } P{:}outPort\\
&)) \wedge\\
& (\circlearrowleft \Subset C{:}Port \Rightarrow (\\
& \quad P{:}name \Subset C{:}Identifier \wedge\ P{:}name \Subset P{:}composite\\
& \quad \wedge |P{:}name| = 1\\
&)) \wedge\\
& (\circlearrowleft \Subset C{:}Block \Rightarrow (\\
& \quad \circlearrowleft \Subset C{:}Component\\
&)) \wedge
\end{aligned}
$$

$$
\begin{aligned}
&(\circlearrowleft \Subset C{:}Network \Rightarrow (\\
&\quad \circlearrowleft \Subset C{:}Component \\
&\\
&\quad \wedge P{:}subcomponent \Subset C{:}Component \wedge\ P{:}subcomponent \Subset P{:}composite \\
&\quad \wedge \text{set?}\, P{:}subcomponent \\
&\\
&\quad \wedge P{:}channel \Subset C{:}Channel \wedge\ P{:}channel \Subset P{:}composite \\
&\quad \wedge \text{set?}\, P{:}channel \\
&)) \wedge \\
&(\circlearrowleft \Subset C{:}Channel \Rightarrow (\\
&\quad P{:}name \Subset C{:}Identifier \wedge\ P{:}name \Subset P{:}composite \\
&\quad \wedge\ |P{:}name| = 1 \\
&\\
&\quad \wedge P{:}fromPort \Subset C{:}Port \\
&\quad \wedge\ |P{:}fromPort| = 1 \\
&\\
&\quad \wedge P{:}toPort \Subset C{:}Port \\
&\quad \wedge \text{set?}\, P{:}toPort \\
&\quad \wedge\ |P{:}toPort| \geq 1 \\
&))
\end{aligned}
$$

As can be seen, the node predicate inv_{net} is built up in a way that is closely related to the structure of a metamodel. After a common section which must hold for every node, there are sections for each concept that must only hold if a node is of the given corresponding concept. Due to this homogeneous structure focus shall be on the common section and the first concept-specific section (i. e. for the concept $DefContainer$).

The common section defines the overall rules for compositions. They are quite the same as has been introduced in our exemplary tree language in Section 6.3, *Defining abstract languages using Edge Algebra*, p. 111. When replacing the property *child* by the property *composite* exactly the denoted formulas except for the last part will be obtained: In contrast to the exemplary tree language for compositions a forest is allowed. Thus, multiple root nodes are possible.

The specific section is encapsulated by a node predicate such as $(\circlearrowleft \Subset C{:}DefContainer \Rightarrow (\ldots))$ for the concept $DefContainer$. It ensures that the included node predicates are only relevant if, and only if, the dedicated node is of the corresponding concept such as $DefContainer$.

$(\circlearrowleft \notin\ T : DefContainer)$ firstly ensures that there is no instance of the concept $DefContainer$ as it should be abstract. Afterwards the contained properties are specified: $(P : name \Subset C : Identifier)$ states that the property *name* directs towards a node of the concept $Identifier$. $(P : name \Subset P : composite)$ states that the property *name* is a composition. $(|P{:}name| = 1)$ states that the multiplicity of the property *name* is [1..1]. The definition of the other properties will follow afterwards. As they are quite similar to the first one, these properties will not be explained explicitly.

6.4.2. Additional Invariants

Up to now, only the metamodel as defined by the MOF diagram in Figure 3.2 has been formalised. Nevertheless, it is now possible to define a set of additional consistency conditions due to the expressive power of the Edge Algebra.

Such a consistency condition can be defined for connecting ports based on our running example: Up to now it has been possible to connect arbitrary ports via a channel although it has not been reasonable in many cases. According to our language, a channel always belongs to a network. First of all, such a channel should only connect ports located within this network. Other ports outside this network must not be connected. Secondly, the source port for a channel must be either an input port of the total network or an output port of one of the sub-components. This rule can be applied for a destination port for a channel the other way round: It must be either an output port of the total network or an input port of one of the sub-components. Formally, it is defined as follows:

$$
\begin{array}{l}
(\circlearrowleft \Subset C{:}Channel \Rightarrow (\\
\quad P{:}fromPort \Subset \leftrightarrows P{:}channel.(P{:}signature.P{:}inPort \\
\qquad \uplus P{:}subcomponent.P{:}signature.P{:}outPort) \\
\\
\quad \wedge P{:}toPort \Subset \leftrightarrows P{:}channel.(P{:}signature.P{:}outPort \\
\qquad \uplus P{:}subcomponent.P{:}signature.P{:}inPort) \\
))
\end{array}
$$

These additional node predicates can be seamlessly integrated into the existing definition of the abstract language:

$$
\begin{array}{l}
(\circlearrowleft \Subset C{:}Channel \Rightarrow (\\
\quad P{:}name \Subset C{:}Identifier \wedge\ P{:}name \Subset P{:}composite \\
\quad \wedge\ |P{:}name| = 1 \\
\\
\quad \wedge P{:}fromPort \Subset C{:}Port \\
\quad \wedge P{:}fromPort \Subset \leftrightarrows P{:}channel.(P{:}signature.P{:}inPort \\
\qquad \uplus P{:}subcomponent.P{:}signature.P{:}outPort) \\
\quad \wedge\ |P{:}fromPort| = 1 \\
\\
\quad \wedge P{:}toPort \Subset C{:}Port \\
\quad \wedge P{:}toPort \Subset \leftrightarrows P{:}channel.(P{:}signature.P{:}outPort \\
\qquad \uplus P{:}subcomponent.P{:}signature.P{:}inPort) \\
\quad \wedge\ \text{set?}\ P{:}toPort \\
\quad \wedge\ |P{:}toPort| \geq 1 \\
))
\end{array}
$$

6.4.3. Inferred edge functions *dependsOn* and *evalOrder*

Finally, the definition of the previously introduced edge functions shall be illustrated. The edge function *dependsOn*, which was used as an introducing example in Section 6.1, *Fundamental Edge Algebra*, p. 95, can be specified as follows:

$$
\begin{array}{lll}
dependsOn & =_{def} & P{:}signature.P{:}inPort.\ \leftrightarrows P{:}toPort. \\
& & \qquad P{:}fromPort.\ \leftrightarrows P{:}outPort.\ \leftrightarrows P{:}signature
\end{array}
$$

It is simply specified by a navigation over signatures, ports, and channels. Some of them are navigated in their inverse direction.

The edge function *evalOrder* has been introduced in Section 4.5, *Running Example*, p. 80 of Chapter 4, *Pomsets in the context of metamodelling*, p. 61. In that section, the evaluation

order for all sub-components of a dataflow network has already been motivated. The difficulty lies in the cycles that must be broken up at the position of the *pre*-block. A formal definition can be as follows:

$$\begin{aligned} & & evalOrder &=_{def} \epsilon(P{:}\,subcomponent.\ \gtrless^{\wedge}(dependsOn \downarrow^{*} \leftrightarrows pre)) \\ \text{where} & & & \\ & & pre &=_{def} \sigma(\circlearrowleft \Subset C{:}\,Block \wedge P{:}\,name = "pre") \end{aligned}$$

6.5. Defining M2L – Step 2: M2L defined by Edge Algebra statements

As shown in Section 6.4.1, *Abstract Syntax using the Edge Algebra*, p. 113, it is now possible to define abstract languages by Edge Algebra statements. This leads to the second step of defining the metamodelling language *M2L* as shown in Figure 6.23.

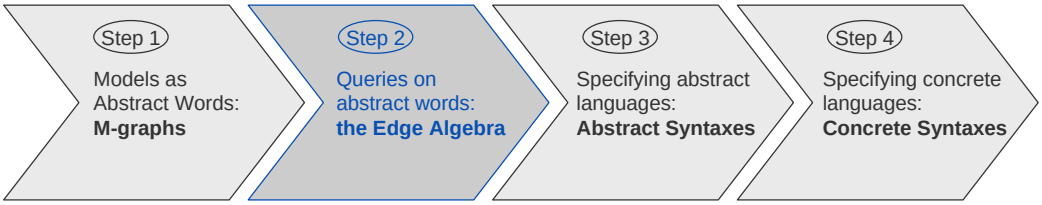


Figure 6.23.: Overview - Second of the four steps of specifying M2L.

Particularly due to that, it is therefore possible to define the abstract language for the total metamodelling language *M2L* as defined in Chapter 9, *The overall specification of M2L*, p. 171 in terms of Edge Algebra statements. Please keep in mind that from a formal point of view still nothing is stated concerning the meaning of the metamodelling language.

Nevertheless is it possible to establish a correlation between the abstract language that can be defined now and the abstract word that has been introduced in Section 5.7, *Defining* M2L *– Step 1:* M2L *Meta-Metamodel in terms of an Abstract Word*, p. 94: The abstract word is valid for the abstract language of *M2L* as the meta-metamodel is a metamodel in particular.

The second relationship that shall be established is still open: Up to now, the abstract language of *M2L* cannot be derived from the abstract word describing *M2L*. This gap will be closed in the following chapter by introducing the semantics of *M2L*.

Chapter 7

Abstract Syntaxes in M2L

In Section 6.4.1, *Abstract Syntax using the Edge Algebra*, p. 113 it has been shown how to specify an abstract syntax by using the Edge Algebra. Moreover it has been shown that the Edge Algebra allows a way of modelling such that the structure of a common metamodel is preserved. Thus all node predicates defining a specific concept can be put together on one position in the formula. Even the predicates for one property are written down in an explicit way.

Nevertheless, using Edge Algebra for defining an abstract syntax results in a complex mathematical definition. In addition, useful concepts such as compositions must always be modelled in an explicit way. This is the reason why an appropriate metamodelling language for defining abstract syntaxes in an easy way according to the requirements defined in Section 2.3, *Requirements to a metamodelling language*, p. 34 shall be defined within this chapter.

Contents

7.1. Relationship between model and metamodel

The aim of metamodelling is the formulation of a definition of an abstract language also by way of an (independent) model – the so-called metamodel. Due to this, a second model $\omega 2 = \langle \Sigma_{\omega 2}, V_{\omega 2}, type_{\omega 2}, edge_{\omega 2} \rangle$, called metamodel, shall be given in addition to the model $\omega =$

$\langle \Sigma_\omega, V_\omega, type_\omega, edge_\omega \rangle$. Now it must be decided for an arbitrary pair of model and metamodel whether the model ω *conforms to* the metamodel $\omega 2$. The corresponding *conformsTo*-Relation on model is expressed as follows:

$$\omega \lhd \omega 2 \tag{7.1}$$

The specification of the *conformsTo*-Relation exactly describes the (structural) semantics of the metamodelling language. It is obvious, however, that the freedom of defining such a metamodelling language is vast. In the practice of metamodelling, some concepts have turned out to be reasonable and vital, however. These concepts shall be introduced in Section 7.4, *Semantics for Abstract Syntaxes – Part 1: Basic metamodelling concepts*, p. 125 in a formal way and shall be enriched by additional and helpful constructs in Section 7.5, *Semantics for Abstract Syntaxes – Part 2: Extended metamodelling concepts*, p. 133.

The abstract language of $\mathcal{L}(\omega 2)$, which is described by the metamodel $\omega 2$, can easily be defined on the basis of the *conformsTo*-Relation:

$$\mathcal{L}(\omega 2) =_{def} \left\{ \omega \in \Sigma^{\circledast} \mid \omega \lhd \omega 2 \right\} \tag{7.2}$$

Thus, the conformity condition can also be expressed as $\omega \in \mathcal{L}(\omega 2)$, as the following is valid:

$$\omega \in \mathcal{L}(\omega 2) \Leftrightarrow \omega \lhd \omega 2 \tag{7.3}$$

It must be noted, however, that the metamodel $\omega 2$ may first be any arbitrary model. In the course of formalisation, the metamodel $\omega 2$ itself will, however, also be restricted regarding its structure. These structural restrictions for metamodels result in the so-called meta-metamodel, which is denoted by $\omega 3$. The meta-metamodel thus represents a perfect metamodel describing the structure of the metamodels suitable for a selected *conformsTo*-Relation. It shall therefore be written as follows:

$$\forall \omega 2 \in \mathcal{L}(\omega 3) : \omega 2 \lhd \omega 3 \tag{7.4}$$

As the meta-metamodel only represents a particular metamodel the following reflexivity will be valid in addition, thus characterising the meta-metamodel $\omega 3$ as such:

$$\omega 3 \lhd \omega 3 \tag{7.5}$$

Each *conformsTo*-Relation should therefore come along with a meta-metamodel. Here again it shall be emphasised that different *conformsTo*-Relations will also result in different meta-metamodels. We denote the set including the one meta-metamodel by $\mathcal{M}3$, the set of all metamodels by $\mathcal{M}2$, and the set of all models by $\mathcal{M}1$. All in all, the following hierarchy of model levels results:

$$\begin{aligned} \mathcal{M}3 &=_{def} \{\omega 3\} \\ \mathcal{M}2 &=_{def} \mathcal{L}(\omega 3) \\ \mathcal{M}1 &=_{def} \textstyle\bigcup_{\omega 2 \in \mathcal{M}2} \mathcal{L}(\omega 2) \end{aligned} \tag{7.6}$$

where

$$\mathcal{M}3 \subset \mathcal{M}2 \subset \mathcal{M}1$$

Before the *conformsTo*-relation can be defined, the link between model and metamodel must be considered: On the one hand, a set of concepts C_ω and a set of properties P_ω exists within

the model ω. On the other hand, a set of validity conditions for these concepts and properties shall be defined by way of the metamodel. Hence, it must be referred to these concepts and properties within the metamodel. Therefore, the two core concepts for metamodels will be introduced beforehand:

Definition 52 (Core metamodel concepts)**.** *Within a metamodel $\omega 2$, the concept Concept represents all concepts which should be restricted by said metamodel; the concept Property represents all properties.*

In the metamodel $\omega 2$ there are thus vertices $v \in V_{\omega 2}$ that are marked by the concepts *Concept* and *Property* respectively. A metamodel may generally contain more than one vertex representing one and the same concept or property. It will be seen that in the meta-metamodel of *M2L* a property is defined as a special identifier and thus represents a set of characters. (Note that in order to support canonical names, a concept will be defined as a structured identifier in terms of a list of identifiers.) Vertices which are equal in terms of node equivalence represent the same concept and property respectively.

Formally, a *coverage function* is defined such that each equivalence class of property and concept vertices respectively must be mapped to a different element out of $C_\omega \cup P_\omega$. Based on this, the link between the model ω and the metamodel $\omega 2$ is established by having a coverage function that maps each element of C_ω and P_ω to one equivalence class of vertex elements out of $V_{\omega 2}$. Formally, it is defined as follows:

Definition 53 (Coverage of concepts and properties)**.** *The coverage function cov maps each element of C_ω and P_ω to an equivalence class of vertices out of $V_{\omega 2}$ which are correspondingly labelled by either the concept* Concept *or the concept* Property*. An element of $C_\omega \cup P_\omega$ can also be mapped to an empty set in order to indicate that there is no counterpart in the metamodel and thus there are no restrictions regarding this concept or property.*

For none of two unequal elements $x \neq y$ the coverage function will result in the same equivalence class of vertices except the empty set.

$$
\begin{aligned}
&\begin{array}{rccl} cov: & C_\omega \cup P_\omega & \rightarrow & \mathcal{P}_{set}\left(V_{\omega 2}\right) \\ & x & \mapsto & cov(x) \end{array} \\
&\textit{where} \\
&\forall c \in C_\omega : cov(c) \in \left(\varnothing \ \cup \bigcup_{v \in V_{\omega 2}} (\hookrightarrow\downarrow C_{\omega 2} : Concept)(v) \right) \\
&\forall p \in P_\omega : cov(p) \in \left(\varnothing \ \cup \bigcup_{v \in V_{\omega 2}} (\hookrightarrow\downarrow C_{\omega 2} : Property)(v) \right) \\
&\forall x, y \in C_\omega \cup P_\omega : \Big((x \neq y) \Rightarrow (cov(x) \neq cov(y) \vee cov(x) = cov(y) = \varnothing) \Big)
\end{aligned}
\tag{7.7}
$$

Please bear in mind that elements out of C and P of an abstract alphabet are just mathematical symbols although they are named by sensitive phrases such as *Component* or *Port*. This approach has been followed when the concepts *Concept* and *Property* for metamodels were introduced. As mentioned above, both concepts and properties are uniquely defined by character sequences within a metamodel. At this point, concepts and properties are mapped to a concrete character sequence for identification. The mathematical symbols will normally be named by the same character sequence as defined within the metamodel for convenience.

By means of the following formalisation, it shall particularly be shown how the structural semantics of a metamodelling language can be defined with the help of Edge Algebra.

7.2. Semi-formal introduction of the abstract syntax

As described in Section 6.5, *Defining M2L – Step 2: M2L defined by Edge Algebra statements*, p. 117, it is already possible to define the abstract syntax of the metamodelling language *M2L* in a formal way. In order to provide a better understanding of the following sections, an overview of the relevant excerpt of the meta-metamodel will be given in advance. Abstract syntax will therefore be shown by using commonly known UML class diagrams (in particular the MOF subset is sufficient) again, as has been defined in [OMG, 2006a]. Figure 7.1 shows an excerpt of the M2L's abstract syntax which concentrates on specifying abstract syntaxes in M2L.

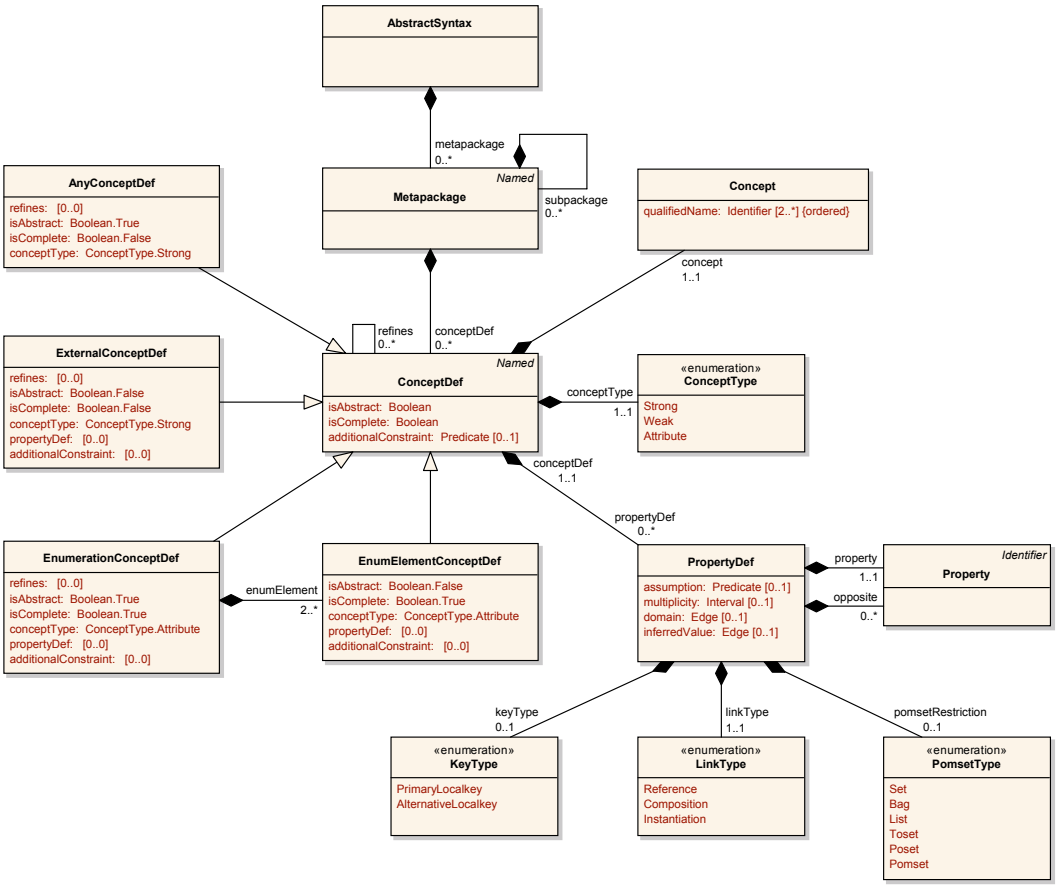


Figure 7.1.: Semi-formal abstract syntax for specifying abstract syntaxes in M2L

Note that here, no difference will be made between attributes and compositions. In addition, attributes without a type but having a multiplicity [0..0] can be found in the diagram. One example is the property $refines$ in the concept $EnumerationConceptDef$. It means that the property $refines$, which is already defined in the concept $ConceptDef$, is now stronger restricted such that the multiplicity is [0..0]. Thus in the concept $EnumerationConceptDef$, the property $refines$ must not occur.

Although a detailed description will be provided in the following three sections, a short overview shall be given beforehand: The top-level concept is *AbstractSyntax* and consists of a set of packages named $Metapackage$. Meta-packages are hierarchically structured and may thus contain sub-packages. Within a meta-package the definitions of the included concepts are enumerated ($ConceptDef$).

First of all, a $ConceptDef$ directs towards a $Concept$, which is simply defined by a qualified name. As mentioned above, the link to the model is defined by the concept $Concept$. This qualified name must be consistent to the package hierarchy and name of the $ConceptDef$. A $ConceptDef$ may refine a set of other concept definitions (which is similar to a specialisation relationship). Due to the fact that the property $refines$ is set-valued, our metamodelling approach supports multi-inheritance. Further, it shall be distinguished between *strong*, *weak*, and *attribute* concepts and a concept can be declared as *abstract* and *complete* (wherein details will be described later on). Finally, a concept definition contains a set of property definitions ($PropertyDef$).

According to the diagram, there are four special kinds of concept definitions which shall shortly be introduced:

- **AnyConceptDef** defines a concept which is refined implicitly by every other concept,
- **ExternalConceptDef** is a stub for concepts that are defined externally and thus in another metamodel,
- **EnumerationConceptDef** allows the definition of enumerations, and
- **EnumElementConceptDef** is a special concept for the elements defined for an enumeration.

A formal definition of the meaning of each of those concepts will be described in detail in the following two sections. In addition, a detailed description of each concept will be given in Section 9.3, *Package* ORG.Metamodels.M2L.AbstractSyntax, p. 188.

The concepts *Edge* and *Predicate* occur particularly within concept and property definitions. These concepts represent a one-by-one statement out of the Edge Algebra: The concept *Edge* represents a statement for an edge function (for context-sensitive domains and inferred values); the concept *Predicate* represents a statement for a node predicate (for additional constraints and assumptions). A detailed definition of these concepts will be given in Section 9.6, *Package* ORG.Metamodels.EdgeAlgebra, p. 233.

As these concepts exhibit exactly the same structure as the mathematical formulas, a formal definition will be skipped. Nevertheless, an operator has to be introduced that converts an abstract word representing an Edge Algebra statement to an edge function, node predicate or node valuation:

Definition 54 (Semantical meaning of Edge Algebra nodes)**.** $[[v]]$ *returns the corresponding edge function* $\mathcal{E}_{V_{\mathcal{M}1}}$*, node predicate* $\mathcal{B}_{V_{\mathcal{M}1}}$*, and node valuation* $\mathcal{N}_{V_{\mathcal{M}1}}$ *respectively to a given node* $v \in V_{\mathcal{M}2}$*. Hereby the vertex* v *represents the root vertex for the edge algebra statement.*

$$\begin{array}{rccl} [[\,\cdot\,]]: & V_{\mathcal{M}2} & \to & \mathcal{E}_{V_{\mathcal{M}1}} \cup \mathcal{B}_{V_{\mathcal{M}1}} \cup \mathcal{N}_{V_{\mathcal{M}1}} \\ & v & \mapsto & [[v]] \end{array} \tag{7.8}$$

7.3. Basic approach defining semantics for Abstract Syntaxes

Before the *conformsTo*-relation will be defined, fitting to the meta-metamodel that has already been introduced, the principle structure of the semantical rules shall be discussed. Basically all conditions will have the form that conditions of the metamodel – denoted as A – imply conditions of the model – denoted as B.

Such conditions will always have the form such that they are universally quantified over all concepts and all properties. As there are multiple representations in a metamodel for a single concept or property, the coverage function as defined in Definition 53, will return a set of relevant vertices out of the metamodel. Hence, extensional quantifiers are additionally necessary in order to demand a single corresponding vertex that fulfils the condition A.

There will be situations for which the condition B will have to be parametrised. This will again be done by a universal quantification over natural numbers as well as vertices out of the metamodel. Natural numbers are e. g. necessary for multiplicities in order to define the lower and upper limit respectively. Vertices are needed to parameterise the condition by a complete edge algebra statement.

Definition 55 (Metamodel-to-model restriction)**.** *Let $A(v_c, v_d, v_p, v_q, v, n)$ be an node predicate over the metamodel $\omega 2$, and $B(c, d, p, q, v, n)$ be an node predicate over the model ω. $v_c, v_d, v_p, v_q, v \in V_{\omega 2}$ are nodes of the metamodel therein, $c, d \in C_\omega$ are concepts of the model, $p, q \in P_\omega$ are properties of the model, and $n \in \mathcal{N}$ is a natural number. Then it will be defined:*

$$
\begin{aligned}
(A \rhd\!\rhd B) \quad \Leftrightarrow \quad & \forall c, d \in C_\omega : \forall p, q \in P_\omega : \forall v \in V_{\omega 2} : \forall n \in \mathbb{N} : (\\
& \quad (\exists v_c \in cov(c), v_d \in cov(d) : \exists v_p \in cov(p), v_q \in cov(q) : \\
& \quad\quad \mathcal{G}^{\omega 2} A(v_c, v_d, v_p, v_q, v, n) \\
& \quad) \\
& \quad \Rightarrow \mathcal{G}^{\omega} B(c, d, p, q, v, n) \\
&)
\end{aligned}
\tag{7.9}
$$

Note that not every bounded variable will be necessarily used in the following definitions. Nevertheless, a variable has to be used in either both node predicates A and B or none of them for a sensitive use.

A second restriction will be introduced in addition that will especially be used when the restriction concerns a special property: $A \blacktriangleright\!\blacktriangleright B$ will be denoted. Initially, this restriction is read as being the same as $A \rhd\!\rhd B$. It shall be redefined later on in order to formalise an extended metamodelling concept. Please refer to Section 7.5.2, *Conditional properties*, p. 134 for further details.

Two additional functions will be defined besides this definition in order to simplify the following definitions. As our restrictions are always bound to either a dedicated concept or a dedicated property of a concept, the corresponding definitions will always have to be figured out. These are the vertices of the metamodel which are labelled by the concepts *ConceptDef* and *PropertyDef*. Formally, it is defined as follows:

Definition 56 (*cDef* and *pDef*)**.** *For a given vertex v_c, labelled by the concept* Concept*,* $\mathrm{cDef}(v_c)$ *returns the corresponding* ConceptDef *node.*

For a given vertex v_c, labelled by the concept Concept*, and a given vertex v_p, labelled by the concept* Property*,* $\mathrm{pDef}(v_c, v_p)$ *returns the corresponding* PropertyDef *node.*

$$\begin{aligned} \text{cDef}(v_c) &=_{def} \quad V_{\omega 2} : v_c.\ \leftrightarrows P_{\omega 2} : concept \\ \text{pDef}(v_c, v_p) &=_{def} \quad \text{cDef}(v_c).P_{\omega 2} : propertyDef) \downarrow (V_{\omega 2} : v_p.\ \leftrightarrows P_{\omega 2} : property) \end{aligned} \tag{7.10}$$

Thus, a model ω conforms to $\omega 2$ – expressed as $\omega \lhd \omega 2$ – if all conditions from the following two sections are fulfilled. All in all, it is distinguished between basic metamodelling concepts and extended metamodelling concepts. In principle there is no difference between these two types except that the former ones are more commonly known as they are somehow available in most metamodelling approaches (even though they are not formalised).

7.4. Semantics for Abstract Syntaxes – Part 1: Basic metamodelling concepts

As has already been mentioned above, the first part of the *conformsTo*-relation will be defined in this section. In contrast to the second part, it contains the basic concepts which are commonly known in the metamodelling domain. The metamodelling concepts are listed in detail in Table 7.1.

basic metamodelling concept
Concept and property refinement
Abstract concepts
Complete concepts
Weak concepts
Attribute concepts
Enumeration concepts
External concepts
Global constraints: the concept *Any*
Compositional properties
Inverse properties
Multiplicities
Pomset-type restrictions

Table 7.1.: Overview of the basic metamodelling concepts

7.4.1. Concept and property refinement

At the very beginning, the refinement of concepts shall be introduced. It is closely related to the idea of specialisation and generalisation. Due to the fact that this approach is entirely based on constraining M-graphs, a refinement can be defined very easily.

Basically, a set of constraints is defined for a dedicated concept. Vertices labelled by such a concept are valid in terms of the defined abstract syntax if all constraints hold. Additionally, a concept may refine one or even more other concepts. Once a concept refines others, the constraints of both the given concept and all refined concepts must hold. As a refined concept may refine other concepts again, all constraints which are transitively reachable must hold for a concept.

Due to this definition, a cyclic refinement does not cause problems. The result will be that all concepts on the cycle have to fulfil exactly the same constraints. Such a cyclic definition is, however, forbidden by abstract syntax as it contradicts the common understanding of refinement and allowing cycles is not seen as being an advantage.

This constraint-based approach for refining concepts allows a straight-forward solution for property refinements: If a property should be refined, it is simply defined in the refining concept with its stronger constraints once more. As both constraint definitions of the property must hold, the property is refined. Note that this approach also supersedes a redundant definition within the refining property. If the type is therefore not stronger restricted but only multiplicity, the type can be skipped in the refining property definition. The same mechanism is used when multiple concepts are refined and if more than one of them defines identical properties. Then again, the property constraints of all refined concepts must hold.

Note that this very generic and powerful approach may also lead to contradictory definitions. From a formal point of view this is not an issue as it simply forbids any models containing nodes marked by such concepts.

Formally, the definition of refinement requests for each node that if a vertex is of a concept c refining a concept d, this vertex must also be of the concept d.

$$\begin{aligned} &\mathrm{cDef}(v_d) \Subset (\mathrm{cDef}(v_c).P_{\omega 2}\!:\!refines) \\ &\quad \rhd\!\rhd \\ &(\circlearrowleft \Subset C_\omega\!:\!c) \Rightarrow (\circlearrowleft \Subset C_\omega\!:\!d) \end{aligned} \tag{7.11}$$

This approach has one major difference compared to usual metamodelling techniques: *If a property is not defined explicitly, it can be used in a model in any arbitrary way.* This leads to a semi-structured way of modelling as known from XML. Not until this very basic idea is it possible to define a refinement in such a formal and straight-forward way.

7.4.2. Abstract concepts

In many situations it is desired to define a concept that is supposed to represent the common basis for a set of (refining) concepts. Nevertheless the concept representing the common basis should not be used for labelling vertices of the model directly: One of the refining concepts should always be used instead. These constraints can easily be defined by marking a concept as being abstract.

No node of the abstract must be labelled by an abstract concept. In inheritance hierarchy, both super- and sub-concepts may be abstract. Abstractness is not inherited by sub-concepts.

$$\begin{aligned} &((\mathrm{cDef}(v_c).P_{\omega 2}\!:\!isAbstract) \Subset C_{\omega 2}\!:\!True) \\ &\quad \rhd\!\rhd \\ &(\circlearrowleft \Subset C_\omega\!:\!c) \Rightarrow (\circlearrowleft \notin T_\omega\!:\!c) \end{aligned} \tag{7.12}$$

7.4.3. Complete concepts

Vertices labelled by incomplete concepts may contain any further property besides those explicitly defined (and thus restricted) in the metamodel due to the semi-structured approach, which has already been mentioned above. This renders the refinement-relation a purely restricting character in contrast to object-oriented approaches: Properties in refining

concepts which have been newly added place a restriction upon the property that could be arbitrarily allocated before (as it had not yet been defined in the refined concept).

If a concept in the metamodel is specified as being *complete*, the instances thereof may only contain those properties that have been explicitly defined. As a result, it will not be possible to add any new properties in refining concepts. As soon as a concept is characterised as being complete, all refining concepts are implicitly complete as well.

Complete concepts therefore exhibit a similar behaviour as the so-called final classes known from object orientation. This comparison is, however, not entirely true as refining concepts from complete concepts may definitely exist to restrict the already existing properties even stronger. The only thing is that new properties must not be added. The following additional definitions will primarily be required for a formalisation:

$$\begin{aligned} \text{sibling} =_{def} \quad & C_{\omega 2} : ConceptDef \downarrow (P_{\omega 2} : concept. \hookrightarrow . \leftrightarrows P_{\omega 2} : concept) \\ \text{relevantConceptDef} =_{def} \quad & C_{\omega 2} : AnyConceptDef \\ & \cup (\text{sibling} .^{\wedge} (P_{\omega 2} : refines. \text{sibling})) \end{aligned} \tag{7.13}$$

As there may be more than one concept definition for one concept in general, the edge function *sibling* directs towards all concept definitions restricting the same concept. Secondly, based on the edge function *sibling*, the edge function *relevantConceptDef* directs towards all relevant concepts that must be taken into account when looking for defined properties. Thereupon the condition for complete concepts can be formulated as follows:

$$\begin{aligned} & (\text{bool})(\text{cDef}(v_c).P_{\omega 2} : isComplete) \\ & \wedge V_{\omega 2} : v_p \notin (\text{cDef}(v_c). \text{relevantConceptDef} .P_{\omega 2} : propertyDef.P_{\omega 2} : property. \hookrightarrow) \\ & \quad \rhd\!\rhd \\ & (\circlearrowleft \Subset C_\omega : c) \Rightarrow (P_\omega : p = \varnothing) \end{aligned} \tag{7.14}$$

7.4.4. Weak concepts

The special property *composite* has already been introduced in Section 5.5.2, *Compositions*, p. 89. It indicates that a node is seen as *being a part of* another one. For some concepts it shall be claimed that nodes labelled by such a concept cannot exist without being part of another node. In this approach such concepts are marked as being *weak*.

Nodes labelled by a weak concept always have to be part of another concept in the sense of a compositional relationship. If a concept is labelled as weak, all sub-concepts will thus be weak as well.

$$\begin{aligned} & \text{cDef}(v_c).P_{\omega 2} : conceptType \Subset C_{\omega 2} : Weak \\ & \quad \rhd\!\rhd \\ & (\circlearrowleft \Subset C_\omega : c) \Rightarrow (\leftrightarrows P_\omega : composite \neq \varnothing) \end{aligned} \tag{7.15}$$

Those concepts for which this constraint is not desired are marked as *strong* concepts. Note that there are no additional constraints for *strong* concepts. Hence, strong concepts may also be part of another concept but it is not required for such strong concepts.

This procedure differs from other approaches such as UML. In UML a concept becomes implicitly weak when a composition is defined. In particular in cyclical compositional definitions this will cause problems: When, for example, a concept *Folder* is defined, which should contain sub-folders in terms of a composition, this will automatically lead to a contradiction in UML as there must be a root folder which does not have a parent although it is a weak concept (implicitly). In *M2L* such a concept would be marked as being *strong* although there is a composition to it stating that there may be root folders as well. Please refer to Section 7.4.9, *Compositional properties*, p. 131 as well.

7.4.5. Attribute concepts

As already seen in Section 5.5.3, *Attributes*, p. 89, a set of primitive types does not need to be defined in the present approach. Arbitrary types can be modelled by attribute concepts instead. Up to now, the difference of attribute concepts to the concepts introduced so far has not yet been discussed. UML, for example, does not exhibit any difference between attributes and compositions.

An attribute concept is the third and strongest type of concept in the present approach, indeed: While *strong* concepts cause no additional constraints, *weak* concepts must be at least part of another node. Finally, *attribute* concepts cause even more constraints than weak ones. Formally, it is defined as follows:

$$\begin{aligned} &\mathrm{cDef}(v_c).P_{\omega 2}:conceptType \Subset C_{\omega 2}:Attribute \\ &\quad \rhd\!\rhd \\ &(\circlearrowleft \Subset C_\omega:c) \Rightarrow (\leftrightarrows P_\omega:composite \neq \varnothing \\ &\quad \wedge \leftrightarrows *_\omega = (\leftrightarrows P_\omega:composite \uplus \leftrightarrows P_\omega:composite)) \end{aligned} \tag{7.16}$$

$$\begin{aligned} &\mathrm{cDef}(v_c).P_{\omega 2}:conceptType \Subset C_{\omega 2}:Attribute \\ &\wedge \mathrm{empty?}(\mathrm{cDef}(v_d).(\mathrm{relevantConceptDef} \downarrow \sigma(P_{\omega 2}:conceptType \Subset C_{\omega 2}:Attribute))) \\ &\quad \rhd\!\rhd \\ &(\circlearrowleft \Subset C_\omega:c) \Rightarrow (\mathrm{empty?}(P_\omega:composite \downarrow C_\omega:d)) \end{aligned} \tag{7.17}$$

First, (7.16) states that *attribute* concepts must not have any incoming edges besides the compositional ones. Thus, there must be exactly two incoming edges: One labelled by *composite* and the second one having the same source node and representing the actual property. Secondly, (7.17) states that *attribute* concepts may only contain attribute concepts. Thus, all nodes that are transitively reachable from a node labelled by an attribute concept over the property *composite* must be labelled by attribute concepts again.

Note that it is not forbidden that a node labelled by an attribute concept has an outgoing edge to a node labelled by a strong or weak concept, as long as there is no compositional link in between. As soon as a concept is characterised as being an attribute concept, all refining concepts are attribute concepts as well.

This definition says that an attribute node cannot be referenced by other nodes except the containing property of its parent node. Such a concept is particularly crucial for database implementations as no extent has to be defined for attribute concepts.

7.4.6. Enumeration concepts

The way of representing enumerations in M-graphs has already been defined in Section 5.5.3, *Attributes*, p. 89. In the present approach, elements of enumeration concepts will, in turn, be realised by a concept respectively. A simple example of an enumeration concept would be the data type *Boolean* including the two elements *True* and *False*. Herein, both *Boolean* as well as *True* and *False* will be understood as concepts. The actual enumeration type will, however, not occur in the model. Instead, the concepts of the elements – i. e. *True* and *False* – will be assigned to the respective nodes. Enumeration concepts therefore typically form the leaves of a model graph.

The formal definition of enumeration concepts is done in a different way, as it is treated as a specialisation of the already known concept definition. It will be defined in detail as follows:

$$
\begin{aligned}
&\mathcal{G}^{\omega 2}((\circlearrowleft \Subset C_{\omega 2}:EnumerationConceptDef \Rightarrow (\\
&\quad \circlearrowleft \Subset C_{\omega 2}:ConceptDef\\
&\quad \wedge P_{\omega 2}:enumElement \Subset C_{\omega 2}:EnumElementConceptDef\\
&\quad \wedge P_{\omega 2}:enumElement \Subset P_{\omega 2}:composite\\
&\quad \wedge |P_{\omega 2}:enumElement| \geq 2\\
&\quad \wedge |P_{\omega 2}:refines| = 0\\
&\quad \wedge P_{\omega 2}:isAbstract \Subset C_{\omega 2}:True\\
&\quad \wedge P_{\omega 2}:isComplete \Subset C_{\omega 2}:True\\
&\quad \wedge P_{\omega 2}:conceptType \Subset C_{\omega 2}:Attribute\\
&\quad \wedge |P_{\omega 2}:propertyDef| = 0\\
&\quad \wedge |P_{\omega 2}:additionalConstraint| = 0\\
&))\\
&\wedge (\circlearrowleft \Subset C_{\omega 2}:EnumElementConceptDef \Rightarrow (\\
&\quad \circlearrowleft \Subset C_{\omega 2}:ConceptDef\\
&\quad \wedge P_{\omega 2}:refines = \leftrightarrows P_{\omega 2}:enumElement\\
&\quad \wedge P_{\omega 2}:isAbstract \Subset C_{\omega 2}:False\\
&\quad \wedge P_{\omega 2}:isComplete \Subset C_{\omega 2}:True\\
&\quad \wedge P_{\omega 2}:conceptType \Subset C_{\omega 2}:Attribute\\
&\quad \wedge |P_{\omega 2}:propertyDef| = 0\\
&\quad \wedge |P_{\omega 2}:additionalConstraint| = 0\\
&\quad \wedge \leftrightarrows P_{\omega 2}:composite \Subset C_{\omega 2}:EnumerationConceptDef\\
&)))
\end{aligned}
\tag{7.18}
$$

While the concept *EnumerationConceptDef* represents the enumeration in total, the concept *EnumElementConceptDef* represents a single element of an enumeration. The concept *EnumerationConceptDef* therefore contains a new property, namely *enumElement*, which must contain at least two enumeration elements. As has been discussed above, each *EnumElementConceptDef* must refine the *EnumerationConceptDef* containing said element. Finally, enumerations are always defined as attribute concepts.

7.4.7. External concepts

Due to the present approach multiple metamodels can easily be combined. In situations like these, concept definitions referencing the same qualified name (thus the same package location and the same concept name), restrict one and the same concept within a model. Thus, the resulting constraints for both concept definitions must hold.

The more frequent case requires that a metamodel references a concept which is (externally) defined in another metamodel. Therefore a concept is required which represents a stub of a concept definition. The stub itself does not add any additional constraint.

The formal definition of external concepts is done in the same way as for enumerations, as it is treated as a specialisation of the already known concept definition. The formal definition will in detail be defined such that no additional constraint for the concept is defined: Hence, an external concept refines no other concept, is not abstract, is not complete, is marked as a strong concept, no property definitions are allowed, and finally no additional constraints can be defined:

$$\begin{aligned}
&\mathcal{G}^{\omega 2}(\circlearrowleft \Subset C_{\omega 2}:ExternalConceptDef \Rightarrow (\\
&\quad \circlearrowleft \Subset C_{\omega 2}:ConceptDef\\
&\quad \wedge\, |P_{\omega 2}:refines| = 0\\
&\quad \wedge\, P_{\omega 2}:isAbstract \Subset C_{\omega 2}:False\\
&\quad \wedge\, P_{\omega 2}:isComplete \Subset C_{\omega 2}:False\\
&\quad \wedge\, P_{\omega 2}:conceptType \Subset C_{\omega 2}:Strong\\
&\quad \wedge\, |P_{\omega 2}:propertyDef| = 0\\
&\quad \wedge\, |P_{\omega 2}:additionalConstraint| = 0\\
&))
\end{aligned} \tag{7.19}$$

7.4.8. Global constraints: the concept *Any*

In order to define global constraints, a dedicated concept is introduced named *Any*. In detail also the qualified name *ORG.Metamodels.BasicConcepts.Any* is fixed. Thus, there may be only one concrete concept of that type. This concept is implicitly refined by every other concept. According to that, restrictions defined by this concept must hold for each and every node of a model. Hence, global constraints can easily be defined by this special type of concept.

$$\begin{aligned}
&\mathrm{cDef}(v_c) \Subset C_{\omega 2}:AnyConceptDef\\
&\quad \rhd\!\rhd\\
&\circlearrowleft \Subset C_{\omega}:c
\end{aligned} \tag{7.20}$$

Besides this definition, the formal definition for the any-concept is extended by refining the already known, general concept definition. An explicit refinement is particularly forbidden due to the implicit refinement. Additionally, the following shall be defined in detail:

$$\begin{aligned}
&\mathcal{G}^{\omega 2}(\circlearrowleft \Subset C_{\omega 2}:AnyConceptDef \Rightarrow (\\
&\quad \circlearrowleft \Subset C_{\omega 2}:ConceptDef\\
&\quad \wedge\, P_{\omega 2}:qualifiedName = \langle \text{"}ORG\text{"}, \text{"}Metamodels\text{"}, \text{"}BasicConcepts\text{"}, \text{"}Any\text{"} \rangle\\
&\quad \wedge\, |P_{\omega 2}:refines| = 0\\
&\quad \wedge\, P_{\omega 2}:isAbstract \Subset C_{\omega 2}:True\\
&\quad \wedge\, P_{\omega 2}:isComplete \Subset C_{\omega 2}:False\\
&\quad \wedge\, P_{\omega 2}:conceptType \Subset C_{\omega 2}:Strong\\
&\quad \wedge\, \text{empty?} \leftrightarrows P_{\omega 2}:refines\\
&))
\end{aligned} \tag{7.21}$$

7.4.9. Compositional properties

As has already been described, abstract words are neither acyclic nor trees or forests in general. A hierarchical structuring of the nodes in the sense of a part-of-relationship plays, however, a vital role in language design. It shall be possible to express that for example a state belongs to a particular automaton. As has been shown in Section 5.5.2, *Compositions*, p. 89, the present approach is well suited for the property *composite* which will always provide the children of a node or will always be empty in case it concerns a leave of a forest. The property *composite* therefore directs from the roots towards the leaves of a tree. Cycles are not allowed. As several roots shall be allowed, the property *composite* forms a forest. The conditions for the property *composite* can be formulated as follows:

$$\begin{aligned}&(\circlearrowleft \notin P_\omega : composite.\hat{}P_\omega : composite)\\ &\wedge (\text{set?}\ P_\omega : composite)\\ &\wedge (\text{singleton?}\ \leftrightarrows P_\omega : composite)\end{aligned} \tag{7.22}$$

With the help of this procedure, the part-of-relationship can be expressed independent of the metamodel in the form of an abstract word. Although it is not explicitly defined what the exact meaning of a composition link would be, it also allows for a modelling of the part-of-relationship without knowing the metamodel.

If a composition is defined within the metamodel, the property *composite* will explicitly be required in the model. Here it is important to remember that the property *composite* is not forbidden for non-compositions. This plays a vital role concerning the consistency of inheritance. Sub-concepts must fulfil all properties of their corresponding super-concepts. It is therefore very well possible that a property that has been defined as a simple association in the refined concept, will be refined to form a composition within the refining concept.

This procedure offers additional possibilities concerning metamodelling as regards the definition of abstract languages. Many languages require the ability to define or else reference constructs on the spot. The first one would require a part-of-relationship, whereas the latter one would not. It would be extremely difficult or even impossible to express such requirements in common metamodelling. Examples thereof would be an inline-type-definition, for example. The composition can be formalised as follows:

$$\begin{aligned}&\text{pDef}(v_c, v_p).P_{\omega 2} : linkType \Subset C_{\omega 2} : Composition\\ &\quad \blacktriangleright\!\blacktriangleright\\ &(\circlearrowleft \Subset C_\omega : c) \Rightarrow (P_\omega : p \Subset P_\omega : composite)\end{aligned} \tag{7.23}$$

It must be noted that the composition property for the respective property must no longer be set explicitly in refining concepts. As soon as it has been defined in one of the refined concepts, the property will propagate in all refining concepts.

7.4.10. Inverse properties

It is often desired, to define the inverse property of a particular property. In case of a conform model, a property will then only be able to occur along with the inverse property thereof. Such bidirectional relationships have already been introduced in Section 5.5.1, *Bidirectional associations*, p. 88.

At this point it should be discussed whether this redundant information that occurs when introducing bidirectional relationships should at all be represented explicitly in the model or

whether is could be omitted and thus the concept of inverse properties itself can be skipped – in particular because the edge inverse of the Edge Algebra can be used for describing the inverse property as well. The edge inverse is, however, always unordered. If, however, both directions of the property are supposed to be ordered, it becomes inevitable to model both directions in two opposite properties. On the hand could properties which are the inverse property itself not be mapped. One example thereof would be the property *spouse* which points to that person a person is married to.

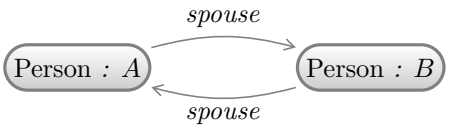


Figure 7.2.: Illustration of a symmetric property by means of a married couple

Here it shall be valid that in case a person A is married to a second person B, person B is also married to person A. In this case, $P_\omega : spouse = \leftrightarrows P_\omega : spouse$ must be valid. Thus, the inverse property cannot be removed as this would result in an elimination of the entire relationship.

All in all, the condition of inverse properties can be formalised as follows:

$$\begin{aligned} & V_{\omega 2} : v_q \Subset \mathrm{pDef}(v_c, v_p).P_{\omega 2} : opposite \\ & \quad \blacktriangleright\!\blacktriangleright \\ & (\circlearrowleft \Subset C_\omega : c) \Rightarrow (\mu P_\omega : p = \leftrightarrows P_\omega : q) \end{aligned} \tag{7.24}$$

Here, it should be noted that this kind of definition does not require an explicit definition of the inverse property, as long as the concept of the target node has not been defined as being complete. This procedure provides advantages in particular when metamodels are being modularised. Moreover is it possible to define different inverse properties via inheritance hierarchy in general. In the sense of the conjunction of all conditions via the inheritance hierarchy, all required inverse properties are also connected in a conjunctive way and are thus required simultaneously. This causes a property to have two or more inverse properties.

7.4.11. Multiplicities

Multiplicity can be used to indicate within which interval the size of the pomset for a property must range. Upper and lower limits will be defined with the help of natural numbers.

$$\begin{aligned} & (\text{edge})\, n \Subset \mathrm{pDef}(v_c, v_p).P_{\omega 2} : multiplicity.P_{\omega 2} : lower \\ & \quad \blacktriangleright\!\blacktriangleright \\ & (\circlearrowleft \Subset C_\omega : c) \Rightarrow (|P_\omega : p| \geq n) \\ & (\text{edge})\, n \Subset \mathrm{pDef}(v_c, v_p).P_{\omega 2} : multiplicity.P_{\omega 2} : upper \\ & \quad \blacktriangleright\!\blacktriangleright \\ & (\circlearrowleft \Subset C_\omega : c) \Rightarrow (|P_\omega : p| \leq n) \end{aligned} \tag{7.25}$$

7.4.12. Pomset-type restrictions

Properties are generally partially ordered multi-sets. With the help of the pomset restrictions, properties of the model can be restricted to specific types of pomsets. As has been

described in Section 4.1, *Relationship between different types of sets*, p. 61, it is differentiated between *Singleton*, *Set*, *Bag*, *List*, *Toset*, *Poset*, and *Pomset*. Formally, it is defined as follows:

$$
\begin{aligned}
&\mathrm{pDef}(v_c, v_p).P_{\omega 2} : pomsetRestriction \\
&\quad \Subset (C_{\omega 2} : Singleton \cup C_{\omega 2} : Set \cup C_{\omega 2} : Bag) \\
&\quad \blacktriangleright\!\blacktriangleright \\
&(\circlearrowleft \Subset C_\omega : c) \Rightarrow (\mathrm{bag?}\ P_\omega : p) \\
&\\
&\mathrm{pDef}(v_c, v_p).P_{\omega 2} : pomsetRestriction \\
&\quad \Subset (C_{\omega 2} : Singleton \cup C_{\omega 2} : Toset \cup C_{\omega 2} : List) \\
&\quad \blacktriangleright\!\blacktriangleright \\
&(\circlearrowleft \Subset C_\omega : c) \Rightarrow (\mathrm{list?}\ P_\omega : p) \\
&\\
&\mathrm{pDef}(v_c, v_p).P_{\omega 2} : pomsetRestriction \\
&\quad \Subset (C_{\omega 2} : Singleton \cup C_{\omega 2} : Set \cup C_{\omega 2} : Toset \cup C_{\omega 2} : Poset) \\
&\quad \blacktriangleright\!\blacktriangleright \\
&(\circlearrowleft \Subset C_\omega : c) \Rightarrow (\mathrm{poset?}\ P_\omega : p)
\end{aligned}
\tag{7.26}
$$

7.5. Semantics for Abstract Syntaxes – Part 2: Extended metamodelling concepts

In the previous section, the aim was to collect, consolidate and formalise the concepts common from the domain of metamodelling. In many situations, however, the current metamodelling concepts are not sufficient for describing real modelling languages. Although it is already possible to express particular context-sensitive properties by means of the basic concepts, there are, however, still many properties of languages which cannot be expressed by the basic concepts.

Therefore, the second part of the *conformsTo*-relation will be defined in this section. The extended metamodelling concepts are listed in detail in Table 7.2.

extended metamodelling concept
Additional concept constraints
Conditional properties
Context-sensitive domains
Inferred properties
Local keys, namespaces and visibility
Instantiating Properties

Table 7.2.: Overview of the extended metamodelling concepts

7.5.1. Additional concept constraints

First of all, a specification of arbitrary node predicates based on Edge Algebra within any concept definition shall be allowed. Here, the full expressiveness of Edge Algebra will be transferred to the metamodelling language *M2L*. Formally, the definition is based on the semantical meaning introduced in Definition 54.

$$\begin{array}{l} V_{\omega 2} : v = \mathrm{cDef}(v_c).P_{\omega 2} : additionalConstraint \\ \quad \rhd\!\rhd \\ (\circlearrowleft \Subset C_\omega : c) \Rightarrow [[v]] \end{array} \tag{7.27}$$

Please keep in mind that such an extension does not supersede additional constructs in a metamodelling language. As will be seen in the following sections, some of these constructs will result in complex Edge Algebra statements. Thus, the *appropriateness* of the metamodelling language *M2L* is highly increased.

7.5.2. Conditional properties

One of the main aims of the present metamodelling approach is to allow a language engineer to specify the abstract syntax of a language including all its consistency constraints. In our running example, all the ports must be connected for example. The advantage of such an approach is, of course, the exact definition of a language.

During a development process, however, situations will come up upon using the language in which some of the constraints should hold whereas others should not. In a late development stage, for example, every requirement should correspond to a respective implementation. In contrast to that, such a constraint will be meaningless in an early stage as no implementation does exist at all. In order to reflect this requirement within our metamodelling language *M2L*, all property definitions can be made conditional by adding an assumption. Hence, a property restriction is only relevant if the given assumption holds. In order to formalise conditional properties, the metamodel-to-model restriction will be redefined for properties introduced in Section 7.3, *Basic approach defining semantics for Abstract Syntaxes*, p. 124.

$$\begin{array}{lll} A \blacktriangleright\!\blacktriangleright B \Leftrightarrow_{def} & (A \wedge (\mathrm{pDef}(v_c, v_p).P_{\omega 2} : assumption = \varnothing) & \rhd\!\rhd \quad B) \\ & \wedge & \\ & (A \wedge (\mathrm{pDef}(v_c, v_p).P_{\omega 2} : assumption = V_{\omega 2} : v) & \rhd\!\rhd \quad ([[v]] \Rightarrow B)) \end{array} \tag{7.28}$$

This construct helps in building up a hierarchy of assumptions as multiple definitions can be added for one property in a single concept definition. These property definitions will then comprise increasingly stronger assumptions for constructing the hierarchy.

7.5.3. Context-sensitive domains

In metamodelling, generally all target nodes of a particular property are supposed to be an element of the set of all nodes labelled by a particular concept. A property definition may therefore contain a specification of its *domain*. Note that according to the refinement of the present approach refining concepts may also refine the domain of a property. In order to avoid a contradictory refinement, the refined domain will have to be a refined concept of the original domain itself or both domains have common refined concepts.

Although this construct is very basic, it has not been defined in the basic metamodelling concepts. The reason is that in the present approach, a more generic solution for specifying property domains is to be defined:

Up to now, the domain of a property has always been restricted to one particular concept. One simple example from the present running example will show that such conditions are not sufficient in most cases. In case the properties *fromPort* and *toPort* of the concept

Channel were restricted to the concept *Port* only, invalid models could possibly occur, as shown in Figure 7.3.

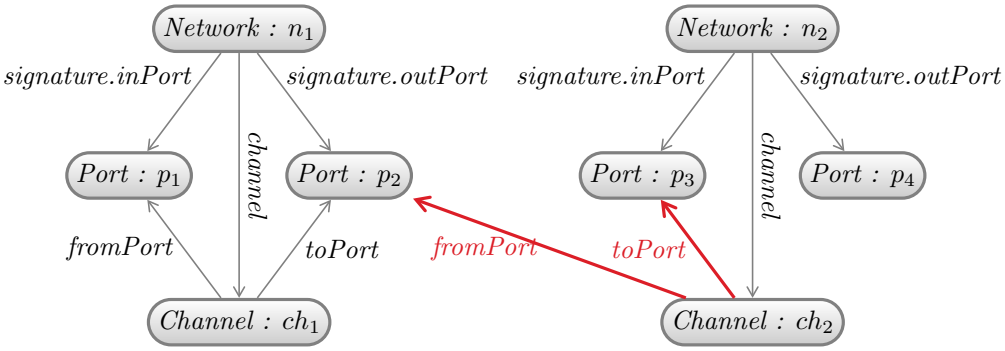


Figure 7.3.: Example: Invalid channels

Channel c_2 connects two ports of different networks n_1 and n_2, although channel c_2 should only connect ports from the network n_2. Additionally, a second error occurs: The property *toPort* directs towards an input port which, in turn, is unwanted although the port is located in the right network. Both errors cannot be avoided with the current metamodelling concepts

The necessary constraints have already been expressed in Section 6.4.2, *Additional Invariants*, p. 115 using Edge Algebra. Now, an extended, context-sensitive domain construct defining such conditions shall be used: In Edge Algebra, the current (simplified) restriction of the domain to a specific concept would be by expressing said by way of a *consistsOf* operator, wherein the right operand thereof would always be a concept equipped with the edging operator. In the present example, the property *fromPort* would be expressed as follows:

$$\circlearrowleft \Subset C_\omega : Channel \Rightarrow \\ P_\omega : fromPort \Subset C_\omega : Port$$

If general expressions of Edge Algebra would now be allowed within the domain, the example could be expressed correctly and as originally desired:

$$\circlearrowleft \Subset C_\omega : Channel \Rightarrow \\ P_\omega : fromPort \Subset \leftrightarrows P_\omega : channel.P_\omega : signature.P_\omega : inPort$$

The domain of the property *fromPort* would therefore now refine in the context of a channel from the original expression of $C_\omega : Port$ to $\leftrightarrows P_\omega : channel.P_\omega : signature.P_\omega : inPort$. This general form of the domain is called *context-sensitive* as the set of allowed nodes does depend on the respective source node. It must be noted that this variant does not explicitly express that *fromPort* is only allowed to direct towards those nodes to which the concept *Port* is allocated. This could, however, be represented explicitly as follows:

$$\circlearrowleft \Subset C_\omega : Channel \Rightarrow \\ P_\omega : fromPort \Subset C_\omega : Port \downarrow \leftrightarrows P_\omega : channel.P_\omega : signature.P_\omega : inPort$$

Simple properties which are supposed to allow more than one concept can, however, also be expressed via general Edge expressions within the domain. One example would be that the identifier should not only be a string but also a natural number:

$$\circlearrowleft \Subset C_\omega : Channel \Rightarrow \\ P_\omega : identifier \Subset (C_\omega : String \cup C_\omega : Natural)$$

Formally, the metamodelling construct for context-sensitive domains is defined by the following condition:

$$V_{\omega 2} : v = \mathrm{pDef}(v_c, v_p).P_{\omega 2} : domain \\ \blacktriangleright\blacktriangleright \\ (\circlearrowleft \Subset C_\omega : c) \Rightarrow (P_\omega : p.1(P_\omega : template \oplus \circlearrowleft)) \Subset [[v]] \tag{7.29}$$

Please note that $1(P_\omega : template \oplus \circlearrowleft)$ results from another metamodelling construct defined in Section 7.5.6, *Instantiating Properties*, p. 145. Currently it can be assumed that $P_\omega : template = \varnothing$ simplifies the condition to:

$$V_{\omega 2} : v = \mathrm{pDef}(v_c, v_p).P_{\omega 2} : domain \\ \blacktriangleright\blacktriangleright \\ (\circlearrowleft \Subset C_\omega : c) \Rightarrow P_\omega : p \Subset [[v]] \tag{7.30}$$

7.5.4. Inferred properties

Many situations exhibit sensitive properties which can be inferred from other properties. A similar construct in UML class diagrams would be a method without any parameters. In the present approach, the implementation of such inferred properties is again defined by an Edge Algebra statement. The formal definition is quite similar to (7.30) except that the *consists-of* operator is replaced by an *equality* operator.

$$V_{\omega 2} : v = \mathrm{pDef}(v_c, v_p).P_{\omega 2} : inferredValue \\ \blacktriangleright\blacktriangleright \\ (\circlearrowleft \Subset C_\omega : c) \Rightarrow P_\omega : p = [[v]] \tag{7.31}$$

Note that an inferred property requires an edge within the M-graph which will then be constrained as defined. In contrast to that, the resulting value is calculated dynamically by a method which also allows to parameterise a method. Hereupon, all other constraints will also need to be fulfilled by inferred properties. Inferred properties do, for example, not need to reference nodes labelled by attribute concepts.

7.5.5. Local keys, namespaces and visibility

Up to now, nodes have always been identified via an element of the node set. The identifier of the element is therefore always globally – i. e. within the model – unique. This is also well sufficient for the mathematical description of an abstract word. Normally, a particular form is desired to be specified for an identifier on the one hand, and on the other hand, a hierarchical structure for the design of the modelling language is also desired. Global uniqueness is by far too restrictive in practice. Different, local variables may very well have the same name in a programming language such as C, for example, as long as they are defined in different validity ranges. Similar to relational databases, the selected properties of a node are therefore supposed to take over the role as a key within the model. This turns the identifiers themselves into referenced nodes of a model. In contrast to databases these keys are, however, not supposed to be globally unique to meet the requirements to a reasonable language design. These keys are called *local keys*.

The property *lkey*

In order to be able to express within the model that one node represents the key of another node, the specific property lkey will be introduced – similar to the parent relationship for compositions. Basically, each node (except for the own one) may function as a key. It is thus not required that these keys necessarily need to be character sequences. Natural numbers or data values would also be possible, for example. Figure 7.4 shows a simple example based on our running example.

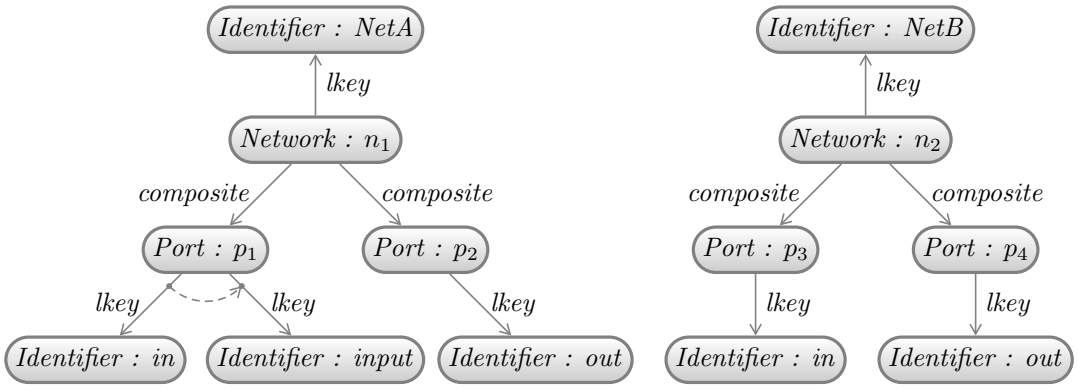


Figure 7.4.: Example: assigning local keys

The present example will now assign a local key to each network and each port by way of the property *lkey* . Network n_1, for example, has the character sequence “NetA” as a key and so on. It can be seen at once that both port p_1 and port p_3 have the same key. As the respective ports are not located within the same network, this is well allowed and also desired. This is already an example for the need of a non-global key concept.

This example also proves that the present approach may generally also comprise several keys for one node: Port p_1 comprises the two keys of “in” and “input”. It can be seen that these local keys are ordered because exactly one *primary local key* is requested being the first element. All other local keys need to be unordered afterwards. Hence, the property *lkey* in general is a pomset such that the following equation holds:

$$\text{singleton?}\, 1P_\omega : lkey \wedge \text{set?}[2;*]P_\omega : lkey \tag{7.32}$$

Thus, there is one first element representing the primary local key, followed by a set of alternative local keys without any additional order. In contrast to alternative local keys the primary local key is used for referencing by default.

Both primary and alternative local keys can be easily defined within a metamodel in *M2L* by marking the corresponding property. Formally, the following two conditions will be defined:

$$\begin{aligned}
&(\text{bool})(\text{pDef}(v_c, v_p).P_{\omega 2} : isPrimaryLocalkey) \\
&\quad \blacktriangleright\blacktriangleright \\
&(\circlearrowleft \Subset C_\omega : c) \Rightarrow ((|P_\omega : template| = 0) \Rightarrow (P_\omega : p = 1P_\omega : lkey)) \\
&\wedge \\
&(\text{bool})(\text{pDef}(v_c, v_p).P_{\omega 2} : isAlternativeLocalkey) \\
&\quad \blacktriangleright\blacktriangleright \\
&(\circlearrowleft \Subset C_\omega : c) \Rightarrow ((|P_\omega : template| = 0) \Rightarrow (P_\omega : p \Subset [2;2]P_\omega : lkey))
\end{aligned} \tag{7.33}$$

Note that the present definition allows a definition of multiple properties as alternative local keys. In contrast, only a single primary local key can be defined.

Local uniqueness

As the keys are not supposed to be globally unique, another definition of uniqueness must be provided, which is not as restrictive, e. g. the so-called *local uniqueness*. Upon reflecting the example mentioned above, it will become obvious that only states within an automaton need to have different names. A generalisation of this requirement can be achieved by demanding that the keys of all children of a node (in the sense of the composition) need to be unique. In addition, it holds that all nodes without parents need to have keys that are unique among themselves. In the present example, this would be the two networks. The property *lkey* is therefore subdivided into three parts in the present example, which need to be unique respectively. These parts will be marked in respectively different colours in Figure 7.5.

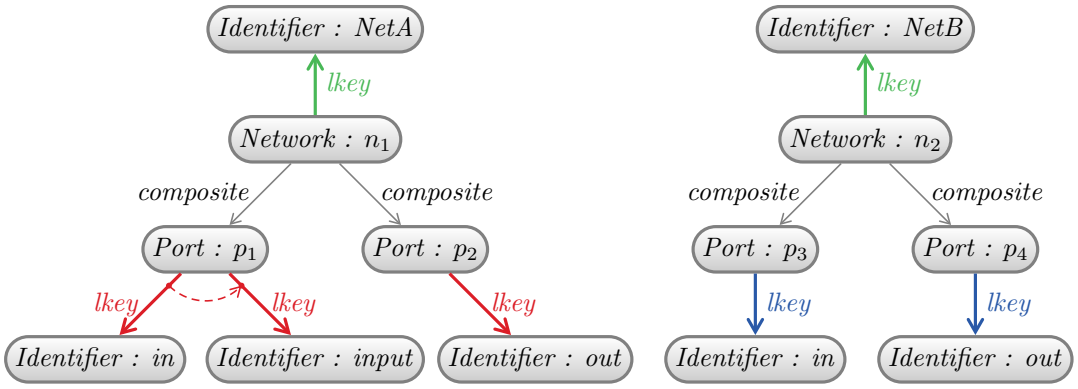


Figure 7.5.: Example: unique parts of the property *lkey*

The local keys of the two networks, namely n_1 and n_2, (marked in green) need to be unique as they do not have a parent. The local keys of the ports p_1 and p_2 (marked in red) and

the ports p_3 and p_4, respectively (marked in blue) need to be unique as they are both part of the same networks n_1 and n_2 respectively. The (still simplified) form of local uniqueness can be formalised as follows:

$$\begin{aligned} &\text{poset?}(\sigma\,\text{root?}\,.P_\omega{:}lkey) \\ &\wedge\,\text{poset?}\,P_\omega{:}lkey \end{aligned} \tag{7.34}$$

Further it shall be required that whenever a node within a M-graph is referenced, a local key must be defined:

$$\leftrightarrows(*_\omega \Downarrow^* P_\omega{:}composite) \neq \leftrightarrows P_\omega{:}template \Rightarrow |P_\omega{:}lkey| \geq 1 \tag{7.35}$$

For the moment it can be assumed that $P_\omega{:}template = \varnothing$, thus simplifying the formula to the following condition which is more easy to understand:

$$(\leftrightarrows(*_\omega \Downarrow^* P_\omega{:}composite) \neq \varnothing) \Rightarrow |P_\omega{:}lkey| \geq 1 \tag{7.36}$$

Lkey holes

In principle, it is not required for each node to dispose of a property *lkey*. If a vertex does not have a local key, this is a so-called *lkey hole*. This rises the question about how to deal with these lkey holes. In the following, an example shall be provided in the scope of the present running example. A network comprises a signature having ports, in turn. Whereas networks and ports have a name which also represents the local key, signatures do not have a key on their own. Signatures are thus lkey holes. Networks additionally have named channels. It is required for the names of all ports and channels to be unique within a network. Figure 7.6 illustrates the situation:

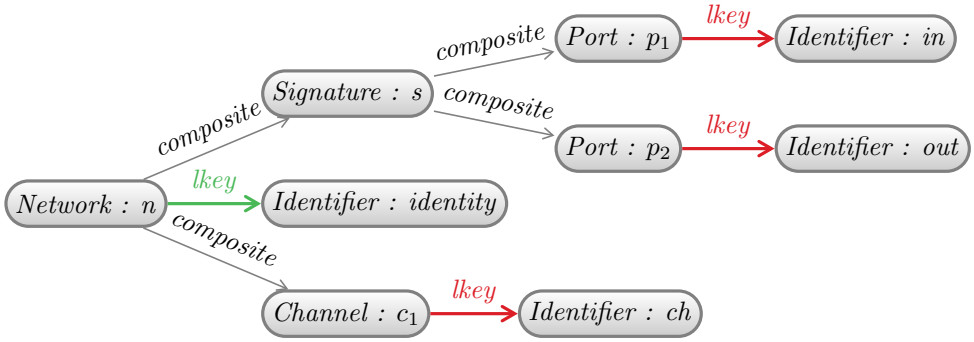


Figure 7.6.: Example: abstract word including *lkey holes*

This example would therefore like to have the three keys marked in red as being unique. According to the current definition it would, however, only be ensured that all ports have unique names as node s is located inbetween node n and the ports. This problem could, however, be avoided by connecting the ports directly to the network. This would, however, in return mean that the concept of the signature would have to be abandoned, which is not desired too often.

The same result could, however, also be achieved by extending the condition for local uniqueness such that lkey holes – i. e. nodes to which no key is assigned, i. e. the signature in the present example – are skipped. The entire condition for local uniqueness will thus be defined as follows:

$$\begin{aligned} &\text{poset?}(\sigma\,\text{root?}\,.P_\omega{:}lkey) \\ &\wedge\,\text{poset?}(^\wedge(P_\omega{:}composite \downarrow \sigma\,|P_\omega{:}lkey| = 0).P_\omega{:}lkey) \end{aligned} \tag{7.37}$$

Concept and property independent uniqueness

It must be noted that the present definition will require a uniqueness across all children, i. e. also across different concepts and properties. So will not only all names of the ports among each other be unique, but all names of the ports, channels and all other concepts, which are still composed to a network, will need to be different.

In principle, there are situations which do not require such a requirement. This will always be the case if due to domain definition (see Section 7.5.3, *Context-sensitive domains*, p. 134) it will be obvious for a reference from the very beginning which concepts is to be referenced. The properties *fromPort* and *toPort* of a channel, for example, can only reference ports anyway. So if a channel having the same name as a port exists, this would not be a problem in this specific situation.

Nonetheless, the approach of weakening the key concept that way is abandoned for the following reasons:

1. In the case of real modelling languages it almost always happens that it can no longer be determined from the context which node is to be referenced, as concerning the context, both nodes would be allowed.
2. Such similarities as regards names should be avoided, let alone in the sense of a good language design, as otherwise a modeller would have to reconstruct said contextual references on his/her own. The question comes up, whether the same name should really be given to a channel and a state within the same network – even if that would be possible in principle.
3. A weakening would result in that a node without context could generally no longer be identified uniquely via its key. The same would be true for the canonical names, which will be introduced in the following.

Composition path and canonical keys

Now that the keys specified by the property *lkey* are not globally unique, the question comes up of how to be able to identify them in a globally unique way. This situation shall be illustrated by way of the present introducing example in Figure 7.4: As has already been described, the key "in", for example, is not globally unique. Only along with the information within which network the port is located, will become obvious which port is meant. The key for network n_1 from the present example would be "NetA". Thus, something as "NetA.in" would be expected to be a globally unique key.

These so-called canonical keys can be generalised as follows: Due to the nature of the composition will the definition of a so-called composition path be allowed by the property *composite*. A unique path for each node of a model exists along the inverse property *composite* until a root node – i. e. a node without parents. If the keys of all these nodes are then

concatenated, exactly that canonical key, *ckey* in short, will be obtained. As more than one local key does generally exist for a node, also more than one canonical key does exist for a node. The canonical keys *ckey* can be derived from the properties *lkey* and *composite* as follows:

$$\begin{array}{lll} \mathrm{ref}_{\mathrm{ckey}}: & \rightarrow & \mathcal{E}_V \\ & \mapsto & \mathrm{ref}_{\mathrm{ckey}} =_{def} \pi(\gtrless^{\wedge} \leftrightarrows P{:}composite.\mu P{:}lkey) \end{array} \tag{7.38}$$

Here, the respective local keys will be totally ordered from root to target node. If more than one canonical key exists, each (totally ordered) connected component represents a single canonical key. The lkey holes have also already been considered in this definition. The respective nodes will simply be skipped. Figure 7.7 shall illustrate the definition of the canonically global keys.

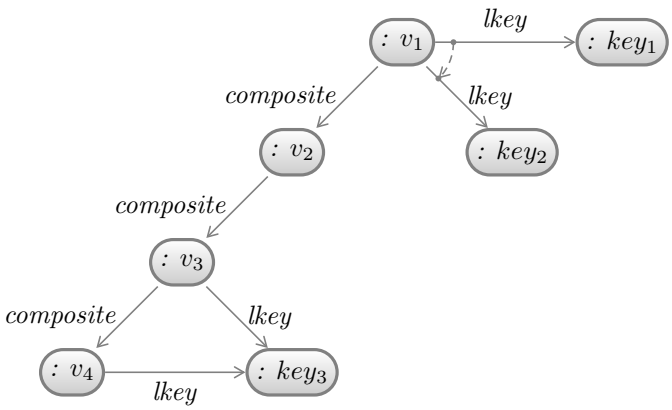


Figure 7.7.: Example: building canonical keys

The canonical keys of the four nodes will result in the following with respect to the present example:

$$\mathrm{ref}_{\mathrm{ckey}}(v_1) = \{key_1, key_2\}$$

$$\mathrm{ref}_{\mathrm{ckey}}(v_2) = \varnothing$$

$$\mathrm{ref}_{\mathrm{ckey}}(v_3) = \left\{ \begin{array}{l} key_1 \rightarrow key_3 \\ key_2 \rightarrow key_3 \end{array} \right\}$$

$$\mathrm{ref}_{\mathrm{ckey}}(v_4) = \left\{ \begin{array}{l} key_1 \rightarrow key_3 \rightarrow key_3 \\ key_2 \rightarrow key_3 \rightarrow key_3 \end{array} \right\}$$

The canonical keys for the model of our running example are, as defined in Figure 7.4:

$$\text{ref}_{\text{ckey}}(p_1) = \left\{ \begin{array}{l} NetA \rightarrow in \\ NetA \rightarrow input \end{array} \right\}$$

$$\text{ref}_{\text{ckey}}(p_2) = \langle NetA,\, out \rangle$$

$$\text{ref}_{\text{ckey}}(p_3) = \langle NetB,\, in \rangle$$

$$\text{ref}_{\text{ckey}}(p_4) = \langle NetB,\, out \rangle$$

The local uniqueness of the property *lkey* along with the required uniqueness of the parent from the property *composite* ensures that the canonical key defined as is will be globally unique.

Referencing by context-sensitive keys

In principle, any node having a local key defined, can be referenced with the help of the canonical keys. The modelling language should, however, support the modeller in shortening the (generally relatively long) canonical key in certain situations. Let's go back to the present running example in Figure 7.8:

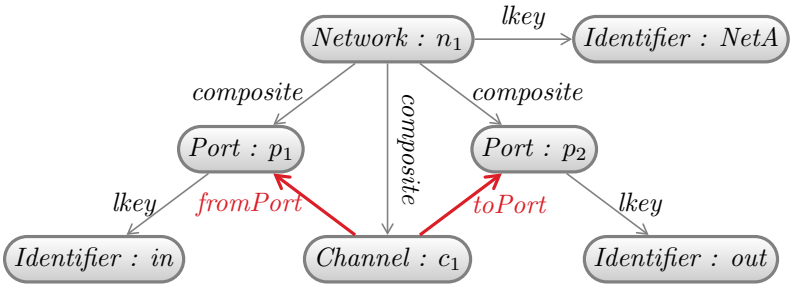


Figure 7.8.: Example: demonstrating context sensitive keys

In case the modellers want to indicate the properties *fromPort* and *toPort* of channel c_1 via a local key, they would, up to now, always have to indicate the canonical key, namely $\langle NetA, in \rangle$ and $\langle NetA, out \rangle$ respectively, although it would actually be obvious that the ports of the network, wherein the channel itself is located as well, are also always meant (see property *composite* between n_1 and c_1). $\langle in \rangle$ and $\langle out \rangle$ respectively should therefore be sufficient. Situations like these can always be re-found in the design of languages. Here, local variables can be considered. In these cases it shall also solely be referenced via the local variable name. This is the reason why here as well a generalisation for referencing shall be provided.

The basic idea here is that the canonical key describing the reference can be shortened depending on the context – i. e. depending on the node from which it is referenced. Informally spoken this means that nodes which are closer to each other in the composition tree shall also be referenced via shorter keys. The node from which the reference emanates will be called source node. The given key will be interpreted as being a context-sensitive key. Beginning at the source node, it will be tried to resolve that key. If no suitable node will be found, the procedure will be repeated at the parent of the source node. This will happen until a root node was reached. In the last step it will be tried to resolve the given key as a global key. An abstract example is supposed to illustrate the process upon resolving keys in Figure 7.9:

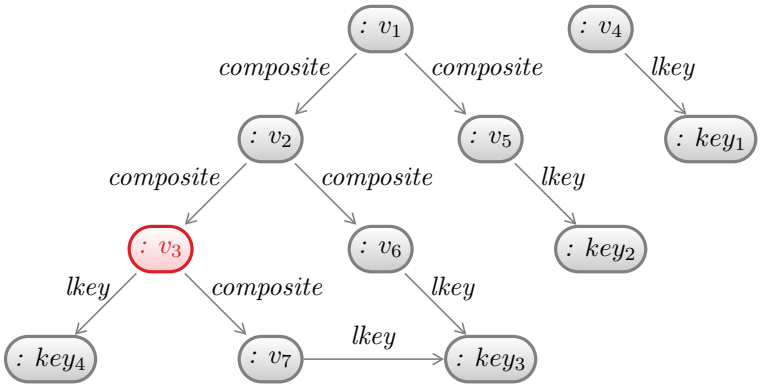


Figure 7.9.: Example: referencing nodes by context-sensitive keys

Beginning at the source node v_3, nodes v_4, v_5 and v_7 can already be referenced by the simple (local) keys key_1, key_2 and key_3. The keys will be checked in order along the inverse property *composite* beginning at v_7 to v_4. Node v_6 cannot be referenced by its local key from node v_3, as v_7 has the same local key key_3. Hence, node v_6 is not visible by its local key.

Visibility

As has just been seen, resolving references may result in an overlapping of nodes as is also known from common programming languages. Nevertheless there is one major difference: Although node v_6 is not visible by its local key, it can still be referenced by its canonical key. Note that the canonical key will again be $\langle key_3 \rangle$ in our example. The difference is that this time it is known that the key represents a canonical key, but for node v_7 the canonical key is $\langle key_4, key_3 \rangle$.

Additional context

Some situations desire a resolving of keys, extending beyond the currently presented procedure, i. e. to include further nodes of the model in the context, besides the source nodes. This procedure is known from traditional programming languages in particular in connection with libraries. Here, a respective import or include statement includes libraries. The respective section may then reference the elements of the library – such as Java classes – by their simple name, instead of indicating the canonical key.

For this purpose, another perfect property will be introduced besides *lkey*: the property *context*. Each node is able to extend its context via the property *context* by a so-called *additional context*. This is to ensure that the node itself will receive higher priority towards the additional context. This is, however, generally not valid for conflicts within additional context. Only if the property *context* is totally ordered, it is ensured that a node will be referenced uniquely. Due to the order of the property *context*, a prioritisation will be defined. A language designer may, however, also specify the context property in an unordered way. In this case, the simple key would not be unique, resulting in the fact that the canonical key would have to be used instead.

Differentiation between referenceable node and valid edge

Generally it is true that each node having a property *lkey*, can be referenced by any other node – at least via its unique canonical key. But even if a node can be referenced by another node via its simple key – either via the original or the extended context, this does not necessarily mean that this edge will be valid within the model as well. Having a look at the networks from the present running example, each port in a network could be referenced by a channel within the same network. This does, however, not imply that this should be allowed within the model at all. An output port of a network shall, for example, not be referenced by the property *fromPort* within a channel.

The same applies for the import statements discussed in the previous section as well: The presented property *context* only does half the work. Nodes of the library can only be referenced by their simple name. The fact that this reference is allowed within the model at all, must be defined beyond the properties *context* and *lkey*.

Formal definition for referencing and de-referencing

As a summary, the formal definitions for both referencing a node and de-referencing a given key for three types of keys that have been introduced shall be provided: canonical keys, local keys as well as context-sensitive keys.

All referencing functions $\mathrm{ref}_{\mathrm{ckey}}$, $\mathrm{ref}_{\mathrm{lkey}}$, and $\mathrm{ref}_{\mathrm{cskey}}$ return a pomset of which each connected component represents one alternative. Hence, each connected component is totally ordered and thus a list.

For both $\mathrm{ref}_{\mathrm{lkey}}$ and $\mathrm{ref}_{\mathrm{cskey}}$ one parameter is required: $s \in \mathcal{E}_V$ is an edge function directing towards the source nodes which should be used for referencing. If more than one source node is given, the keys for all source nodes will be returned. Order and duplicates of source nodes will be ignored. The referenced node is defined by the reflexive edge $\circlearrowleft$.

- The definition for $\mathrm{ref}_{\mathrm{ckey}}$ has already been defined in (7.38). The definition thereof was repeated herein for completeness. $\mathrm{ref}_{\mathrm{ckey}}$ is a constant function as the canonical key is independent of the source node.

- $\mathrm{ref}_{\mathrm{lkey}}$ only allows to reference nodes that are part of the composition sub-tree of the source node. Thus, if the source node is not a parent of the referenced node, an empty pomset will be returned. A canonical key will never be returned.

- $\mathrm{ref}_{\mathrm{cskey}}$ returns the full-featured context-sensitive key for a node from given source nodes. The additional context is also taken into account.

$$
\begin{array}{lllll}
\mathrm{ref}_{\mathrm{ckey}}: & & \to & \mathcal{E}_V & \\
& & \mapsto & \mathrm{ref}_{\mathrm{ckey}} =_{def} \pi(\gtrdot^{\wedge} \leftrightarrows P{:}\,composite.\mu P{:}\,lkey) & \\
\mathrm{ref}_{\mathrm{lkey}}: & \mathcal{E}_V & \to & \mathcal{E}_V & \\
& s & \mapsto & \mathrm{ref}_{\mathrm{lkey}}(s) =_{def} \pi(((\mu\epsilon s.P{:}\,composite.{}^{\wedge}P{:}\,composite) & \\
& & & \downarrow^{\wedge} \leftrightarrows P{:}\,composite).\mu P{:}\,lkey) & \\
\mathrm{ref}_{\mathrm{cskey}}: & \mathcal{E}_V & \to & \mathcal{E}_V & \\
& s & \mapsto & \mathrm{ref}_{\mathrm{cskey}}(s) & \\
& & & =_{def} \mathrm{ref}_{\mathrm{temp}}(\varnothing, s.(^{\wedge} \leftrightarrows P{:}\,composite \oplus P{:}\,context)) & (7.39)
\end{array}
$$

while

$$
\begin{array}{lllll}
\mathrm{ref}_{\mathrm{temp}}: & {\mathcal{E}_V}^2 & \to & \mathcal{E}_V & \\
& \langle \bar{k}, s \rangle & \mapsto & \mathrm{ref}_{\mathrm{temp}}(\bar{k}, s) & \\
& & & =_{def} \begin{cases} \varnothing & \text{if } s = \varnothing \\ (\mathrm{ref}_{\mathrm{lkey}}(1\,s) \setminus \bar{k}) \cup \mathrm{ref}_{\mathrm{temp}}(\bar{k} \cup (\pi(& \\ \quad 1\,s.{}^{\wedge}P{:}\,composite.\mu P{:}\,lkey), [1]s)) & \text{if } s \neq \varnothing \end{cases} &
\end{array}
$$

For each referencing function a corresponding de-referencing function $\mathrm{deref}_{\mathrm{ckey}}$, $\mathrm{deref}_{\mathrm{lkey}}$, and $\mathrm{deref}_{\mathrm{cskey}}$ also exists:

$$
\begin{array}{lllll}
\mathrm{deref}_{\mathrm{ckey}}: & \mathcal{E}_V & \to & \mathcal{E}_V & \\
& k & \mapsto & \mathrm{deref}_{\mathrm{ckey}}(k) & \\
& & & =_{def} \sigma(k \subseteq \pi(\gtrdot^{\wedge} \leftrightarrows P{:}\,composite.\mu P{:}\,lkey)) & \\
\mathrm{deref}_{\mathrm{lkey}}: & \mathcal{E}_V & \to & \mathcal{E}_V & \\
& k & \mapsto & \mathrm{deref}_{\mathrm{lkey}}(k) =_{def} \sigma(n \Subset {}^{\wedge} \leftrightarrows P{:}\,composite & (7.40) \\
& & & \wedge\, k \subseteq \pi(\gtrdot^{\wedge}(\leftrightarrows P{:}\,composite \downdownarrows^{*} n).\mu P{:}\,lkey)) & \\
\mathrm{deref}_{\mathrm{cskey}}: & \mathcal{E}_V & \to & \mathcal{E}_V & \\
& k & \mapsto & \mathrm{deref}_{\mathrm{cskey}}(k) & \\
& & & =_{def} 1((^{\wedge} \leftrightarrows P{:}\,composite \oplus P{:}\,context).\,\mathrm{deref}_{\mathrm{lkey}}(k)) &
\end{array}
$$

Note that the property *context* must be totally ordered in order to ensure a unique result. In principle, also a partial order is allowed. If the context-sensitive key does not return a unique result, the canonical key will have to be used.

7.5.6. Instantiating Properties

Re-use represents an essential concept in modelling languages used in practice. Examples can be found in the most diverse fields. The most common example will surely be (pre-)defining functions and the latter, normally multiple use thereof for defining further functions. Another but similar example from the present running example would be the definition of components and the multiple instantiation thereof in networks, which shall be considered in more detail later on.

Although re-use represents a common requirement to modelling languages, it is not explicitly supported by current metamodelling approaches. It is very difficult to model re-use with conventional means. Although re-use represents a very simple and intuitively easily understandable concept in principle, a metamodel which is to support such a concept, will turn out to be extremely confusing and complex. The actual concepts of the language will mix with those becoming technically necessary due to re-use. This is the reason why a specific construct for instantiation in metamodelling shall be introduced in the following.

In order to provide a better understanding of the problems upon the introduction of re-use, a closer look shall be taken at the present running example. Figure 7.10 provides the relevant part from the present metamodel that was introduced in Section 3.4, *A first, semi-formal abstract syntax*, p. 50 for the present running example. In addition to the current scope, an additional concept, namely *ProcessingUnit*, will be introduced. A *ProcessingUnit* represents a physical unit that may execute a set of components thereon. All components are mapped to a processing unit during a deployment step. This mapping is represented by the property *deployedComponent* of the concept *ProcessingUnit*. Especially in this context is it important for a single instance of a component to be referenced herein.

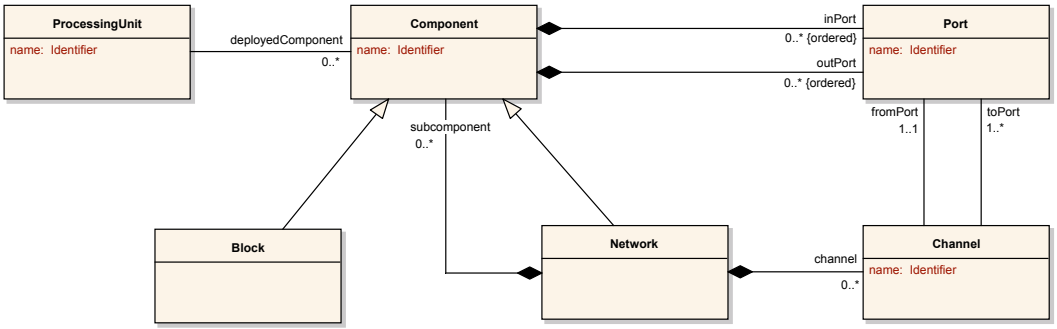


Figure 7.10.: Dataflow networks without instantiation

Once again, our abstract syntax still does not consider re-use. In order to illustrate the problem, a short exemplary model will be developed. Said network should realise a double integration by connecting two integrator networks as defined in Figure 3.3 in series. The resulting network is shown in Figure 7.11.

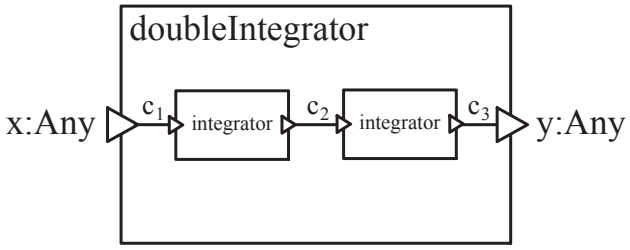


Figure 7.11.: Dataflow network including two identical instances

Here, re-use will be taken into consideration. The network *doubleIntegrator* uses network *integrator* twice. With the current metamodel, modellers would have to re-define the network *integrator* each time. The fact that it is actually the same network, that has been used twice, would not be reflected by the model. The fact that one component can be used in one network at most is primarily due to the required composition of the property *subcomponent*. It could

be assumed that weakening it to an association might solve the problem. It is, however, correct that now, one component could be used within several networks. Difficulties arise, however, upon the definition of channels. Due to that fact that the network *integrator* is used twice in *doubleIntegrator*, no differentiation can be made between the ports of the two *integrator* components thus rendering a connection of the channels as well as the definition of the network impossible. In order to achieve that, individual nodes will have to be created for each use (instance) of the *integrator* component. This is not only valid for the components used but also the ports thereof. This results in the introduction of the two new concepts *ComponentInstance* and *PortInstance* in abstract syntax.

It could be assumed that channels would now connect PortInstances instead of ports. In the present case, this is, however, not sufficient as channels should also be able to connect ports of the network to be defined. The individual ports doe, however, not yet have PortInstances as the network to be defined is not yet in use. The resulting extended abstract syntax is shown in Figure 7.12.

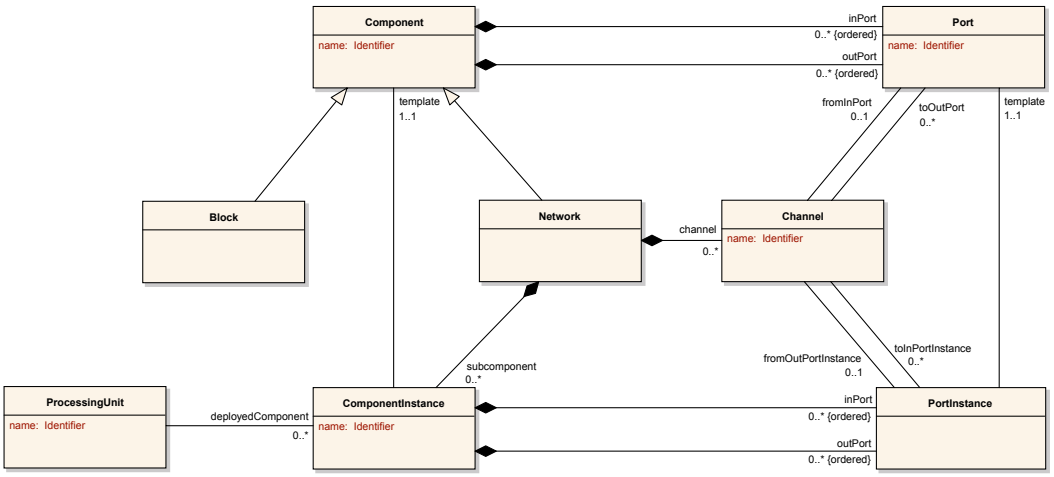


Figure 7.12.: Dataflow networks with component instantiation

Despite the extensions introduced, not all problems have been solved yet. The extension of the abstract syntax made it possible for the individual components used including the ports thereof to be distinguishable within the network. If the component used is, however, a network itself, the instances thereof will generally remain undistinguishable. The two instances of the multiplication block in the two *integrator* instances would, in our example, not be distinguishable, for example. The model would only comprise one *ComponentInstance* node for both. The reason therefore is that the instances of a network itself are not instantiated themselves. Figure 7.13 illustrates the situation when networks are instantiated again. The result will be instances of instances.

In order to define the deployment in our running example, the distinguishability of all instances is necessary. Plainly speaking, each *instance* is assigned to one computing node. In the present example, one multiplication could be assigned to one computing node, whereas another multiplication could be assigned to another computing node. One solution for this problem would be the introduction of additional instance concepts in analogy to *ComponentInstance* and *PortInstance*.

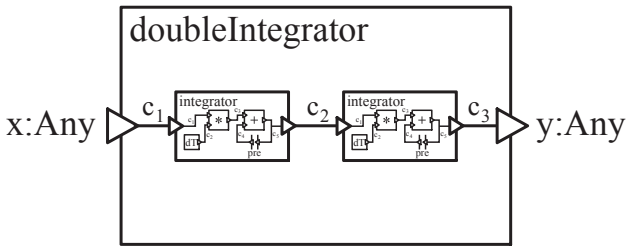


Figure 7.13.: Dataflow network including instances of instances

Another approach would be that instances will only be uniquely referenced by the decomposition path thereof along the networks. The two multiplication blocks in our example could then be distinguished by $integrator_1.mult$ and $integrator_2.mult$, wherein $integrator_1$ and $integrator_2$ denote the two instances in *doubleIntegrator*. A corresponding extension of abstract syntax would be as follows:

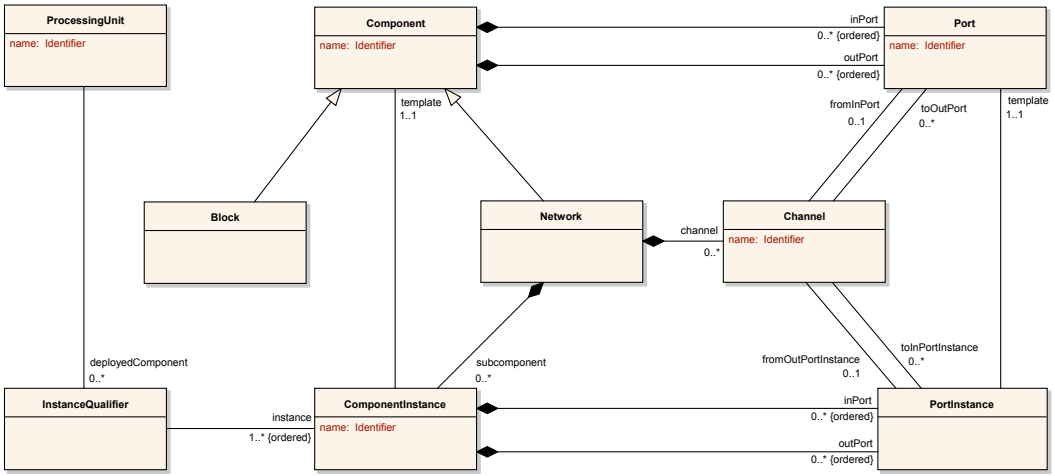


Figure 7.14.: Dataflow networks with component instantiation and instance qualifier

This results in a definition of abstract syntax which has significantly increased as regards complexity in comparison to the one originally introduced. The actual concepts of the language mix with those additional ones that have become technically necessary. Moreover would it be necessary to define many additional consistency conditions beyond the presented abstract syntax in order to obtain reasonable models. The PortInstances of a NetworkInstance would, for example, always have to correspond to the ports of the respective network.

A specific instantiation construct for re-use in metamodelling shall be introduced in the following. The aim thereof is that the introduction of re-use into the modelling language must not be extended by additional concepts introduced due to technical reasons, and that re-use will explicitly be characterised as such within the model.

The example mentioned above shows that upon instantiation it is generally required to provide individual nodes for the instances within the model. If this is strictly conducted,

all instances can be referenced uniquely. Accordingly, this is not only valid for ports and channels but also for the instances of instances discussed. This will lead to the central strategy for the instantiation concept. Instances of concepts will be realised in the model as specific clones. In contrast to the conventional approach presented above, the instance will receive the same concept as its template. The relationship between instance and template will be realised via the property *template*, which is reserved for this.

$$\begin{array}{l} \mathrm{pDef}(v_c, v_p).P_{\omega 2}{:}linkType \Subset C_{\omega 2}{:}Instantiation \\ \quad \blacktriangleright\!\blacktriangleright \\ (\circlearrowleft \Subset C_\omega{:}c) \Rightarrow (P_\omega{:}p \Subset P_\omega{:}composite \\ \quad \wedge |P_\omega{:}p.P_\omega{:}template| = |P_\omega{:}p|) \end{array} \tag{7.41}$$

In the following, specific conditions will be required for the property *template* which control the relationship between instance and template thereof in detail.

One template at most

One node must comprise one template at most. If $|P_\omega{:}template| = 1$ holds for one node, this is an instance node or just an instance. If $|\leftrightarrows P_\omega{:}template| \geq 1$ holds for one node, this is a template node or just a template.

$$|P_\omega{:}template| \leq 1 \tag{7.42}$$

Cycle-freeness of instantiations

The template must neither directly or indirectly be an instance of itself. Note that instances of instances will be allowed as long as they do not contain a cycle.

$$\circlearrowleft \notin P_\omega{:}template.^\wedge P_\omega{:}template \tag{7.43}$$

Instance keys

When looking at our example, it will be recognised that the cloning of all properties will even result in an identical local key. When two instances are present in one network – as described in our example – this will lead to duplicate local keys in one network and thus to a constraint violation. Because of that, a so-called *instance key* must additionally be defined for each instance. If a node is an instance, the instance key will always be the local key. This special situation has already been taken into account. Please refer to (7.33). The following equations will be required in addition.

$$\begin{array}{l} \mathrm{singleton?}\, 1P_\omega{:}ikey \\ \wedge\, \mathrm{set?}[2;*]P_\omega{:}ikey \\ \wedge\, |P_\omega{:}template| = 0 \Rightarrow |P_\omega{:}ikey| = 0 \\ \wedge\, |P_\omega{:}template| = 1 \Rightarrow P_\omega{:}lkey = P_\omega{:}ikey \end{array} \tag{7.44}$$

Same concept for template and instance

The concept of each instance node has to correspond exactly to the concept of the template node. Sub- and super-concepts are forbidden as well.

$$|P_\omega{:}template| = 1 \Rightarrow \forall C_\omega : c : (\circlearrowleft \Subset c \Leftrightarrow P_\omega{:}template \Subset c) \tag{7.45}$$

Instantiation of all composed nodes

If two nodes are in a template relationship, all composed nodes thereof need to be in a template relationship as well. This implies the requirement of an instantiation of exactly all composed nodes

$$\begin{aligned} &|P_\omega{:}template| = 1 \Rightarrow \\ &\quad (P_\omega{:}template.P_\omega{:}composite = P_\omega{:}composite.P_\omega{:}template) \end{aligned} \tag{7.46}$$

Edges remain within the instance area

The previous rule defines which nodes will be instantiated and where the instantiation will end. An instance range defines this set of nodes that is connected via the property *composite* and which are part of the same instance. The instance range is therefore defined as the set of all transitively reachable children along with the set of all transitively reachable parents which still belong to the instance (i. e. having a property *template*).

Edges within the template range need to be within the instance range, correspondingly. Edges leaving the template range will thus also leave the instance range and direct towards the corresponding identical node.

$$\begin{aligned} &|P_\omega{:}template| = 1 \Rightarrow (\\ &\quad \forall P_\omega : p \setminus \{lkey, ikey, ckey, composite, template\} : \\ &\qquad (P_\omega{:}template.p = p.1(((\circlearrowleft \downarrow (\leftrightarrows p. \\ &\qquad \hat{}((P_\omega{:}composite \uplus \leftrightarrows P_\omega{:}composite) \downarrow \sigma\, |P_\omega{:}template|) \\ &\qquad)).P_\omega{:}template) \oplus \circlearrowleft)) \\ &\quad) \end{aligned} \tag{7.47}$$

7.6. Running Example

Abstract syntaxes in the form of metamodels have been introduced in the present chapter. Thus, abstract syntax can now be described by a special kind of model. The structure of this model could already be defined as an Edge Algebra statement. Now the meaning and thus the structural semantics of a metamodel are also known. This shall be shown by defining the concept *Network* from our running example. The definition in the form of an Edge Algebra statement will be provided in (7.48). The abstract word representing exactly the same definition in the form of a metamodel will be shown in Figure 7.15.

$$
\begin{aligned}
&\circlearrowleft \Subset C{:}Network \Rightarrow (\\
&\quad \circlearrowleft \Subset C{:}Component\\
&\quad \wedge \leftrightarrows P{:}composite \neq \varnothing\\
&\\
&\quad \wedge P{:}subcomponent \Subset P{:}composite\\
&\quad \wedge |P{:}subcomponent.P{:}template| = |P{:}subcomponent|\\
&\quad \wedge \text{set?}\ P{:}subcomponent\\
&\quad \wedge P{:}subcomponent \Subset P{:}includedContext.P{:}componentDef\\
&\\
&\quad \wedge P{:}channel \Subset P{:}composite\\
&\quad \wedge \text{set?}\ P{:}channel\\
&\quad \wedge P{:}channel \Subset C{:}Channel\\
&)
\end{aligned}
\tag{7.48}
$$

Figure 7.15.: Example: specification of the concept *Network* in the form of an abstract word

7.7. Defining M2L – Step 3: Relationship between Meta-Metamodel and Edge Algebra

As has been illustrated in Section 7.6, *Running Example*, p. 150, it is now possible to define abstract languages by abstract syntaxes in the form of a metamodel. This leads to the third step of defining the metamodelling language *M2L* as shown in Figure 7.16.

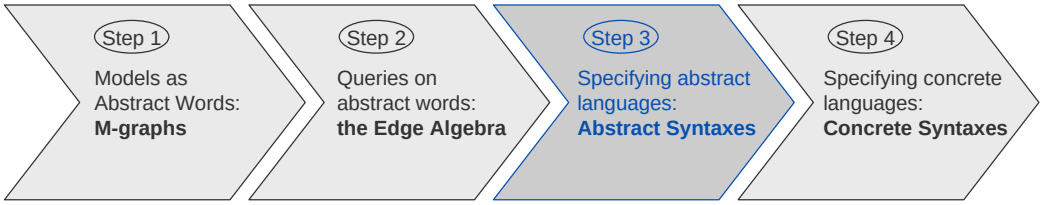


Figure 7.16.: Overview - Third of the four steps of specifying M2L.

It has been possible to specify both the M-graph representing the meta-metamodel in terms of an abstract word (see Chapter 5, *Models as Abstract Words*, p. 81) as well as the Edge Algebra statements defining the structure of the abstract language of *M2L* (see Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95) in the last chapters.

In Section 6.5, *Defining M2L – Step 2: M2L defined by Edge Algebra statements*, p. 117 it has already been discussed that it was also possible to check the consistency of the M-graph to abstract language. Now it is possible to additionally define the second relationship in a formal way: The definition of abstract language in terms of Edge Algebra statements can be derived from the abstract word representing the meta-metamodel.

Up to now, the total semantics of the metamodelling language *M2L* has not been defined. The part of textual concrete syntaxes is still open. Thus, it is currently not possible to write down the abstract syntax definition in a sensitive way. As can be seen in Figure 7.15 the M-graph becomes very large and complex even for a simple example. This is the reason why the final part of *M2L* shall be defined to be able to define the metamodel in a textual way.

Chapter 8

Textual Concrete Syntaxes in M2L

The structural design and the concepts of a language can be perfectly described by abstract syntax. Nevertheless is abstract syntax not sufficient for humans when working with real-life models. The present examples Figure 5.13 in Section 5.6, *Running Example*, p. 92, and Figure 7.15 in Section 7.6, *Running Example*, p. 150 for example illustrate how large an M-graph becomes even for a small model. For a further, automated processing there are situations in which abstract syntax is not a sensitive way of representing data.

Hence, concrete syntaxes are crucial when talking about language engineering. Although the present work includes various types of concrete syntaxes, focus shall be on textual concrete syntaxes as they are the most fundamental ones: All other types of concrete syntaxes, such as diagrammatic or tabular ones, always include textual parts.

Contents

8.1. Relationship between abstract and concrete syntaxes

In the past, modelling languages have often been defined by a (textual) formal language. This textual representation then defines the primary way of representing a model in such a language. Nowadays, graphical interfaces are gaining increasing significance. This is the reason why models are nowadays often represented in different ways, such as diagrams, XML-files, tables, and others. It has to be pointed out that such a scenario does not comprise a single model representation but a number of alternative ways of representing a model. First of all shall be distinguished between practical and technical reasons:

- **Practical reasons** result from a desire to optimise processing for humans. In an entirely model-based development process, each process task describes a change of the model (to be developed). Depending on the respective process task, different representations of the model will generally be more suitable, from an objective point of view. This ranges from solely textual representations via mixed versions to mainly diagrammatic representations. Moreover, different representations are subjectively perceived to be more comprehensible by the process roles involved due to different qualifications, expectations and other background knowledge. Personal preferences regarding textual or diagrammatic representations should also be taken into consideration. Subjective preferences are, however, often considered as being unnecessary, but when it comes to acceptance upon introduction and efficiency of the development process, they play a vital role. Intentional programming [Simonyi, 1995] perceives these objective and subjective preferences of different model representations as a central development paradigm.

- **Technical reasons** in contrast do result from an optimisation for machine processing. Different components of a development environment require different, partially contradictory requirements to model representations in order to be able to define efficient algorithms. Thus, the representation of a model in the database component, for example, differs from that of the same model in a model checker component, which, in turn, differs from that in an editor component. The number of participating component increases in particular with regard to approaches of distributed and integrated development environments, thus emphasising the effect described in modern development environments. Further representations will additionally become necessary besides these inherent problems, due to heterogeneous execution and development environments. Examples therefore are different processor architectures, transmission protocols or programming languages.

Due to the above reasons is it a good idea to abstract the actual information content from concrete representations. In this context, the abstract words, which have already been introduced, shall be mentioned (see Chapter 5, *Models as Abstract Words*, p. 81). As introduced in Chapter 7, *Abstract Syntaxes in M2L*, p. 119, abstract syntaxes describe exactly the structure of such abstract words in the form of abstract languages. Accordingly, the concrete syntaxes describe the so-called concrete languages, based on abstract syntax. Decisive is that a concrete language is not solely defined by concrete syntax but only in combination with abstract syntax. This leads to the methodical reasons for separating abstract and concrete parts of a language.

- **Methodical reasons** result from the central role played by abstract syntaxes when specifying a modelling language. Abstract syntax confines itself to defining the structure and thus the concepts of a modelling language that can be described. Both, for understanding and for the formal definition of semantics, exactly these parts of the language definition are of central importance. On the one hand, naming the concepts results in a taxonomy of the respective domain which, in turn, allows for a description of the language in an intuitively understandable way without losing grip on the formal definition. On the other hand, formal semantics can already be defined based on abstract syntax without having to define a concrete syntax, as one word of the abstract language provides an exact representation of the actual information content. This systematisation of language development also facilitates the standardisation of modelling languages, which will become essential for the required integrated model basing, in order to be able to have tools of different manufacturers working together seamlessly: Assuming a canonical concrete syntax for data exchange (i. e. a concrete syntax which can canonically be derived from abstract syntax), an agreement upon a

common abstract syntax along with semantics definition is already sufficient. Based on that, each manufacturer may differentiate from others via its own concrete (textual or else diagrammatic) syntaxes.

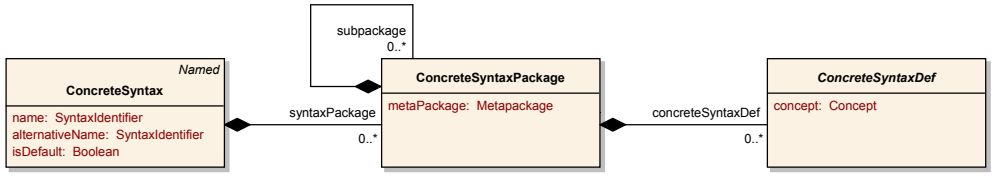
All in all, the advantages of a systematic separation of abstract and concrete syntax can be summarised as follows:

1. **Methodical advantages:** structured and integrated language development
   - Abstract syntax for simultaneously defining concepts in a formal and intuitive way
   - Abstract syntax sufficient as least common denominator for standardising the syntactical parts
2. **Practical advantages:** optimisation of syntax for processing by humans
   - Different, objectively more suitable concrete syntaxes for different process tasks
   - Different, subjectively perceived as being more comprehensible, concrete syntaxes due to different qualifications of different process roles
3. **Technical advantages:** optimisation of syntax to match machine processing
   - Inherently necessary, different concrete syntaxes due to different algorithmic requirements
   - Artificially created different concrete syntaxes due to a lack of across-tool standards

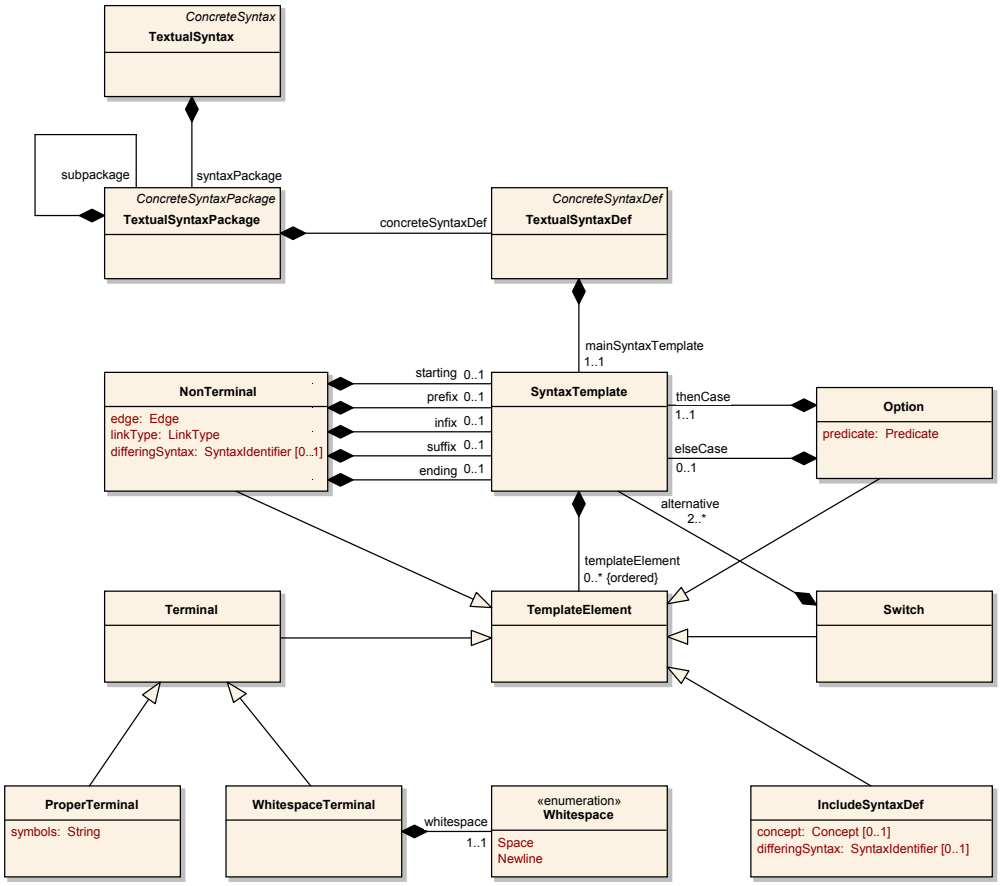
Exactly that separation of abstract and concrete parts of language does not exist in the field of formal languages. There, focus is on a concrete (textual) language from the very beginning. A kind of abstract syntax is, however, implicitly defined in the context of grammars by the application order of productions, but in contrast to the approach presented in the present thesis, it only plays a minor role.

8.2. Semi-formal introduction of the abstract syntax

As described in Section 6.5, *Defining M2L – Step 2: M2L defined by Edge Algebra statements*, p. 117, it is already possible to define the abstract syntax of the metamodelling language *M2L* in a formal way. Up to now, the semantical meaning for specifying abstract syntaxes has been defined. For that part of the metamodel which represents the concrete syntax definition the semantical meaning is still open. In order to provide a better understanding of the following sections, an overview of the relevant excerpt of the meta-metamodel will be given in advance. Abstract syntax will therefore be shown by using commonly known UML class diagrams (in particular the MOF subset is sufficient) again, as has been defined in [OMG, 2006a]. Figure 8.1 shows an excerpt of the M2L's abstract syntax which concentrates on specifying textual concrete syntaxes in M2L.



(a) General part for specifying concrete syntaxes



(b) Specific part for specifying textual concrete syntaxes

Figure 8.1.: Semi-formal abstract syntax for specifying concrete syntaxes in M2L

In the following two sections, a formal definition of those concepts will be provided. Nevertheless, a short overview shall be provided beforehand. The metamodel is divided into two parts:

The first part – shown in Figure 8.1(a) – provides the general basis for specifying concrete syntax independent of whether it is textual, diagrammatic, tabular, or even something else. In analogy to the structure of abstract syntaxes, the top-level concept for concrete syntaxes is *ConcreteSyntax*, consisting of a set of packages named *SyntaxPackage*. In contrast to the concept *AbstractSyntax*, a concrete syntax consists of a name as multiple concrete syntaxes are possible. In addition, one syntax can be marked as being a default syntax. Syntax-packages are hierarchically structured and may thus contain sub-packages. Note that the package structure must match with the package structure of abstract syntax. Within a

syntax-package the concrete syntax definitions of the concepts included are enumerated (*ConcreteSyntaxDef*). In analogy to the concept *ConceptDef*, each *ConcreteSyntaxDef* refers to a concept.

The second part – shown in Figure 8.1(b) – defines the specific abstract syntax for specifying textual concrete syntaxes. The concepts *TextualSyntax*, *TextualSyntaxPackage*, and *TextualSyntaxDef* are refining concepts of those mentioned above. They make sure that a *TextualSyntax* only consists of *TextualSyntaxDefs*. Each *TextualSyntaxDef* consists of a main *SyntaxTemplate* which consists of a list of *TemplateElements* in turn. There are five different types of template elements:

- **Terminal** represents terminal symbols. Here, it is distinguished between *ProperTerminal* and *WhitespaceTerminal*.
- **Nonterminal** represents non-terminal symbols. They consist particularly of an Edge Algebra statement directing towards the nodes from the model to be encoded.
- **Option** allows the definition of an alternative encoding depending on the valuation of a given node predicate defined as an Edge Algebra statement.
- **Switch** allows the definition of an alternative encoding independent of any condition. It can be used for defining alternative syntaxes.
- **IncludeSyntaxDef** allows a re-use of already defined syntax definitions.

8.3. Basic approach defining semantics for Concrete Syntaxes

In order to define the semantics for textual concrete syntaxes, a bidirectional mapping from a M-graph called ω to a sequence of characters and vice-versa shall be specified in a formal way.

In general, such a mapping is not unique. On the one hand there may be multiple character sequences representing the same abstract word. On the other hand a character sequence may result in more than one valid M-graph. While the former is a possibly desired mechanism as there may be multiple representations for one and the same information (e. g. “0” and “±0” both represent the same number), the latter is undesired, of course.

8.3.1. The encoding alphabet for textual concrete words

In order to represent all possible character sequences, the set of all possible concrete textual words, namely $\mathbb{C}oding$, will be introduced as follows.

$$\mathbb{C}oding =_{def} \mathcal{P}_{pomset}\left(\mathbb{C}har \cup \{ws\}\right) \tag{8.1}$$

Herein, $\mathbb{C}har$ represents the set of basic characters. It may be the set of all valid unicode characters which are not used as white-spaces. ws is a special element representing an arbitrary number (even zero) of white-space characters.

It is possible to define alternative encodings in an easy way as $\mathbb{C}oding$ is defined as a pomset: Unordered elements are seen as alternative encodings. According to that $\kappa\pi c$ returns a set of all valid character sequences by using the *path*-operator for an encoding $c \in \mathbb{C}oding$. Herein, each element ws represents an arbitrary sequence of white-space characters.

According to the special white-space character a special type of concatenation shall be introduced by way of (8.2) which additionally adds a ws between the two encodings.

$$\begin{array}{rccl} \circ: & \mathbb{C}oding^2 & \rightarrow & \mathbb{C}oding \\ & \langle A, B\rangle & \mapsto & A \circ B =_{def} A \oplus \{ws\} \oplus B \end{array} \tag{8.2}$$

8.3.2. Coding and Decoding functions

It has already been mentioned that a definition of multiple textual syntaxes is possible. Thus, the mapping must be parametrised by a syntax identifier. The concept *SyntaxIdentifier* will therefore be defined in the meta-metamodel. $\mathcal{I}d_{syn} \subseteq V_{\omega 2}$ represents all vertices of a metamodel that are marked by the concept *SyntaxIdentifier*.

$$\mathcal{I}d_{syn} =_{def} (C_{\omega 2} : SyntaxIdentifer)() \tag{8.3}$$

It will now be possible to introduce the function $\text{code}_{\omega 2, id_{syn}}$ which maps a model ω having a given metamodel $\omega 2$ and syntax identifier $\mathcal{I}d_{syn}$ to an encoding $c \in \mathbb{C}oding$.

$$\begin{array}{rccl} \text{code}: & \mathcal{M}1 \times \mathcal{M}2 \times \mathcal{I}d_{syn} & \rightarrow & \mathbb{C}oding \\ & \langle \omega, \omega 2, id_{syn}\rangle & \mapsto & \text{code}_{\omega 2, id_{syn}}(\omega) \end{array} \tag{8.4}$$

Up to now, a textual concrete word could be derived from a given abstract word. This direction from the abstract word to a textual concrete word is also known as code generation. The difference between code generation and this approach is that the resulting textual representation is more closely related to the abstract language as the major issue is not the creation of another language but the definition of a textual representation for the same language. Hence, the term *pretty-printing* instead of code generation might be more appropriate, although focus is not on providing e. g. a correct indent and/or a line break.

Besides pretty-printing, the inverse mapping to encoding is also crucial: The abstract word should be derived for a given textual concrete word. This inverse direction is commonly known by parsing a textual representation of a model. Based on (8.4), the inverse function defining the parsing in an abstract way can easily be defined.

$$\begin{array}{rccl} \text{code}^{-1}: & \mathbb{C}oding \times \mathcal{M}2 \times \mathcal{I}d_{syn} & \rightarrow & \mathcal{P}_{set}(\mathcal{M}1) \\ & \langle c, \omega 2, id_{syn}\rangle & \mapsto & \text{code}^{-1}_{\omega 2, id_{syn}}(c) \\ & & & =_{def} \left\{ \omega \in \mathcal{M}1 \;\middle|\; \pi\ \text{code}_{\omega 2, id_{syn}}(\omega) \cap \pi\, c \neq \varnothing \right\} \end{array} \tag{8.5}$$

Note that this definition neither ensures a unique parsing result nor an algorithm for parsing such a textual syntax. Nevertheless has a formal understanding of what the textual language is about been provided. Important questions about parsing algorithms, parsability, complexity of parsing, or classifying the language description in the Chomsky hierarchy are still open and are beyond the scope of the present work.

Up to now, the topic was encoding the complete abstract word into a textual representation. As this textual representation is not the primary one in the present approach but only a view for e. g. humans to provide an adequate representation, only an excerpt will be desired in most situations. A textual Java editor should not show the entire Java code within one window but only one Java class, for example. A useful definition is therefore the encoding

of one vertex (including the vertices contained therein) within an abstract word. The *vertex coding* function $\text{vcode}_{\omega 2, id_{syn}}$ will therefore be defined by adding the parameter $v \in V_{\mathcal{M}1}$ which is the vertex to be encoded.

$$\begin{array}{rlll} \text{vcode}: & V_{\mathcal{M}1} \times \mathcal{M}1 \times \mathcal{M}2 \times \mathcal{I}d_{syn} & \to & \mathbb{C}oding \\ & \langle v_\omega, \omega, \omega 2, id_{syn} \rangle & \mapsto & \text{vcode}^{\omega}_{\omega 2, id_{syn}}(v_\omega) \end{array} \tag{8.6}$$

Based on this vertex coding function it can now be defined that the encoding of the total abstract word is simply a concatenation of the encoding of all root vertices of the abstract word in an arbitrary order. Formally, it is defined as follows:

$$\begin{array}{rlll} \text{code}: & \mathcal{M}1 \times \mathcal{M}2 \times \mathcal{I}d_{syn} & \to & \mathbb{C}oding \\ & \langle \omega, \omega 2, id_{syn} \rangle & \mapsto & \text{code}_{\omega 2, id_{syn}}(\omega) \\ & & & =_{def} \tau((\sigma\,\text{root?})()\,\chi\ \text{vcode}^{\omega}_{\omega 2, id_{syn}}) \end{array} \tag{8.7}$$

8.3.3. The pomset encoding function

Before the definition of the semantics of textual concrete syntaxes as defined in *M2L* will be begun, an important helper function on encodings will be introduced. It will be relevant for encoding multi-valued properties as lists or sets. It allows an encoding of an arbitrary pomset of encodings to a single encoding such that a *starting*, a *prefix*, an *infix*, a *suffix*, and an *ending* can be specified. The meaning of those parameters in detail is:

- By starting and ending an encoding can be specified that is added at the very beginning and the very ending respectively if, and only if, the pomset consists of at least one element.
- By prefix and suffix an encoding can be specified that is added before and after each element of the pomset.
- By infix an encoding can be specified that is added inbetween of two elements each.

The formal definition is:

$$\begin{array}{rlll} (\cdot\,||\,\cdot\,|\,\cdot\,|\,\cdot\,||\,\cdot)\cdot: & \mathbb{C}oding^5 \times \mathcal{P}_{pomset}(\mathbb{C}oding) & \to & \mathbb{C}oding \\ & \langle start, pre, in, post, end, A \rangle & \mapsto & (start\,||\,pre\,|\,in\,|\,post\,||\,end)A \end{array}$$

where

$$\begin{array}{l} (start\,||\,pre\,|\,in\,|\,post\,||\,end)A =_{def} \\ \quad (A \neq \varnothing\,?\,start \oplus \{ws\}) \\ \quad \oplus\,((\text{perm}\,A)\,\chi\,(e \mapsto \{pre \circ e \circ post\})) \overrightarrow{\chi}\,\{\{ws\} \oplus in \oplus \{ws\}\}\,\chi\,(e \mapsto e)) \\ \quad \oplus\,(A \neq \varnothing\,?\{ws\} \oplus end) \end{array} \tag{8.8}$$

As can be seen, the signature of this operator is closely related to the syntactical definition of non-terminals: Non-terminal represents exactly the encoding of an arbitrary property which is generally multi-valued.

8.4. Canonical textual syntax for M-graphs

In the first stages of engineering a new language, concrete syntaxes have not been defined yet. In order to be able to validate abstract syntax in early stages, a canonical textual syntax is sensitive, however. Thus, a textual representation should be available even if no textual concrete syntax has been defined: The textual concrete syntax should be derived from abstract syntax in a canonical way. Then, the canonical textual syntax can stepwise be customised during the language engineering process if it is not yet sufficient for the final language.

Besides this methodical advantage, a canonical textual syntax also helps in representing abstract syntax in a textual way. The fundamental structure that is explicitly represented by an abstract word, will be fully preserved when encoding an abstract word by a canonical textual syntax. When, for example, a large abstract word is to be specified – such as necessary when defining the meta-metamodel in Section 5.7, *Defining* M2L – *Step 1:* M2L *Meta-Metamodel in terms of an Abstract Word*, p. 94 – the textual way will be the only one suitable.

Before introducing the formal definition of the canonical syntax, a short example shall be provided. Nodes are basically encoded hierarchically according to the *composite*-property, starting with the root vertices. A vertex itself is encoded by a list of property-value-pairs. The signature of our running example as defined in Figure 5.1(b) shall be encoded for illustration purposes. This figure will be repeated in Figure 8.2 for convenience, including the edge representing the local keys. The property *composite* is still omitted.

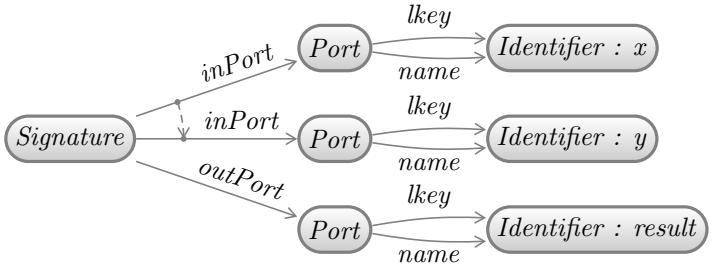


Figure 8.2.: Example: abstract word of a signature

The textual representation of the given abstract word when encoded by the canonical textual syntax should be as shown in Listing 8.1.

Listing 8.1: Signature represented by a canonical textual word

```
Signature {
  inPort: Port x {
    name: x;
  };
  inPort: Port y {
    name: y;
  };
  outPort: Port result {
    name: result;
  };
}
```

The name of the concept will be encoded at first. Then the three property-value-pairs will be placed in curly brackets. One for each outgoing edge, except for those which are labelled by the property *composite* or *lkey*: Two input ports and one output port. The ports are in turn vertices within the abstract word and should therefore be encoded the same way. There is only one edge to represent: It is labelled by the property *name*. In addition, ports comprise a local key represented by the property *lkey*. This local key is encoded in front of the curly brackets.

8.4.1. The encoding of identifiers for concepts and properties

For the canonical syntax it has been defined that the identifiers for both concepts and properties are encoded within the textual representation. As has already been discussed in Section 7.1, *Relationship between model and metamodel*, p. 119, the corresponding elements from the abstract alphabet are only mathematical symbols. In order to be able to encode these symbols, a corresponding function code_Σ has to be defined.

$$\begin{array}{rlcl} \text{code}_\Sigma: & C_\omega \cup P_\omega & \rightarrow & \mathbb{C}oding \\ & s & \mapsto & \text{code}_\Sigma(s) \end{array} \tag{8.9}$$

8.4.2. The encoding of references

Our previous example does not contain any edges which direct towards a non-compositional vertex. Naturally this is not the general case as M-graphs are not always structured as trees along the *composite*-edge. In the case the linked vertex of a property-value-pair is not directly composed to the source vertex, the vertex should be referenced by a context-sensitive key as defined in Section 7.5.5, *Referencing by context-sensitive keys*, p. 142. In connection with abstract syntax, a concrete symbol for separating local keys has not been defined yet. As in most languages, the dot character ('.') shall also be used herein. In order to mark a key as a canonical key, a dot will be added at the beginning of the key.

Given a source vertex v^s_ω, and a referenced vertex v^r_ω, the function code_{ref} will formally be defined as follows:

$$\begin{array}{rlcl} \text{code}_{\text{ref}}: & {V_{\mathcal{M}1}}^2 \times \mathcal{M}1 & \rightarrow & \mathbb{C}oding \\ & \langle v^s_\omega, v^r_\omega, \omega \rangle & \mapsto & \text{code}^\omega_{\text{ref}}(v^s_\omega, v^r_\omega) \end{array}$$

where

$$\text{code}^\omega_{\text{ref}}(v^s_\omega, v^r_\omega) =_{def} \begin{cases} (\text{'.'} \,||\, \varnothing \,|\, \text{'.'} \,|\, \varnothing \,||\, \varnothing)\, \text{ref}_{\text{cskey}}(v^s_\omega)(v^r_\omega) & \text{if } \text{ref}_{\text{cskey}}(v^s_\omega)(v^r_\omega) \neq \varnothing \\ (\varnothing \,||\, \varnothing \,|\, \text{'.'} \,|\, \varnothing \,||\, \varnothing)\, \text{ref}_{\text{ckey}}(v^r_\omega) & \text{if } \text{ref}_{\text{cskey}}(v^s_\omega)(v^r_\omega) = \varnothing \end{cases} \tag{8.10}$$

8.4.3. The canonical encoding of values

As could be seen, the encoding of the values of each property-value-pair depends on whether the value is a composite vertex or not. In case of a composition, the vertex will be encoded inline; in case of a link to a non-composite vertex, a reference by a context-sensitive or canonical key will be encoded. Formally, it is defined as follows:

$$\begin{array}{lll} \text{code}_{\text{canonic/ref}}: & {V_{\mathcal{M}1}}^2 \times \mathcal{M}1 & \rightarrow \quad \mathbb{C}oding \\ & \langle v_\omega^s, v_\omega^r, \omega\rangle & \mapsto \quad \text{code}^\omega_{\text{canonic/ref}}(v_\omega^s, v_\omega^r) \\ & & =_{def} \begin{cases} \text{vcode}^\omega_{\text{canonic}}(v_\omega^r) & \text{if } v_\omega^r \in P_\omega : composite(v_\omega^s) \\ \text{code}^\omega_{\text{ref}}(v_\omega^s, v_\omega^r) & \text{if } v_\omega^r \notin P_\omega : composite(v_\omega^s) \end{cases} \end{array} \tag{8.11}$$

8.4.4. The canonical encoding of vertices

Based on the previous definitions, it will now be possible to define the canonical encoding for vertices in total.

$$\begin{array}{l} \text{vcode}_{\text{canonic}}: \quad V_{\mathcal{M}1} \times \mathcal{M}1 \rightarrow \mathbb{C}oding \\ \qquad\qquad\quad \langle v_\omega, \omega\rangle \mapsto \text{vcode}^\omega_{\text{canonic}}(v_\omega) \\ \quad \text{where} \\ \\ \text{vcode}^\omega_{\text{canonic}}(v_\omega) =_{def} \\ \quad \text{code}_\Sigma(\text{type}_\omega(v_\omega)) \circ \text{vcode}^\omega_{\text{canonic}}((1P_\omega : lkey)(v_\omega)) \\ \quad \circ (\text{'('} \,||\, \varnothing \,|\, \text{','} \,|\, \varnothing \,||\, \text{')'})(([1]P_\omega : lkey)(v_\omega)\, \chi\, (u_\omega \mapsto \{\text{vcode}^\omega_{\text{canonic}}(u_\omega)\})) \\ \quad \circ (\text{'\{'} \,||\, \varnothing \,|\, \varnothing \,|\, \varnothing \,||\, \text{'\}'})((P_\omega \setminus lkey)\, \chi\, (p \mapsto \\ \qquad \{(\varnothing \,||\, \text{code}_\Sigma(p) \circ \text{':'} \,|\, \varnothing \,|\, \text{';'} \,||\, \varnothing)(P_\omega : p(v_\omega)\, \chi\, (u_\omega \mapsto \text{code}^\omega_{\text{canonic/ref}}(v_\omega, u_\omega)))\})) \end{array} \tag{8.12}$$

As already mentioned by this definition, no metamodel is necessary at all. It is possible to encode an abstract word without any additional information due to the semi-structured approach of M-graphs. The following section shall illustrate how to define specific textual representations.

8.5. Semantics for Concrete Syntaxes – a template-based approach

The abstract syntax for defining textual concrete syntaxes has already been introduced in Section 8.2, *Semi-formal introduction of the abstract syntax*, p. 155. Up to now, a definition was given of how to encode an abstract word without an existing definition of a textual concrete syntax. The *vcode*-function introduced in (8.6) will therefore need to be specified.

The specification has basically been realised by visiting the corresponding part of the meta-metamodel according to the *Visitor-Pattern* as defined in [Gamma et al., 1995]. The starting point for the visitor will be defined by a concept named $c \in C_{\mathcal{M}1}$. The function *conceptVisitor* will be introduced in a formal way. It encodes a given vertex v_ω from a M-graph ω according to a given concept c, a meta-model $\omega 2$ and a syntax identifier $\mathcal{I}d_{syn}$.

$$\begin{array}{lll} \text{conceptVisitor}: & C_{\mathcal{M}1} \times V_{\mathcal{M}1} \times \mathcal{M}1 \times \mathcal{M}2 \times \mathcal{I}d_{syn} & \rightarrow \quad \mathbb{C}oding \\ & \langle c_\omega, v_\omega, \omega, \omega 2, id_{syn}\rangle & \mapsto \quad \text{conceptVisior}^\omega_{\omega 2, id_{syn}}(c_\omega, v_\omega) \end{array} \tag{8.13}$$

The specification of this first part of the visitor will be defined in the following section (see Section 8.5.1, *Finding the suitable syntax definition*, p. 163). Note that the given concept

c must not be the same as the concept which labels the vertex v_ω. It may also be another concept, such as a refined concept.

Nevertheless, the starting point of the encoding function *vcode* will be defined by using the concept $c = \text{type}_\omega(v_\omega)$ which labels the vertex v_ω.

$$\begin{array}{lll} \text{vcode}: & V_{\mathcal{M}1} \times \mathcal{M}1 \times \mathcal{M}2 \times \mathcal{I}d_{syn} & \rightarrow \quad \mathbb{C}oding \\ & \langle v_\omega, \omega, \omega 2, id_{syn} \rangle & \mapsto \quad \text{vcode}^{\omega}_{\omega 2, id_{syn}}(v_\omega) \\ & & \quad =_{def} \text{conceptVisior}^{\omega}_{\omega 2, id_{syn}}(\text{type}_\omega(v_\omega), v_\omega) \end{array} \tag{8.14}$$

8.5.1. Finding the suitable syntax definition

As already defined in Section 2.1.3, *Vertical tooling aspects: the four aspects of a metamodel*, p. 26 multiple concrete syntaxes can be defined for one abstract syntax. Each concrete syntax is identified by syntax identifier out of $\mathcal{I}d_{syn}$. In addition a concrete syntax have not to be defined for each and every concept explicitly: if it is e. g. similar to the canonical syntax or the default syntax the definition can be skipped in some situations. Hence the relevant textual syntax definition for a *requested* concrete syntax given by a syntax identifier must be found first.

Once the right textual syntax definition has been found, further processing will be delegated to the function *syntaxVisitor*. It encodes a given vertex v_ω from a M-graph ω according to a given vertex $v_{\omega 2}$ from the metamodel $\omega 2$ and a syntax identifier $\mathcal{I}d_{syn}$.

$$\begin{array}{lll} \text{syntaxVisitor}: & \mathcal{M}1 \times \mathcal{M}2 \times \mathcal{I}d_{syn} \times V_{\mathcal{M}2} \times V_{\mathcal{M}1} & \rightarrow \quad \mathbb{C}oding \\ & \langle \omega, \omega 2, id_{syn}, v_{\omega 2}, v_\omega \rangle & \mapsto \quad \text{syntaxVisitor}^{\omega}_{\omega 2, id_{syn}}(v_{\omega 2}, v_\omega) \end{array} \tag{8.15}$$

The specification of this second part of the visitor will be defined in the following section (see Section 8.5.2, *Visiting the syntax definition*, p. 165).

In order to provide a flexible way of specifying concrete syntaxes, an encoding independent of whether such a syntax is defined or not will always be defined for a given syntax identifier $\mathcal{I}d_{syn}$. In case the syntax definition requested by the syntax identifier does not exist, a default syntax can be used. A syntax is marked as being a default one by the property *isDefault* in the concept *ConcreteSyntax*. If neither a default syntax exists, the already defined canonical syntax will be used instead. All in all there are nine priority levels of finding the right syntax definition:

1. **Requested syntax explicitly defined.** At first it will be tested whether the requested syntax is explicitly defined for the desired syntax type within a directly activated metamodel.
2. **Requested syntax inferred from refined metamodels.** Secondly it will be tested whether the requested syntax is explicitly defined for the desired syntax type within at least one metamodel that is refined by the given metamodel. If so, a syntax definition will be used that combines all these syntax definitions as alternatives by the concept *Switch*.
3. **Requested syntax inferred from refined concepts.** Thirdly it will be tested whether the requested syntax is explicitly defined for the desired syntax type within

at least one refined concept within a directly activated metamodel. If so, a syntax definition will be used that combines all these syntax definitions as alternatives by the concept *Switch*.

4. **Requested syntax inferred from refined concepts of refined metamodels.** Fourthly it will be tested whether the requested syntax is explicitly defined for the desired syntax type within at least one refined concept within at least one metamodel that is refined by the given metamodel. If so, a syntax definition will be used that combines all these syntax definitions as alternatives by the concept *Switch*.
5. **Default syntax explicitly defined.** At first it will be tested whether the default syntax is explicitly defined for the desired syntax type within a directly activated metamodel.
6. **Default syntax inferred from refined metamodels.** Secondly it will be tested whether the default syntax is explicitly defined for the desired syntax type within at least one metamodel that is refined by the given metamodel. If so, a syntax definition will be used that combines all these syntax definitions as alternatives by the concept *Switch*.
7. **Default syntax inferred from refined concepts.** Thirdly it will be tested whether the default syntax is explicitly defined for the desired syntax type within at least one refined concept within a directly activated metamodel. If so, a syntax definition will be used that combines all these syntax definitions as alternatives by the concept *Switch*.
8. **Default syntax inferred from refined concepts out of refined metamodels.** Fourthly it will be tested whether the default syntax is explicitly defined for the desired syntax type within at least one refined concept within at least one metamodel that is refined by the given metamodel. If so, a syntax definition will be used that combines all these syntax definitions as alternatives by the concept *Switch*.
9. **Canonical syntax.** Finally, the canonical syntax will be used if all other rules failed.

Table 8.1 summarises the nine priority levels of finding the right syntax definition:

priority	syntax	concept	metamodel
1	requested	requested	directly activated
2	requested	requested	refined
3	requested	refined	directly activated
4	requested	refined	refined
5	default	requested	directly activated
6	default	requested	refined
7	default	refined	directly activated
8	default	refined	refined
9	canonic	–	–

Table 8.1.: Nine priority levels of finding the right syntax definition

Note that there is a difference between skipping the definition of the concept and explicitly defining the concept within an *IncludeSyntaxDef* by the same concept as the present syntax definition is defined for: In case it is skipped, the syntax inclusion is polymorphic. Refined concepts will therefore use a potentially redefined syntax definition. If the concept is explicitly given, the syntax included will not be polymorphic. The formal definition of the visitor function *conceptVisitor* as defined in (8.13) is given in an algorithmic way by the following pseudo-code listing:

Listing 8.2: pseudo-code for *conceptVisitor*

$$
\begin{array}{l}
\mathbb{C}oding\ \text{conceptVisior}^{\omega}_{\omega 2,id_{syn}}(c_\omega, v_\omega)\ \{ \\
\quad \mathcal{E}_{V_{\mathcal{M}2}}\ cs := \leftrightarrows(P_{\omega 2}{:}syntaxPackage.\hat{}P_{\omega 2}{:}subpackage.P_{\omega 2}{:}concreteSyntaxDef)\ ; \\
\quad \mathcal{P}_{set}\,(V_{\mathcal{M}2})\ tsd_{reqSyntax} := \sigma(id_{syn} \Subset cs.P_{\omega 2}{:}lkey) \downarrow C_{\omega 2}{:}TextualSyntaxDef\ ; \\
\quad \mathcal{P}_{set}\,(V_{\mathcal{M}2})\ tsd_{defSyntax} := \sigma(bool)(cs.P_{\omega 2}{:}default) \downarrow C_{\omega 2}{:}TextualSyntaxDef\ ; \\
\quad \\
\quad \mathcal{E}_{V_{\mathcal{M}2}}\ c_{req} := C_{\omega 2}{:}ConceptDef \downarrow \sigma(P_{\omega 2}{:}concept \Subset cov(c_\omega))\ ; \\
\quad \mathcal{E}_{V_{\mathcal{M}2}}\ c_{equiv} := C_{\omega 2}{:}ConceptDef \downarrow \sigma(P_{\omega 2}{:}qualifiedName = n.P_{\omega 2}{:}qualifiedName)\ ; \\
\quad \mathcal{P}_{set}\,(V_{\mathcal{M}2})\ tsd_{reqConcept} := (c_{req}.\leftrightarrows P_{\omega 2}{:}conceptDef) \downarrow C_{\omega 2}{:}TextualSyntaxDef\ ; \\
\quad \mathcal{P}_{pomset}\,(V_{\mathcal{M}2})\ tsd_{refConcept} := (c_{req}.\hat{}(P_{\omega 2}{:}refines.c_{equiv}).\leftrightarrows P_{\omega 2}{:}conceptDef) \\
\qquad \downarrow C_{\omega 2}{:}TextualSyntaxDef\ ; \\
\quad \\
\quad \mathcal{P}_{set}\,(V_{\mathcal{M}2})\ tsd_{actMM} := (C_{\omega 2}{:}MetamodelFolder.\mu\hat{}P_{\omega 2}{:}composite) \\
\qquad \downarrow C_{\omega 2}{:}TextualSyntaxDef\ ; \\
\quad \mathcal{P}_{pomset}\,(V_{\mathcal{M}2})\ tsd_{refMM} := (C_{\omega 2}{:}MetamodelFolder.P_{\omega 2}{:}activeMetamodel \\
\qquad .\mu\hat{}P_{\omega 2}{:}composite) \downarrow C_{\omega 2}{:}TextualSyntaxDef\ ; \\
\quad \\
\quad \mathcal{P}_{set}\,(V_{\mathcal{M}2})\ tsd := tsd_{reqConcept} \downarrow tsd_{actMM} \downarrow tsd_{reqSyntax}\ ; \\
\quad \text{if}\ \ (tsd = \varnothing)\ \ tsd := 1(tsd_{refMM} \downarrow tsd_{reqConcept} \downarrow tsd_{reqSyntax})\ ; \\
\quad \text{if}\ \ (tsd = \varnothing)\ \ tsd := 1(tsd_{refConcept} \downarrow tsd_{actMM} \downarrow tsd_{reqSyntax})\ ; \\
\quad \text{if}\ \ (tsd = \varnothing)\ \ tsd := 1(tsd_{refConcept} \downarrow tsd_{refMM} \downarrow tsd_{reqSyntax})\ ; \\
\quad \text{if}\ \ (tsd = \varnothing)\ \ tsd := tsd_{reqConcept} \downarrow tsd_{actMM} \downarrow tsd_{defSyntax}\ ; \\
\quad \text{if}\ \ (tsd = \varnothing)\ \ tsd := 1(tsd_{refMM} \downarrow tsd_{reqConcept} \downarrow tsd_{defSyntax})\ ; \\
\quad \text{if}\ \ (tsd = \varnothing)\ \ tsd := 1(tsd_{refConcept} \downarrow tsd_{actMM} \downarrow tsd_{defSyntax})\ ; \\
\quad \text{if}\ \ (tsd = \varnothing)\ \ tsd := 1(tsd_{refConcept} \downarrow tsd_{refMM} \downarrow tsd_{defSyntax})\ ; \\
\quad \\
\quad \text{if}\ \ (tsd = \varnothing) \\
\quad\ \ \text{return}\ \ \text{vcode}^{\omega}_{\text{canonic}}(v_\omega)\ ; \\
\quad \text{else} \\
\quad\ \ \text{return}\ \ tsd\,\chi\,(u_{\omega 2} \mapsto \text{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_\omega))\ ; \\
\}
\end{array}
$$

8.5.2. Visiting the syntax definition

Finally, the visitor function *syntaxVisitor* as introduced in (8.15) need to be defined. This function basically defines an encoding for each of the non-abstract concepts defined in the meta-metamodel for textual syntax definitions as shown in Figure 8.1(b). In detail, the behaviour of the visitor function can be described depending on the visiting concept:

- For **TextualSyntaxDef** the visitor delegates to the *SyntaxTemplate* given by the property *mainSyntaxTemplate*.
- For **SyntaxTemplate** the visitor delegates to the given list of *TemplateElement* nodes. *TemplateElement* is an abstract concept. Its concrete refining concepts are the following six concepts *ProperTerminal*, *WhitespaceTerminal*, *NonTerminal*, *Option*, *Switch*, and *IncludeSyntaxDef*. Between each *TemplateElement* whitespaces are coded.
- For **ProperTerminal** the visitor codes the given terminal symbols.
- For **WhitespaceTerminal** the visitor does nothing since the whitespaces are already coded during the *SyntaxTemplate* is visited.

- For **NonTerminal** the visitor codes the defined value given by the property *edge*. As defined in Section 8.3.3, *The pomset encoding function*, p. 159, the properties *starting*, *prefix*, *infix*, *suffix*, and *ending* are taken into account.
- For **Option** the visitor codes either the *thenCase* or the *elseCase* depending on the evaluation of the node predicate defined by the property *predicate*.
- For **Switch** the visitor codes all alternatives by delegating to the given set of *SyntaxTemplate* nodes.
- For **IncludeSyntaxDef** the visitor delegates to the corresponding *TextualSyntaxDef*.

The formal definition for the *syntaxVisitor* is given in an algorithmic way by the following pseudo-code listing:

Listing 8.3: pseudo-code for *syntaxVisitor*

$$
\begin{array}{l}
\mathbb{C}oding\ \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(v_{\omega 2}, v_{\omega})\ \{ \\
\quad \text{if}\ \ (v_{\omega 2} \Subset C_{\omega 2} : TextualSyntaxDef) \\
\quad\quad \text{return}\ \ P_{\omega 2} : mainSyntaxTemplate(v_{\omega 2})\, \chi\, (u_{\omega 2} \mapsto \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega}))\ ; \\
\\
\quad \text{else if}\ \ (v_{\omega 2} \Subset C_{\omega 2} : SyntaxTemplate) \\
\quad\quad \text{return}\ \ P_{\omega 2} : templateElement(v_{\omega 2}) \\
\quad\quad\quad\quad \chi\, (u_{\omega 2} \mapsto \{\mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega})\})\ \overrightarrow{\chi}\ \{\{ws\}\}\, \chi\, (e \mapsto e)\ ; \\
\\
\quad \text{else if}\ \ (v_{\omega 2} \Subset C_{\omega 2} : ProperTerminal) \\
\quad\quad \text{return}\ \ P_{\omega 2} : symbols(v_{\omega 2})\ ; \\
\\
\quad \text{else if}\ \ (v_{\omega 2} \Subset C_{\omega 2} : WhitespaceTerminal) \\
\quad\quad \text{return}\ \ \varnothing\ ; \\
\\
\quad \text{else if}\ \ (v_{\omega 2} \Subset C_{\omega 2} : NonTerminal)\ \{ \\
\quad\quad \text{if}\ \ (|P_{\omega 2} : differingSyntax(v_{\omega 2})| = 1) \\
\quad\quad\quad id_{syn} := P_{\omega 2} : differingSyntax(v_{\omega 2})\ ; \\
\\
\quad\quad \mathbb{C}oding\ start := P_{\omega 2} : starting(v_{\omega 2})\, \chi\, (u_{\omega 2} \mapsto \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega}))\ ; \\
\quad\quad \mathbb{C}oding\ pre := P_{\omega 2} : prefix(v_{\omega 2})\, \chi\, (u_{\omega 2} \mapsto \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega}))\ ; \\
\quad\quad \mathbb{C}oding\ in := P_{\omega 2} : infix(v_{\omega 2})\, \chi\, (u_{\omega 2} \mapsto \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega}))\ ; \\
\quad\quad \mathbb{C}oding\ post := P_{\omega 2} : suffix(v_{\omega 2})\, \chi\, (u_{\omega 2} \mapsto \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega}))\ ; \\
\quad\quad \mathbb{C}oding\ end := P_{\omega 2} : ending(v_{\omega 2})\, \chi\, (u_{\omega 2} \mapsto \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega}))\ ; \\
\\
\quad\quad (V_{\mathcal{M}1} \to \mathbb{C}oding)\ f\ ; \\
\quad\quad \text{if}\ \ ((bool) P_{\omega 2} : reference(v_{\omega 2})) \\
\quad\quad\quad f := (u_{\mathcal{M}1} \mapsto \mathrm{code}^{\omega}_{\mathrm{ref}}(v_{\omega}, u_{\mathcal{M}1}))\ ; \\
\quad\quad \text{else} \\
\quad\quad\quad f := (u_{\mathcal{M}1} \mapsto \mathrm{code}^{\omega}_{\omega 2,id_{syn}}(u_{\mathcal{M}1}))\ ; \\
\\
\quad\quad \text{return}\ \ (start \,||\, pre \,|\, in \,|\, post \,||\, end)([[P_{\omega 2} : edge(v_{\omega 2})]](v_{\omega})\, \chi\, f)\ ; \\
\quad \} \\
\\
\quad \text{else if}\ \ (v_{\omega 2} \Subset C_{\omega 2} : Option)\ \{ \\
\quad\quad \text{if}\ \ ([[P_{\omega 2} : predicate(v_{\omega 2})]](v_{\omega})) \\
\quad\quad\quad \text{return}\ \ P_{\omega 2} : thenCase(v_{\omega 2})\, \chi\, (u_{\omega 2} \mapsto \mathrm{syntaxVisitor}^{\omega}_{\omega 2,id_{syn}}(u_{\omega 2}, v_{\omega}))\ ; \\
\quad\quad \text{else}
\end{array}
$$

```
38        return P_ω2 : elseCase(v_ω2) χ (u_ω2 ↦ syntaxVisitor^ω_ω2,id_syn(u_ω2, v_ω)) ;
39    }
40
41    else if (v_ω2 ∈ C_ω2 : Switch) {
42      return P_ω2 : alternative(v_ω2) χ (u_ω2 ↦ syntaxVisitor^ω_ω2,id_syn(u_ω2, v_ω)) ;
43    }
44
45    else if (v_ω2 ∈ C_ω2 : IncludeSyntaxDef) {
46      if (|P_ω2 : differingSyntax(v_ω2)| = 1)
47        id_syn := P_ω2 : differingSyntax(v_ω2) ;
48      if (|P_ω2 : conceptDef(v_ω2)| = 1)
49        return conceptVisior^ω_ω2,id_syn(P_ω2 : conceptDef.P_ω2 : concept(v_ω2), v_ω) ;
50      else
51        return code^ω_ω2,id_syn(v_ω) ;
52    }
53  }
```

8.6. Running Example

The canonical textual syntax for a signature from our running example has already been illustrated. The abstract syntax is shown in Figure 8.2. The canonical textual syntax is shown in Listing 8.1.

Having defined the semantics for specific textual syntax definitions, it is now possible to define a textual syntax for our running example. The exemplary signature should be encoded as illustrated in Listing 8.4.

Listing 8.4: Signature represented by a textual word

```
1 (x:Any, y:Any -> result:Any)
```

The textual syntax definition will therefore be defined in Figure 8.3 as an abstract word according to the meta-metamodel for textual concrete syntaxes as introduced in Section 8.2, *Semi-formal introduction of the abstract syntax*, p. 155.

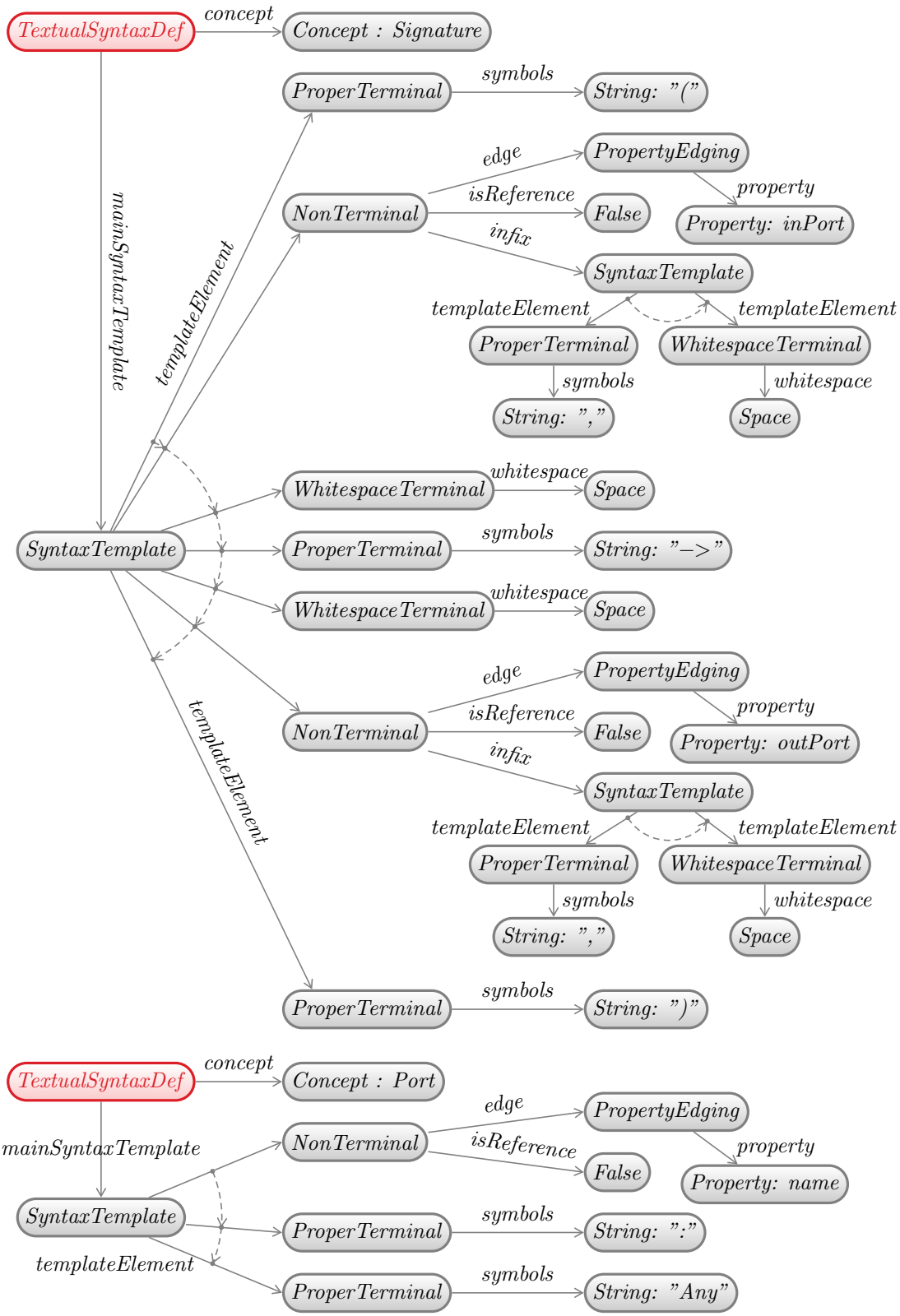


Figure 8.3.: Example: specification of the concrete syntax for the concepts *Signature* and *Port* in the form of an abstract word

8.7. Defining M2L – Step 4: M2L finally defined by M2L itself

As has been illustrated in Section 8.6, *Running Example*, p. 167, it is now possible to define specific textual languages by textual concrete syntaxes in the form of a metamodel. This leads to the fourth and final step of defining the metamodelling language *M2L* as shown in Figure 8.4.

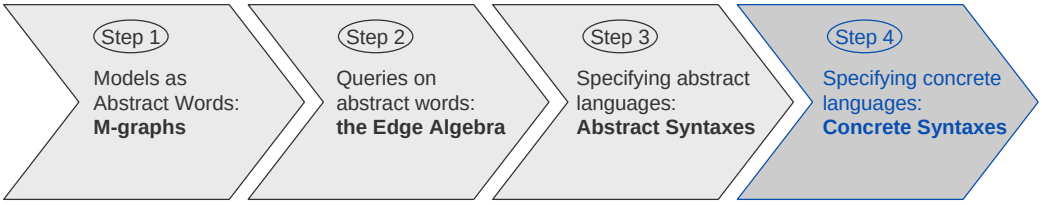


Figure 8.4.: Overview - Last of the four steps of specifying M2L.

Up to now, a metamodel always had to be defined in terms of an abstract word and thus a M-graph. As can be seen in Figure 7.15 and Figure 8.3, such M-graphs increase in size very quickly. It is now possible to define a specific textual syntax for the metamodelling language *M2L* itself. Strictly speaking, the textual syntax definition is already included in the abstract word representing the meta-metamodel of *M2L* in Section 5.7, *Defining* M2L – *Step 1:* M2L *Meta-Metamodel in terms of an Abstract Word*, p. 94. Only the semantics of the corresponding part of the meta-metamodel has still been missing.

As a textual syntax definition for metamodels is now available, and the meta-metamodel is just a specific metamodel, it is also possible to encode the meta-metamodel, which is given as an abstract word, in its specific textual representation. This last step results in the final definition of the metamodelling language *M2L*. The meta-metamodel thereof, represented by its textual syntax, is entirely defined in Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291. In the next chapter, both the abstract and the textual concrete syntax of the entire meta-metamodel will be described in detail.

Due to the self-describing mechanism, the difficulty of reading the meta-metamodel is that the language must already be known in order to be able to learn it. From a formal point of view no problem will occur as the abstract word can be described in terms of a M-graph. The textual representation can be understood along with the given semantical definitions.

The meta-metamodel described by an M-graph will, however, remain unreadable for humans because of its size. (This was also the reason why an explicit representation of the entire meta-metamodel as an M-graph was skipped in Section 5.7, *Defining* M2L – *Step 1:* M2L *Meta-Metamodel in terms of an Abstract Word*, p. 94.) Thus, a small introduction into the concrete syntax of *M2L* shall now be provided before entering the next chapter. Our running example will therefore be used again. The textual syntax of signatures and ports is shown in a textual way in Listing 8.5. It represents exactly that abstract word given in Figure 8.3 in a textual way.

Listing 8.5: Concrete syntax definition for the concepts *Signature* and *Port*

```
1 Signature: "(" (P:inPort / "," _) _ "->" _
2   (P:outPort / "," _) ")" ;
3 Port: (P:name) ":" "Any" ;
```

As can be seen, the *TemplateElements* are directly concatenated, just separated by a space. *ProperTerminals* are quoted strings. *WhitespaceTerminals* are represented by an underscore. *NonTerminals* are always denoted in round brackets. The *infix* is separated by a slash.

Additionally, abstract syntaxes can now be specified in a textual way as defined in Chapter 7, *Abstract Syntaxes in M2L*, p. 119. Listing 8.6, for example, shows the abstract syntax of the concepts *Signature* and *Port*.

Listing 8.6: Abstract syntax definition for the concepts *Signature* and *Port*

```
1 Signature! ::>
2   inPort[0..*](List) : C:Port,
3   outPort[0..*](List) : C:Port ;
4
5 Port! refines Named ::>
6   isInPort := (edge)¬empty? ⇆P:inPort,
7   isOutPort := (edge)¬empty? ⇆P:outPort ;
```

The exclamation mark at the end of the concepts' names denotes that these concepts are *weak*. The character sequence "::>" denotes that such concepts are not complete. (The character sequence "::=" in contrast would denote that a concept is complete.) Then the property definitions follow. A multiplicity can be defined after the property name. Afterwards, a pomset type can be given. The domain is specified after the colon. "C:" herein denotes a concept edging, so just the allowed concept for the corresponding property will be defined.

The concept *Port* refines the concept *Named*. Both properties are inferred, which is denoted by ":=". An Edge Algebra statement follows on the right side.

Finally, a look shall be taken at a second definition for an abstract syntax. The concept *Network* has been defined in terms of an abstract word in Figure 7.15 in Section 7.6, *Running Example*, p. 150. The corresponding textual representation is given in Listing 8.7.

Listing 8.7: Abstract syntax definition for the concept *Network*

```
1 Network! refines Component ::>
2   %subcomponent[0..*](Set) :
3     P:includedContext.P:componentDef,
4   channel[0..*](Set) : C:Channel ;
```

Besides all that has already been described, a "%" can be seen at the beginning of the definition for the property *subcomponent*. It denotes that this property is an instantiating property.

After this brief introduction, all details concerning the metamodelling language *M2L* and the meta-metamodel thereof shall be provided in the following chapter.

Chapter 9

The overall specification of M2L

The metamodelling language *M2L* has above been defined step by step in a formal way as described in Section 2.4, *Procedure specifying the (self-describing) metamodelling language* M2L , p. 42. Due to the self-describing nature of a meta-metamodel, this formal definition is not suited to show the overall picture of *M2L*: Not before the entire metamodelling language *M2L* has been specified can *M2L* be used for describing the language by itself in terms of a metamodel (i. e. the meta-metamodel). As the formal specification of *M2L* has been conducted, the overall picture of *M2L* shall now be provided in this chapter by describing both the abstract and concrete syntax of each concept introduced in a bootstrapping way.

Besides a specification of all syntactical details of *M2L* by providing the present complete definition (i. e. every defined concept will be enumerated herein), both a realistic proof-of-concept and a substantial real-live example of how to specify the syntactical part of a modelling language shall be provided as well.

Contents

All in all, the concepts are structured in nine packages and sub-packages which will be described in the following sections successively. Due to namespace conditions all packages begin with *ORG.Metamodels*. The complete meta-metamodel within one listing can be found in Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291. All (partial) listings below are parts taken from this complete listing. The line numbers given will match in order to find the original location more easily.

Structure of the following sections. Each section presents one package by describing all concepts included. One paragraph is divided into four parts for each concept: After an introduction (part 1), which describes the major aspects of the concept, the formal abstract syntax (part 2) in *M2L* including a detailed description of each property will be given. Then, textual concrete syntax will normally be specified (part 3). As the formal definition has already been given in the previous chapters, references to these chapters will be given in many cases instead of repeating the definitions. Nevertheless will a short description of each concept always be given which allows for the use of the present chapter as a reference guide, too. As can be seen in line 402 in Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291, the syntax identifier is *M2L/Text* and is marked as the default syntax as well. Concrete textual syntax will not be defined in the scope of *M2L* for three of the basic concepts, namely (*Natural*, *Character*, and *String*). Alternative syntaxes will be defined for two of these concepts: Namely they are *RAW* (for the concept *Character*), and *QUOTED* (for the concept *String*). Finally, an example (part 4) of the corresponding concept will be given as a textual concrete word. If different syntaxes are defined, all encodings will be given.

9.1. Package *ORG.Metamodels.BasicConcepts*

This package defines all basic concepts, such as boolean values, character strings, and natural numbers. This very limited set of concepts is necessary for the metamodelling language itself, but it can, of course, also be used when defining arbitrary other metamodels. This package also embodies one of the central characteristics of the underlying theory: M-graphs are not based on any fixed set of basic types.

Table 9.1 shows the list of all concepts defined:

Concept	Description
Any	the concept, which is implicitly refined by every other concept
Boolean	boolean value
Natural	natural number including zero
Interval	interval on natural number
Character	single unicode character
String	sequence of characters
Identifier	string with at least one character
Named	arbitrary concept which has a name
Folder	folder to create a directory structure
FolderEntry	entry of a folder in a directory structure

Table 9.1.: List of concepts defined in *ORG.Metamodels.BasicConcepts*

9.1.1. Concept *Any*

The concept *Any* is implicitly refined by each concept. Thus, the constraints herein must hold for each node. The concept *Any* is marked by the keyword *anyconcept* which ensures that there is only one any-concept. Please refer to Section 9.3.6, *Concept* AnyConcept-Def, p. 196 as well.

In particular, the any-concept defines all predefined properties, namely **lkey** (local key), **ikey** (instance key), **ckey** (canonical key), **composite** (marking all composed nodes), **template**

(marking the template of an instance), and **context** (defining an additional context relevant for creating context-sensitive keys).

Abstract Syntax. The formal abstract syntax for *Any* is defined in Listing 9.1.

Listing 9.1: Abstract Syntax for *Any*

```
6  anyconcept [Any] ::>
7    lkey(Poset) : C:Identifier ,
8    ikey(Poset) : C:Identifier ,
9    ckey(List) := ⩾^⇆P:composite.1 P:lkey ,
10   &composite(Set),
11   &template[0..1],
12   &includedContext(Poset)
13   where |1 P:lkey| ≤ 1 ∧ |1 P:ikey| ≤ 1
14     ∧ ⇆(* ⇵* P:composite) ≠ ⇆P:template ⇒ |P:lkey| ≥ 1
15     ∧ poset? (σroot?.P:lkey)
16     ∧ poset? (^(P:composite ↓ σ|P:lkey| = 0).P:lkey)
17     ∧ |P:template| = 0 ⇒ |P:ikey| = 0
18     ∧ |P:template| = 1 ⇒ P:lkey = P:ikey
19     ∧ |⇆P:composite| ≤ 1
20     ∧ ↻ ∉ P:composite.^P:composite
21     ∧ ↻ ∉ P:template.^P:template
22     ∧ (|P:template| = 1 ⇒ (
23       P:template.P:composite = P:composite.P:template
24       ∧ ∀C:c:(↻ ∈ c ⇔ P:template ∈ c)
25       ∧ ∀P:p\{lkey, ckey, ikey}:
26         (P:template.p = p.1 (((↻ ↓ (⇆p.
27         ^((P:composite ⊎ ⇆P:composite) ↓ σ|P:template|)
28         )).P:template) ⊕ ↻))
29     )) ;
```

The properties of the concept *Any* are:

- **lkey:** local keys of a node. Even more than one local key is generally allowed. Due to the first part of the constraint in line 13, the partially ordered set of local keys must comprise exactly one first (i. e. smallest) element if there is more than one local key. This first local key will be used as default key during encoding (see Chapter 8, *Textual Concrete Syntaxes in M2L*, p. 153). Local keys must be locally unique which means that firstly, all root nodes have unique local keys (line 15), and secondly, that all keys of a node's composite nodes must be unique (line 16). If a composite does not have any local key, the composites thereof will again be included in this uniqueness criterion. If a node is referenced by a property other than *template* which is not in parallel to *composite*, at least one local key must be defined (according to constraint in line 14).
- **ikey:** instance key of a node. This property is empty if the present node is not an instance node (which will be the case when the *template*-property is not empty, see constraint in line 17). According to the constraint in line 18, the local key equals the instance key if a node is an instance node. Just as the local key, also the instance key must comprise one single smallest element (second part of line 13).
- **ckey:** canonical key of a node. If a node does not have a local key, this property will define the prefix of the canonical keys for composite nodes. The canonical key is defined as a list of all default local keys for the nodes located on the composition

path from the root node to the given node. According to the local key constraints, a canonical key is globally unique if a node has a local key.

- **composite:** references the set of all nodes which are *part of* the referencing node according to the composition relationship. A node must have one parent (line 19) at most and compositions are generally acyclic (line 20).
- **template:** references the template if the node is an instance node. Hence, an instance node is identified by having a template. A maximum of one template is allowed. This template-instance relationship is acyclic (line 21). The constraint given in lines 22 to 28 formally defines a template relationship. Informally speaking, it states that all composite nodes are recursively cloned and linked by the property *template* for an instance node. Please refer to Chapter 7, *Abstract Syntaxes in M2L*, p. 119 for further details.
- **context:** additional context for a node which is relevant when creating context-sensitive keys. Please refer to Chapter 7, *Abstract Syntaxes in M2L*, p. 119 for further details.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *Any*.

Example. As no concrete syntax is defined for this abstract concept, please refer to all other concepts.

9.1.2. Concept *Boolean*

Boolean represents a boolean value which may be *True* or *False* and which is defined by an enumeration concept allowing these two values.

Abstract Syntax. The formal abstract syntax for *Boolean* is defined in Listing 9.2.

Listing 9.2: Abstract Syntax for *Boolean*

```
31 enum Boolean = { True, False } ;
```

The enumeration values of the concept *Boolean* are:

- **True:** represents the truth-value *true*.
- **False:** represents the truth-value *false*.

Concrete Syntax. The formal textual concrete syntax for *Boolean* is defined in Listing 9.3.

Listing 9.3: Textual Concrete Syntax for *Boolean*

```
414 Boolean.True: ("⊤" OR "true") ;
415 Boolean.False: ("⊥" OR "false") ;
```

Example. An example for the concept *Boolean* will be provided in the following Listing 9.4:

Listing 9.4: Example for *Boolean*

```
true
```

9.1.3. Concept *Natural*

Natural represents a natural number including zero. The number is represented by modelling its predecessor. If no predecessor is modelled, the number zero will be represented.

Abstract Syntax. The formal abstract syntax for *Natural* is defined in Listing 9.5.

Listing 9.5: Abstract Syntax for *Natural*

```
32 Natural!! ::= pred[0..1] : C:Natural ;
```

The properties of the concept *Natural* are:

- **pred:** predecessor of the number. If not set, zero will be represented.

By using Edge Algebra, the encoded value can easily be determined by 9.1:

$$(number)\ \circlearrowleft = |P{:}pred.\hat{}P{:}pred| \tag{9.1}$$

Note that this abstract syntax is closely related to the inductive definition of natural numbers. Many other representations are possible as the theory itself abstracts from concrete encoding.

Concrete Syntax. *Natural* is one of three concepts wherein the formal textual concrete syntax thereof will be defined beyond the theory. Thus, an EBNF statement will be given in Listing 9.6 instead of an *M2L* statement defining textual concrete syntax.

Listing 9.6: Textual Concrete Syntax for *Natural* as EBNF

```
Natural = '0' | ('1'..'9' {'0'..'9'}) ;
```

Note that in contrast to *M2L* this EBNF statement does not define the mapping from the textual concrete to the abstract word in a formal way. Informally spoken, the number will be encoded in the decimal system.

Example. An example for the concept *Natural* will be provided in the following Listing 9.7:

Listing 9.7: Example for *Natural*

```
1017
```

9.1.4. Concept *Interval*

Interval represents a (potentially unlimited) interval of natural numbers by defining a lower and an (optional) upper limit. The limits themselves are included in the interval. An *Interval* will in particular be used to model multiplicities within property definitions (Section 9.3.10, *Concept* PropertyDef, p. 201).

Abstract Syntax. The formal abstract syntax for *Interval* is defined in Listing 9.8.

Listing 9.8: Abstract Syntax for *Interval*

```
33 Interval!! ::=
34   lower[1..1] : C:Natural,
35   upper[0..1] : C:Natural
36   where |P:upper| = 1 ⇒
37     (number)P:lower ≤ (number)P:upper ;
```

The properties of the concept *Interval* are:

- **lower:** lower limit of the interval. It must always be set.
- **upper:** upper limit of the interval. If the upper limit is not set, the upper bound is unrestricted. If set, the upper limit must always be greater than or equal to the lower limit.

Concrete Syntax. The formal textual concrete syntax for *Interval* is defined in Listing 9.9.

Listing 9.9: Textual Concrete Syntax for *Interval*

```
416 Interval: (P:lower) ".."
417   (|P:upper| = 0 ? "*" : (P:upper)) ;
```

The star ("*") denotes that the interval has no upper limit.

Example. An example for the concept *Interval* including all natural numbers beginning with 2 will be provided in the following Listing 9.10:

Listing 9.10: Example for *Interval*

```
2..*
```

9.1.5. Concept *Character*

Character represents an arbitrary unicode character. Available characters are defined by the Unicode Consortium [Unicode, 2010]. *Character* will in particular be used to model strings as a sequence of characters (Section 9.1.6, *Concept* String, p. 177).

Abstract Syntax. The formal abstract syntax for *Character* is defined in Listing 9.11.

Listing 9.11: Abstract Syntax for *Character*

```
38 Character!! ::= unicode[1..1] : C:Natural
39   where (number)unicode ≤ 1114111 ;
```

The properties of the concept *Character* are:

- **unicode:** code point of the modelled unicode character. Note that the defined encodings such as UTF8 or UTF16 are irrelevant in this context. At this level, the code point is defined as a natural number out of the valid unicode code space ranging between 0 and 0x10FFFF (=1114111).

Concrete Syntax. *Character* is one of three concepts wherein the formal textual concrete syntax thereof will be defined beyond *M2L*. Two different concrete syntaxes will be defined for *Character*:

- The default syntax *M2L/Text* encodes the character within single quotes (').
- The *RAW* syntax encodes the character simply by itself. This encoding should only be used within a multiplicity of fixed size. Otherwise parsing will not be possible. An exception is the concept String wherein the concrete textual syntax thereof will be defined beyond *M2L* as well.

According to this definition, two EBNF statements for *Character* are given in Listing 9.12.

Listing 9.12: Textual Concrete Syntax for *Character* in EBNF

```
Character = "'" Character_RAW "'" ;
Character_RAW = ? arbitrary unicode character ? ;
```

Example. An example for the concept *Character* encoding the mathematical symbol for *less-or-equal* (code point *U2264*) will be provided in the following Listing 9.13:

Listing 9.13: Example for *Character*

```
'≤' // coding with default syntax M2L/Text
≤   // coding with syntax RAW
```

9.1.6. Concept *String*

String represents an arbitrary, potentially empty sequence of unicode characters. As this concept is not marked as being complete, it could contain additional information, such as what the language of the string is, which should normally be done by refining the concept.

Abstract Syntax. The formal abstract syntax for *String* is defined in Listing 9.14.

Listing 9.14: Abstract Syntax for *String*

```
40 String!! ::> character[0..*](List) : C:Character ;
```

The properties of the concept *String* are:

- **character:** a (totally ordered) sequence of characters building the string. As a list is a totally ordered multiset, a string can of course contain duplicate characters.

Concrete Syntax. *String* is one of three concepts wherein the formal textual concrete syntax thereof will be defined beyond *M2L*. Three different concrete syntaxes will be defined for *String*:

- The default syntax *M2L/Text* encodes the string within double quotes (see syntax *QUOTED*), and if it contains special characters (including white-spaces), it starts with a number or is an empty string. Here, every character besides *'A'* to *'Z'*, *'a'* to *'z'*, *'_'*, and *'0'* to *'9'* is a special character. Otherwise it is encoded without quotes. This syntax is normally used for encoding identifiers in modelling languages. It allows

you to use special characters within identifiers but omits the quotes if a usual identifier is given.

- The syntax *QUOTED* always encodes the string within double quotes ("). The quotes themselves as well as the backslash (\) are escaped by a preceding backslash (thus \" and \\).

According to this definition, two EBNF statements for *String* are given in Listing 9.15.

Listing 9.15: Textual Concrete Syntax for *String* in EBNF

```
String = String_QUOTED | (('A'..'Z' | 'a'..'z' | '_')
         { 'A'..'Z' | 'a'..'z' | '_' | '0'..'9' }) ;
String_QUOTED = '"' { Character_RAW - ('"' | '\')
                | '\\' | '\"' } '"' ;
```

Example. Two examples for the concept *String* will be provided in the following Listing 9.16. The first one does not contain any special characters, whereas the second one does:

Listing 9.16: Example for *String*

```
StringWithoutSpecialChar    // coding with default syntax M2L/Text
"StringWithoutSpecialChar"  // coding with syntax QUOTED
"String including Spaces"   // equal coding for both syntaxes
                            // M2L/Text and QUOTED
```

9.1.7. Concept *Identifier*

Identifier represents a refinement of the concept *String* excluding empty strings. This concept is used for identifiers as empty strings are undesired in this context (see also Section 9.1.8, *Concept* Named, p. 179).

Abstract Syntax. The formal abstract syntax for *Identifier* is defined in Listing 9.17.

Listing 9.17: Abstract Syntax for *Identifier*

```
41 Identifier!! refines String ::> character[1..*] ;
```

The properties of the concept *Identifier* are:

- **character (refined):** characters of the string. As an *Identifier* must not be empty, the lower limit of the multiplicity is more restrictive and set to 1.

Concrete Syntax. Concrete syntax is inherited from *String* (see Section 9.1.6, *Concept* String, p. 177).

Example. As the concrete syntax is the same as for *String*, refer to Section 9.1.6, *Concept* String, p. 177 for examples.

9.1.8. Concept *Named*

Named represents an arbitrary concept which has a (primary) name and a set of additional alternative names. Due to the key constraint, both the primary name and all alternative names must be locally unique (see Section 7.5.5, *Local keys, namespaces and visibility*, p. 137). When encoding references within non-terminals the primary name will be used. During a decoding of non-terminal references the alternative names can be dereferenced as well.

Alternative names are helpful in many situations:

- **Synonyms.** Multiple terms are established for one and the same item (represented by a node in our context) in many situations. Alternative names can be used in order to be able to reference this node by all possible terms. The term having preference upon use is set as the (primary) name. This approach allows for establishing a well-defined set of terms. In particular the concept *Concept* (see Section 9.3.5, *Concept* ConceptDef, p. 192) itself is a good example: In most cases there are various names for one concept which should be modelled explicitly.
- **Abbreviations.** Another use-case for alternative names are abbreviations. The abbreviations can be set as an alternative name in such situations. During editing, for example, these abbreviations can be used instead of the full names which may significantly increase usability.
- **Multi-language support.** Specifying names in different languages is a third situation where alternative names are useful. In this case, the English term (or whatever the primary language may be) may be set as the (primary) name. All other languages will be defined within the alternative names. Note that *String* and *Identifier* do not specify a language identifier. A special refining concept of *String* may therefore be useful.

In case alternative names are not allowed, the concept *UniquelyNamed* can be used.

Abstract Syntax. The formal abstract syntax for *Named* is defined in Listing 9.18.

Listing 9.18: Abstract Syntax for *Named*

```
43 [Named] ::>
44   (PK)name[1..1] : C:Identifier ,
45   (K)alternativeName[0..*](Set) : C:Identifier ;
```

The properties of the concept *Named* are:

- **name:** primary name of a node. Exactly one name must always be set.
- **alternativeName:** (unordered) set of alternative names. These additional names are optional. In case alternative names are not allowed, the concept *UniquelyNamed* can be used.

Concrete Syntax. The formal textual concrete syntax for *Named* is defined in Listing 9.19.

Listing 9.19: Textual Concrete Syntax for *Named*

```
419 Named: (P:name)
420   ("(" || (P:alternativeName) / "," _ || ")") ;
```

Although this concept is abstract, a concrete textual syntax will be defined in order to include this definition within refining concepts. The primary name will therefore be simply encoded at first, followed by a comma-separated list of alternative names within parentheses.

Example. An example for the concept *Named*, although it is abstract, having two alternative names besides the primary name will be provided in the following Listing 9.20 :

Listing 9.20: Example for *Named*

```
"phone number" (fon, Telefonnummer)
```

Note that by default the names are encoded by the default string encoding (thus, quotes are only used if special characters are included). If all names shall be quoted, the syntax *QUOTED* can be used.

9.1.9. Concept *UniquelyNamed*

UniquelyNamed represents a refinement of the concept *Named* which forbids alternative names.

Abstract Syntax. The formal abstract syntax for *UniquelyNamed* is defined in Listing 9.21.

Listing 9.21: Abstract Syntax for *UniquelyNamed*

```
46 [UniquelyNamed] refines Named ::>
47   alternativeName[0..0] ;
```

The properties of the concept *UniquelyNamed* are:

- **alternativeName (refined):** alternative names for a node. As alternative names are not allowed for the concept *UniquelyNamed*, multiplicity is set to *0..0*.

Concrete Syntax. Concrete syntax is inherited from *Named* (see Section 9.1.8, *Concept* Named, p. 179). As the alternative names are always empty, only the primary name will be encoded.

Example. An example for the concept *UniquelyNamed*, although it is abstract, having only a primary name (as required) will be provided in the following Listing 9.22:

Listing 9.22: Example for *UniquelyNamed*

```
"phone number"
```

9.1.10. Concept *Folder*

In order to be able to structure the content of a repository, the concept *Folder* will be provided. On the one hand, a folder may contain subfolders building a directory tree. On the other hand, a folder may reference a set of folder entries (*FolderEntry*). Note that these folder entries are not composed to the folder. This results in a number of characteristics:

- It is not sufficient to reference *weak* folder entries by a folder as this reference is not a composition.
- The canonical name of a folder entry is not influenced by a concrete location within the folder tree.
- Two folder entries that are not composed to other nodes must have different names – even if they are located in different directories.
- In general, folder entries can be referenced from more than one folder. A primary reference does not exist. All references are equal.
- As *Folder refines FolderEntry*, a folder can be referenced by other folders as well. Thus, in contrast to other folder entries, folders have a primary reference (described by the *subfolder* composition).

Abstract Syntax. The formal abstract syntax for *Folder* is defined in Listing 9.23.

Listing 9.23: Abstract Syntax for *Folder*

```
48 Folder refines FolderEntry ::>
49   subfolder[0..*](Toset) : C:Folder,
50   &entry[0..*](Toset) : C:FolderEntry ;
```

The properties of the concept *Folder* are:

- **subfolder:** totally ordered set of subfolders. Due to this property, the folder tree is spanned. Thus, duplicates and cycles are not allowed.
- **entry:** totally ordered set of folder entries. The folder entries are not composed to the folder but are only referenced. Folders themselves can, in particular, also be referenced herein. Concepts refining *Folder* does typically constrain the property *entry*stronger in order to restrict the allowed set of concepts. Note that this can be done recursively or not. An example will be *MetamodelFolder* (see Section 9.2.1, *Concept* MetamodelFolder, p. 183).

Concrete Syntax. The formal textual concrete syntax for *Folder* is defined in Listing 9.24.

Listing 9.24: Textual Concrete Syntax for *Folder*

```
421 Folder: "folder" _ <Named> _ "{"
422   (nl | P:subfolder) (nl | &P:entry) nl "}" ;
```

Example. An example for the concept *Folder* will be provided in the following Listing 9.25:

Listing 9.25: Example for *Folder*

```
folder "A demonstration folder" (demoFolder) {
  folder "A subfolder" {
    entry M2L.ORG.Metamodels.BasicConcepts.String;
  }
  entry M2L;
  entry M2L.ORG.Metamodels.BasicConcepts;
}
```

The folder is named "A demonstration folder" and has an alternative name, namely "demoFolder". It contains a subfolder named "A subfolder" which references the concept *String*. In addition, the main folder references the metamodel "M2L" and the metapackage "M2L.ORG.Metamodels.BasicConcepts".

9.1.11. Concept *FolderEntry*

FolderEntry represents all (possible) entries which may be referenced within folders. Thus, all concepts wherein the nodes thereof should be able to be referenced by folders have to refine *FolderEntry*. Besides the fact that a folder entry must have a name, nothing particular is required by such concepts.

Abstract Syntax. The formal abstract syntax for *FolderEntry* is defined in Listing 9.26.

Listing 9.26: Abstract Syntax for *FolderEntry*

```
51 [FolderEntry] refines Named ;
```

No additional properties will be defined for the concept *FolderEntry*.

Concrete Syntax. Concrete syntax is inherited from *Named* (see Section 9.1.8, *Concept* Named, p. 179).

Example. As the concrete syntax is the same as for *Named*, refer to Section 9.1.8, *Concept* Named, p. 179 for examples.

9.2. Package *ORG.Metamodels.M2L*

This package defines the core concepts of the metamodelling language *M2L*. It is structured into two sub-packages for abstract as well as concrete syntax. As these sub-packages will be described by the subsequent sections, this section contains just three concepts, which are listed in Table 9.2:

Concept	Description
MetamodelFolder	the root folder for active metamodels
Metamodel	describes a metamodel including abstract and concrete syntax
Metametamodel	the metamodel which describes the metamodeling language itself

Table 9.2.: List of concepts defined in *ORG.Metamodels.M2L*

9.2.1. Concept *MetamodelFolder*

A model repository may contain a set of metamodels (which configure the repository themselves by adding constraints defined by these metamodels). As not every metamodel stored within a repository should generally also influence the behaviour of that repository, it is distinguished between active and inactive metamodels. Only active metamodels configure the repository. In order to distinguish between active and inactive metamodels, the metamodel folder will be introduced. Exactly one metamodel folder is allowed. It references all active metamodels. The metamodels may be referenced indirectly as the metamodel folder may contain normal folders referencing the metamodels. Metamodels that are not contained within the metamodel folder are inactive.

An important use-case is the internet platform *METAMODELS.org* itself: It provides a collection of various metamodels of different fields of application. Anyway, the only active metamodel should be the meta-metamodel (i. e. the metamodel for *M2L*) itself, as metamodels shall only be stored within this repository. The metamodel folder will therefore only contain the meta-metamodel in the present case. All other metamodels will be stored in a different folder structure.

Abstract Syntax. The formal abstract syntax for *MetamodelFolder* is defined in Listing 9.27.

Listing 9.27: Abstract Syntax for *MetamodelFolder*

```
55 MetamodelFolder refines Folder ::>
56   name := {{ Identifier("Active Metamodels") }},
57   &entry : C:Metamodel,
58   &activeMetamodel := μ^P:subfolder.P:entry.^P:basedOn
59   where |C:MetamodelFolder| = 1
60     ∧ ^P:subfolder.P:entry ∈ C:Metamodel
61     ∧ C:Metametamodel ∈ P:activeMetamodel ;
```

The properties of the concept *MetamodelFolder* are:

- **name (refined):** the name of the metamodel folder must be set to "Active Metamodels".
- **entry (refined):** the entry must only reference metamodels. Note that the additional constraint will be transitively defined for all subfolders. Thus, even all sub-folders must only reference metamodels.

- **activeMetamodel:** this inferred property specifies all active metamodels, i. e. those metamodels which are relevant for the validity of a model. Active metamodels are those that are directly stored within the metamodel folder or within one of the sub-folders thereof, supplemented by those metamodels that are transitively included (see property *basedOn* in Section 9.2.2, *Concept* Metamodel, p. 184). The partial order of this inferred property reflects the dependencies according to the property *basedOn*. Metamodels that are directly stored within the metamodel folder structure therefore represent the smallest elements within the partial order. Although the property *basedOn* is acyclic, the property *activeMetamodel* may contain duplicates as a metamodel may form the basis for more than one metamodel. (Hence the property *activeMetamodel* may be a real pomset in general.) Note that the meta-metamodel (see Section 9.2.3, *Concept* Metametamodel, p. 186) must always be active.

Concrete Syntax. The formal textual concrete syntax for *MetamodelFolder* is defined in Listing 9.28.

Listing 9.28: Textual Concrete Syntax for *MetamodelFolder*

```
426 MetamodelFolder: "metamodel" _ <Folder> ;
```

The metamodel folder is encoded in exactly the same way as normal folders, except for the prefix *metamodel folder* instead of *folder*.

Example. An example for the concept *MetamodelFolder* only containing the meta-metamodel within two sub-folders will be provided in the following Listing 9.29:

Listing 9.29: Example for *MetamodelFolder*

```
metamodel folder "Active Metamodels" {
  folder "Modeling Languages" {
    folder "Language Engineering" {
      entry "Metamodeling Language M2L";
    }
  }
}
```

9.2.2. Concept *Metamodel*

Literally, a metamodel is the model behind models - thus describing a modelling language. Abstract and concrete syntaxes are currently supported in *M2L*. According to that, a metamodel contains an abstract syntax and in general a set of concrete syntaxes which are based on the abstract syntax. The other two language aspects, namely *process definition* and *semantics*, are not supported yet.

Metamodels may, in general, *base on* other metamodels. This relationship between metamodels must not be cyclic: Two metamodels must not be based on each other (not even transitively). Due to this mechanism, a large metamodel can be split into a set of smaller metamodels. Note that besides this strong coupling mechanism between metamodels there is also a loose coupling by concept name equivalence based on the canonical key (see also Section 9.3.9, *Concept* ExternalConceptDef, p. 200).

Abstract Syntax. The formal abstract syntax for *Metamodel* is defined in Listing 9.30.

Listing 9.30: Abstract Syntax for *Metamodel*

```
62 Metamodel refines FolderEntry ::>
63   &basedOn[0..*](Set) : C:Metamodel ⇣* ^⇆P:basedOn,
64    abstractSyntax[1..1] : C:AbstractSyntax,
65    concreteSyntax[0..*] : C:ConcreteSyntax,
66   &exportedMetapackage :=
67     P:abstractSyntax.P:exportedMetapackage,
68   &visibleMetapackage :=
69     (↻ ⊕ P:basedOn).P:exportedMetapackage ;
```

The properties of the concept *Metamodel* are:

- **basedOn:** set of metamodels this metamodel is based on. Metamodels that are based on this metamodel must not be included as a cyclic dependency is not allowed.
- **abstractSyntax:** the abstract syntax definition of this metamodel within which all concepts of this metamodel will be defined. A metamodel must have exactly one abstract syntax.
- **concreteSyntax:** set of concrete syntax definitions of this metamodel. Multiple concrete syntaxes are possible and even quite usual. In principle, different kinds of concrete syntax definitions are possible. Nevertheless, *M2L* only defines a specification technique for textual concrete syntaxes. Concrete syntax definitions may also be omitted completely. In this case, the canonical syntaxes will be used instead.
- **exportedMetapackage:** references all meta-packages which are exported by this metamodel. Exported packages are those which can be referenced by other metamodels based on this metamodel. This inferred property states that all meta-packages defined within this metamodel will be exported. Packages taken from metamodels this metamodel is based on, will not be exported. Thus, *M2L* has no re-export mechanism.
- **visibleMetapackage:** references all meta-packages which are visible within this metamodel. Thus, concepts from these meta-packages can be referenced. This inferred property states that all exported meta-packages of the metamodels this metamodel is based on and all meta-packages defined within this metamodel will be visible. Note that meta-packages will not be exported transitively.

Concrete Syntax. The formal textual concrete syntax for *Metamodel* is defined in Listing 9.31.

Listing 9.31: Textual Concrete Syntax for *Metamodel*

```
427 Metamodel: "metamodel" _ <Named>
428   (nl "based" _ "on" _ || &P:basedOn / "," _) "{"
429   (nl | P:abstractSyntax) (nl | P:concreteSyntax)
430   nl "}" ;
```

Example. An example for the concept *Metamodel* will be provided in the following Listing 9.32:

Listing 9.32: Example for *Metamodel*

```
metamodel "A demonstration metamodel" (demoMetamodel)
  based on "Metamodeling Language M2L" {
  abstract syntax {
    ... // the details will be omitted here
  }
  textual default concrete syntax "The default syntax" {
    ... // the details will be omitted here
  }
  textual concrete syntax "An additional syntax" {
    ... // the details will be omitted here
  }
}
```

The metamodel is named "A demonstration metamodel" and has an alternative name, namely "demoMetamodel". It is based on the metamodel "Metamodelling Language M2L". Besides the abstract syntax it contains two textual concrete syntaxes. The details will be omitted in this example as they have been described in Section 9.3, *Package* ORG.Metamodels.M2L.AbstractSyntax, p. 188 and Section 9.4, *Package* ORG.Metamodels.M2L.ConcreteSyntax, p. 210.

9.2.3. Concept *Metametamodel*

The concept *Metametamodel* refines the concept *Metamodel* in order to mark the metamodel describing the metamodelling language (in particular *M2L*) itself. The meta-metamodel is characterized in that the syntax thereof (both abstract as well as concrete syntax) is conform to itself. As all metamodels must be conform to the same meta-metamodel, exactly one meta-metamodel is allowed. Note that within this Chapter 9, *The overall specification of M2L*, p. 171 the meta-metamodel is exactly described.

Abstract Syntax. The formal abstract syntax for *Metametamodel* is defined in Listing 9.33.

Listing 9.33: Abstract Syntax for *Metametamodel*

```
70 Metametamodel refines Metamodel ::>
71   name := {{ Identifier("Metamodelling Language M2L"),
72     Identifier(M2L) }},
73   &basedOn[0..0],
74   where |C:Metametamodel| = 1 ;
```

The properties of the concept *Metametamodel* are:

- **name (refined):** the name of the meta-metamodel must be equal to "Metamodelling Language M2L".
- **basedOn (refined):** the meta-metamodel must not be based on other metamodels.

Concrete Syntax. The formal textual concrete syntax for *Metametamodel* is defined in Listing 9.34.

Listing 9.34: Textual Concrete Syntax for *Metametamodel*

```
431 Metametamodel: "meta–" <Metamodel> ;
```

The meta-metamodel is encoded in exactly the same way as normal metamodels, except for the prefix *meta-metamodel* instead of *metamodel*.

Example. An example for the concept *Metametamodel* will be provided in the following Listing 9.35:

Listing 9.35: Example for *Metametamodel*

```
meta–metamodel "Metamodelling Language M2L" (M2L) {
  abstract syntax {
    ... // the details will be omitted here
  }
  textual default concrete syntax "M2L/Text" {
    ... // the details will be omitted here
  }
}
```

Note that this example corresponds to the listing from Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291.

9.3. Package *ORG.Metamodels.M2L.AbstractSyntax*

This package defines the concepts necessary for describing the abstract syntax of a modelling language. An abstract syntax generally defines an *abstract language*, thus the modelling language independent of a concrete syntactical notation. It concentrates on the underlying *concepts* of the modelling language and the relationships thereof among each other (*properties*).

Formally, a model at this abstract level (*abstract word*) is defined by a special kind of directed labelled multi-graph, called M-graph (see Chapter 5, *Models as Abstract Words*, p. 81): The nodes are labelled by concepts, whereas the edges are labelled by properties. Based on this, an abstract language is a (potentially infinite) set of abstract words which is valid with regard to the abstract language. In order to describe such a set of abstract words, the semantics of an abstract syntax is a set of constraints expressed as Edge Algebra statements: If, and only if, all these constraints are then fulfilled by an abstract word, it is valid with regard to the abstract language defined by abstract syntax (see Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 and Chapter 7, *Abstract Syntaxes in M2L*, p. 119).

By way of the concept *Metapackage*, a namespace for the concepts is built. The concept *Concept* itself combines all constraints which are relevant for a dedicated concept. (Formally it follows the assumption/guarantee pattern; see Chapter 7, *Abstract Syntaxes in M2L*, p. 119.) Note that constraints are always associated with a concept in *M2L*. In order to specify global constraints, the concept *ORG.Metamodels.BasicConcepts.Any* will be defined. Formally, each concept refines the concept *Any* implicitly. Besides the fact that an abstract word without any node is included in every abstract language, everything that may be expressed with the Edge Algebra may also be expressed by an abstract syntax. Constraints that focus on a dedicated *property* are defined by the concept *PropertyDef* which is also associated with a concept. All in all, the defined concepts are listed in Table 9.3:

Concept	Description
AbstractSyntax	the main concept encapsulating abstract syntax
Metapackage	hierarchical packages structuring abstract syntax
Concept	identifier for concepts representing a qualified name
Property	the properties which can be restricted by concepts
ConceptDef	the concept definition for the concept definition itself
AnyConceptDef	the concept definition representing the any-concept
EnumerationConceptDef	refinement of *ConceptDef* for enumerations
EnumElementConceptDef	refinement of *ConceptDef* for enumeration elements
ExternalConceptDef	refinement of *ConceptDef* for externally defined concepts
PropertyDef	the restrictions for a dedicated property within a concept
ConceptType	enumerates the concept types such as *Weak*
KeyType	enumerates the key types such as *PrimaryLocalkey*
LinkType	enumerates the link types such as *Composition*
PomsetType	enumerates the pomset types such as *Bag* or *List*

Table 9.3.: List of concepts defined in *ORG.Metamodels.M2L.AbstractSyntax*

9.3.1. Concept *AbstractSyntax*

The concept *AbstractSyntax* encapsulates the definition of abstract syntax within a meta-model. Structured by meta-packages, it contains all concepts defined within this abstract

syntax (i. e. indirectly via meta-packages). Note that two abstract syntaxes of different metamodels may contain the same meta-package structure even with overlapping concept names. When combining these metamodels, the conjunction of the resulting constraints must hold for the two concepts with the same name and package location (i. e. the same qualified name; see Section 9.3.5, *Concept* ConceptDef, p. 192).

Abstract Syntax. The formal abstract syntax for *AbstractSyntax* is defined in Listing 9.36.

Listing 9.36: Abstract Syntax for *AbstractSyntax*

```
77 AbstractSyntax! ::>
78   metapackage[0..*](Set) : C:Metapackage,
79   &exportedMetapackage :=
80     P:metapackage.P:exportedMetapackage,
81   &visibleMetapackage :=
82     ⇆P:composite.P:visibleMetapackage ;
```

The properties of the concept *AbstractSyntax* are:

- **metapackage:** set of all (root) meta-packages this abstract syntax consists of. Note that the transitively reachable sub-packages will not be included herein.
- **exportedMetapackage (inferred):** set of all transitively reachable meta-packages and sub-meta-packages this abstract syntax consists of. It is also the set of meta-packages that are exported by the metamodel (see Section 9.2.2, *Concept* Metamodel, p. 184).
- **visibleMetapackage (inferred):** set of meta-packages this abstract syntax can reference, and which are the same as those the metamodel can access (see Section 9.2.2, *Concept* Metamodel, p. 184).

Concrete Syntax. The formal textual concrete syntax for *AbstractSyntax* is defined in Listing 9.37.

Listing 9.37: Textual Concrete Syntax for *AbstractSyntax*

```
434 AbstractSyntax: "abstract" _ "syntax" _ "{"
435   (nl | P:metapackage) nl "}" ;
```

Example. An example for the concept *AbstractSyntax* will be provided in the following Listing 9.38:

Listing 9.38: Example for *AbstractSyntax*

```
abstract syntax {
  metapackage ORG {
    metapackage Demo {
      ...
    }
  }
  metapackage COM {
    ...
  }
```

```
}
```

The abstract syntax shown contains two top-level meta-packages, namely *ORG* and *COM*. The meta-package *ORG* additionally contains a sub-package called *Demo*. All other details will be omitted in this example.

9.3.2. Concept *Metapackage*

The package structure for the abstract syntax will be built by the concept *Metapackage*. Meta-packages may again contain meta-packages as sub-packages. The resulting tree structure forms the namespace for the concepts contained therein: Two concepts have the same *qualified name* if, and only if, they have the same name and are located at the same meta-package position (see Section 9.3.5, *Concept* ConceptDef, p. 192).

Abstract Syntax. The formal abstract syntax for *Metapackage* is defined in Listing 9.39.

Listing 9.39: Abstract Syntax for *Metapackage*

```
83 Metapackage! refines Named ::>
84   subpackage[0..*](Set) : C:Metapackage,
85   concept[0..*](Set) : C:Concept,
86   &exportedMetapackage :=
87     ↻ ⊕ (P:subpackage.P:exportedMetapackage),
88   &visibleMetapackage :=
89     ⇆P:composite.P:visibleMetapackage ;
```

The properties of the concept *Metapackage* are:

- **subpackage:** set of all sub-packages contained in this meta-package. Note that the transitively reachable sub-packages will not be included herein.
- **concept:** set of concepts defined within this meta-package.
- **exportedMetapackage (inferred):** set of all transitively reachable sub-packages including the present meta-package itself. It is also the set of meta-packages this meta-package contributes to the exported meta-packages (see Section 9.2.2, *Concept* Metamodel, p. 184).
- **visibleMetapackage (inferred):** set of meta-packages this meta-package can reference, and which are the same as those the metamodel can access (see Section 9.2.2, *Concept* Metamodel, p. 184).

Concrete Syntax. The formal textual concrete syntax for *Metapackage* is defined in Listing 9.40.

Listing 9.40: Textual Concrete Syntax for *Metapackage*

```
436 Metapackage: "metapackage" _ <Named> _ "{"
437   (nl | P:concept | _ ";") (nl | P:subpackage)
438   nl "}" ;
```

Example. An example for the concept *Metapackage* will be provided in the following Listing 9.41:

Listing 9.41: Example for *Metapackage*

```
metapackage Demo {
  "A simple concept" ::> ;
  metapackage "A sub package" {
    "Another concept" ::> ;
  }
}
```

The meta-package shown contains a sub-package named *"A sub package"*. In addition, both the top-level meta-package and the sub-package contain one concept definition. Please refer to Section 9.3.5, *Concept* ConceptDef, p. 192 for details regarding an encoding of a concept.

9.3.3. Concept *Concept*

The concept *Concept* represents an identifier for a concept. Concepts are organised by namespaces. Thus, the *qualified name* consists of a list of identifiers. Two concepts are only seen as being equal if the total list is equal.

Abstract Syntax. The formal abstract syntax for *Concept* is defined in Listing 9.42.

Listing 9.42: Abstract Syntax for *Concept*

```
91 Concept!! ::=
92   qualifiedName[2..*](List) : C:Identifier ;
```

The properties of the concept *Concept* are:

- **qualifiedName:** qualified name of the concept. It is represented by a (totally ordered) list of meta-package names concluded by the concept name. In contrast to the canonical key, it skips the metamodel name at the beginning. Note that this definition will also apply to leaf concepts defined within enumeration concepts (see Section 9.3.7, *Concept* EnumerationConceptDef, p. 197): i. e. the qualified name of the concept *True* is *ORG.Metamodels.BasicConcepts.Boolean.True*.

Concrete Syntax. The formal textual concrete syntax for *Concept* is defined in Listing 9.43.

Listing 9.43: Textual Concrete Syntax for *Concept*

```
440 Concept: (P:qualifiedName / ".") ;
```

Each element of the qualified name is separated by a dot.

Example. An example for the concept *Concept* representing the concept *Any*, as it can be found within the meta-package *ORG.Metamodels.BasicConcepts* (see Section 9.1.1, *Concept* Any, p. 172), will be provided in the following Listing 9.44:

Listing 9.44: Example for *Concept*

```
ORG.Metamodels.BasicConcepts.Any
```

Quotes will not be necessary in the present example as no special characters (or spaces) will be used within none of the identifiers.

9.3.4. Concept *Property*

The concept *Property* represents an identifier for a property. Properties are not organised by namespaces or something equivalent. Thus, two properties are seen as being equal if they have the same name. This fact deeply impacts the semantics of abstract syntaxes whenever two concept definitions are combined. This is mainly the case when a concept refines one or more other concepts or when two metamodels are combined which contain concepts having the same qualified name. In all these cases the resulting constraints of the concepts involved will be united. As properties having the same name are treated as the same property, the constraints for equal properties must be fulfilled at the same time (due to the conjunction). Note that the resulting constraint may obviously be contradictory. Please refer to Chapter 7, *Abstract Syntaxes in M2L*, p. 119 for details.

Some properties have a special meaning pre-defined by the language *M2L*. In detail they are *lkey*, *ikey*, *ckey*, *composite*, *template*, and *context*. They are defined within the concept *Any* in Section 9.1.1, *Concept* Any, p. 172.

Abstract Syntax. The formal abstract syntax for *Property* is defined in Listing 9.45.

Listing 9.45: Abstract Syntax for *Property*

```
93 Property!! refines Identifier ;
```

A property is just a refinement of a non-empty string without any additional restrictions. As abstract syntax is not further restricted, properties may also contain any kind of special characters.

Concrete Syntax. Concrete syntax is (indirectly) inherited (via *Identifier*) from *String* (see Section 9.1.6, *Concept* String, p. 177).

Example. An example for the concept *Property* representing the property *name*, as it can be found within the concept *Named* (see Section 9.1.8, *Concept* Named, p. 179), will be provided in the following Listing 9.46:

Listing 9.46: Example for *Property*

```
name
```

Quotes will not be necessary in the present example as no special characters (or spaces) will be used within the property *name*.

9.3.5. Concept *ConceptDef*

The concept *Concept* defines the definition of a concept itself. According to the present theory, the semantics of a concept definition is a set of constraints expressed as an Edge Algebra statement. In contrast to properties (see Section 9.3.4, *Concept* Property, p. 192), concepts are organised within a meta-package structure resulting in qualified concept names:

Two concepts are seen as being equal if they have the same qualified name: thus, two concepts are equal if, and only if, they have the same name and they are located within the same meta-package. Due to the lkey-constraint (see Section 7.5.5, *Local keys, namespaces and visibility*, p. 137), two concepts like these will always be located in different metamodels. Note that the qualified name is not equal to the canonical key: Whereas the canonical key must be globally unique by definition, for this reason starting with the (unique) metamodel name, the qualified name on the other hand skips the metamodel name, starts directly with the first meta-package name and is thus not globally unique. Formally, the relationship between canonical key (*ckey*) and qualified name is defined by 9.2:

$$\begin{aligned} metamodelName &= (^\wedge \leftrightarrows P{:}composite \downarrow C{:}Metamodel)\,.P{:}name \\ ckey &= metamodelName.P{:}qualifiedName \end{aligned} \tag{9.2}$$

In contrast to most other metamodel approaches *M2L* does not state what is allowed but states what is not allowed. In case of an "empty" metamodel anything will be allowed - so, every model (formally every abstract word) will be valid. By defining a concept, restrictions are imposed on every node, the type of which being set to said concept (or a concept refining said concept). This procedure leaves many questions open: When for example a concept *Person* is specified, which must have a property *name* directing towards a concept *String*, no restrictions on other properties will be imposed, such as *age*, as the property *age* has not been defined and is thus not constrained. Consequently, a model including a node with the type *Person* may indeed have a property *age* without rendering the model invalid. This approach makes it easy to define a sound concept refinement: A refining concept *AgedPerson* may, for example, additionally constrain the property *age*. As the concept *Person* states nothing about *age*, no contradiction occurs and the constraints of both concepts can be simply conjuncted in order to get the total constraints for the refining concept.

As has been mentioned, the semantics of each concept definition can be expressed as an Edge Algebra statement. As the name suggests, concept definitions make assertions concerning a dedicated concept. Thus, the constraints should only be relevant if the type of the node is of the respective concept or one of the refined concepts thereof. Said will be described within Edge Algebra by an implication as shown in 9.3

$$\begin{aligned} &\circlearrowleft \Subset C{:}\langle \textbf{qualified name of concept} \rangle \Rightarrow (\top \\ &\quad \wedge \langle \textbf{Constraint 1} \rangle \\ &\quad \wedge \langle \textbf{Constraint 2} \rangle \\ &\qquad\quad \vdots \\ &\quad \wedge \langle \textbf{Constraint n} \rangle \\ &) \end{aligned} \tag{9.3}$$

Note that the constant boolean value *true* ($\top$) at the beginning of the right side of the implication is more technical as this renders said pattern as being valid even if a concept has no constraints.

For details on how to map a concept definition into an Edge Algebra statement please refer to Chapter 7, *Abstract Syntaxes in M2L*, p. 119.

Abstract Syntax. The formal abstract syntax for *ConceptDef* is defined in Listing 9.47.

Listing 9.47: Abstract Syntax for *ConceptDef*

```
95 ConceptDef! refines Named ::>
96   concept[1..1] : C:Concept,
```

```
97  &refines[0..*](Set) :
98    (P:visibleMetapackage.P:conceptDef) ↓* ^⇆P:refines,
99  isAbstract[1..1] : C:Boolean,
100 isComplete[1..1] : C:Boolean,
101 conceptType[1..1] : C:ConceptType,
102 propertyDef[0..*](Set) ↔ conceptDef : C:PropertyDef,
103 additionalConstraint[0..1] : C:Predicate,
104 &visibleMetapackage :=
105   ⇆P:composite.P:visibleMetapackage,
106 &includedContext := P:visibleMetapackage
107 where P:concept.P:qualifiedName = (≥^⇆P:composite
108   ↓ (C:Metapackage ⊎ C:ConceptDef)).P:name ;
```

The properties of the concept *ConceptDef* are:

- **concept:** qualified name of the concept. It is represented by a (totally ordered) list of meta-package names concluded by the concept name. In contrast to the canonical key, it skips the metamodel name at the beginning.
- **refines:** set of concepts which is refined by this concept. Thus, all defined constraints for those refined concepts must also hold for the present concept. A concept may refine more than one other concept. All concepts owned by visible meta-packages can be referenced here. Concepts refining said concept are excluded in order to avoid a cyclic refinement relationship.
- **isAbstract:** marks whether the concept is abstract or not. An abstract concept must not be the type of any node. Thus, the type must be one of the refining concepts (which is not abstract).
- **isComplete:** marks whether the concept is complete or not. All properties which are not explicitly defined by a property definition in the present or in one of the refined concepts (even transitively) must result in an empty set for a complete concept. Note that this definition results from the semi-structured approach which allows to set properties that have not been defined explicitly (and are hence not constrained). Based on that definition, concepts refining complete concepts cannot add any additional properties but can, indeed, restrict existing ones stronger.
- **conceptType:** type of the concept. Please refer to Section 9.3.11, *Concept* ConceptType, p. 204 for further details.
- **propertyDef:** set of defined and thus restricted properties. Please refer to Section 9.3.10, *Concept* PropertyDef, p. 201 for further details.
- **additionalConstraint:** optional, additional constraint expressed as an Edge Algebra statement. Here, constraints can be located that cannot be expressed by the previous options. Please refer to Section 9.6, *Package* ORG.Metamodels.EdgeAlgebra, p. 233 for further details.
- **visibleMetapackage (inferred):** set of meta-packages this concept can reference. They are the same as those the parent metamodel can access (see Section 9.2.2, *Concept* Metamodel, p. 184).
- **context (inferred):** context included for context-sensitive keys. According to the given definition, all concepts contained by visible meta-packages can be accessed by the simple key as long as it is unique. For further details concerning the formal definition

and use of context included please refer to Section 7.5.5, *Local keys, namespaces and visibility*, p. 137.

Concrete Syntax. The formal textual concrete syntax for *ConceptDef* is defined in Listing 9.48.

Listing 9.48: Textual Concrete Syntax for *ConceptDef*

```
441 ConceptDef: ((bool)(P:isAbstract)
442     ? "[" <Named> (P:conceptType) "]"
443     : <Named> (P:conceptType))
444   (_ "refines" _ || &P:super / "," _) _
445   ((bool)P:isComplete ? "::=")
446   ((¬(bool)P:isComplete ? "::>")
447     || nl | P:propertyDef / ",")
448   (nl "where" _ | P:additionalConstraint) ;
```

The concrete syntax for concept definitions has been defined such that a compact language evolves which is reminiscent of a grammar definition enriched with notations from UML class diagrams. A concept definition is encoded as a kind of assignment denoted by **"::="** (in case of complete concepts) or **"::>"** (in case of concepts which are not marked as being complete). The concept name including some additional information is located on the left side, whereas property definitions and additional constraints are encoded on the right side. The greater-than-symbol for non-complete concepts should illustrate the fact that a concept may be more than the given property restrictions (as any undefined properties are allowed). If a concept is marked as being abstract, the concept name is given in squared brackets. The concept type will be encoded after the concept name. Two exclamation marks (**"!!"**) represent an attribute concept, one exclamation mark (**"!"**) represents a weak concept or none a strong concept. Please refer to Section 9.3.11, *Concept* ConceptType, p. 204 for further details. Following that approach, the refined concepts will be enumerated on the left side of the assignment, marked by the keyword **"refines"** at the beginning. All property definitions will be enumerated on the right side of the assignment. Please refer to Section 9.3.10, *Concept* PropertyDef, p. 201 for further details. The additional constraint, if any, will finally be encoded, marked by the keyword **"where"** at the beginning.

Example. An example for the concept *ConceptDef* will be provided in the following Listing 9.49:

Listing 9.49: Example for *ConceptDef*

```
[Person!] refines Named, Commentable ::>
  age[1..1] : C:Natural,
  &friend[0..*] : C:Person
  where ((number)P:age ≤ 150)
```

The given example defines a non-complete concept with the name *Person*, refining the two concepts *Named* and *Commentable*. Due to the square brackets it is marked as being abstract and the exclamation mark states that this concept is weak. Both given property definitions, namely *age* and *friend*, will be skipped here as they will be described in Section 9.3.10, *Concept* PropertyDef, p. 201. Finally, a conditional constraint can be found, stating that nobody should be older than 150.

Note that each of the definitions for abstract syntaxes in this Chapter 9, *The overall specification of M2L*, p. 171 that are no enumerations or external concepts represent further possible examples.

9.3.6. Concept *AnyConceptDef*

The concept *AnyConceptDef* is a special concept definition for the any-concept. The any-concept is that concept which is implicitly refined by every other concept. Due to this, the any-concept neither refines nor is it explicitly refined by any other concept. In order to be able to define additional properties for any other concept, the any-concept must not be marked as being complete. Besides that, the any-concept must be marked as a strong concept because at least one node of a M-graph must be a root node. This requests at least one (non-abstract) strong concept which is used as type for this root node. Nevertheless, a node should not be of the type *Any*; therefore the any-concept is marked as being abstract.

Abstract Syntax. The formal abstract syntax for *AnyConceptDef* is defined in Listing 9.50.

Listing 9.50: Abstract Syntax for *AnyConceptDef*

```
110 AnyConceptDef! refines ConceptDef ::>
111   concept :=
112     {{ Concept(ORG.Metamodels.BasicConcepts.Any) }},
113   &refines[0..0],
114   isAbstract : C:Boolean.True,
115   isComplete : C:Boolean.False,
116   conceptType : C:ConceptType.Strong
117   where |⇆P:refines| = 0 ;
```

The properties of the concept *AnyConceptDef* are:

- **qualifiedName (refined):** The qualified name of the any-concept must be *ORG.Metamodels.BasicConcepts.Any*. This also ensures that there is one any-concept within each metamodel at most. Nonetheless is it possible to add additional constraints to the any-concept by other metamodels.
- **refines (refined):** No refining concept can be defined as this concept forms the basis.
- **isAbstract (refined):** The any-concept is abstract as the type of a node should not be *Any*.
- **isComplete (refined):** The any-concept is not complete. Otherwise, no properties could be defined for any concept.
- **conceptType (refined):** The any-concept is a strong concept. Otherwise, no strong concepts would be allowed which is contradictory to the fact that at least one root node must exist within a M-graph.

Concrete Syntax. The formal textual concrete syntax for *AnyConceptDef* is defined in Listing 9.51.

Listing 9.51: Textual Concrete Syntax for *AnyConceptDef*

```
450 AnyConceptDef: "anyconcept" _ <Concept> ;
```

The any-concept is encoded as an ordinary concept definition. In order to indicate that this concept is the any-concept, it is marked by the preceding keyword **"anyconcept"**.

Example. An example for the concept *AnyConceptDef* will be provided in the following Listing 9.52:

Listing 9.52: Example for *AnyConceptDef*

```
anyconcept [Any] ;
```

The given example shows the simplest form of an any-concept without any constraints. The basic any-concept for *M2L* is defined in Section 9.1.1, *Concept* Any, p. 172.

9.3.7. Concept *EnumerationConceptDef*

EnumerationConceptDef is a special type of concept definition and thus a refinement of the concept *ConceptDef*. This concept allows for the definition of enumerations. Enumerations within this context are defined as a finite set of identifiers without any formal syntactical relation to other concepts such as natural numbers. As has been introduced in Section 9.1.2, *Concept* Boolean, p. 174, the concept *Boolean* is defined by an enumeration of the two elements *True* and *False*. Formally, the elements of an enumeration concept are in turn concepts refining the enumeration concept (called leaf concepts as they must not have any outgoing property edges, see Section 9.3.8, *Concept* EnumElementConceptDef, p. 198). The enumeration concept itself is abstract. This definition results in nodes which are marked by said leaf concepts (i. e. the values of the enumeration). It will then be possible to associate these nodes with properties requiring the corresponding enumeration concept.

Abstract Syntax. The formal abstract syntax for *EnumerationConceptDef* is defined in Listing 9.53.

Listing 9.53: Abstract Syntax for *EnumerationConceptDef*

```
118 EnumerationConceptDef! refines ConceptDef ::>
119   enumElement[2..*] : C:EnumElementConceptDef,
120   &refines[0..0],
121   isAbstract : C:Boolean.True,
122   isComplete : C:Boolean.True,
123   conceptType : C:ConceptType.Attribute,
124   propertyDef[0..0],
125   additionalConstraint[0..0] ;
```

The properties of the concept *EnumerationConceptDef* are:

- **enumerationValue:** set of (at least two) values for the present enumeration concept. As mentioned before, the enumeration values are described by leaf concepts (see Section 9.3.8, *Concept* EnumElementConceptDef, p. 198). Note that as this property is a composing one, leaf concepts are the only concepts that are composed to a concept instead of a metapackage. This is also the reason why these concepts are referenced by ⟨**name of enumeration concept**⟩.⟨**name of leaf concept**⟩: the enumeration concepts themselves are not part of the context included.
- **refines (refined):** An enumeration concept itself must not refine other concepts.

- **abstract (refined):** From a mathematical point of view, an enumeration concept is always abstract. This definition prevents that a node is marked by the enumeration concept itself which is obviously not desired, as the corresponding nodes should be marked by the refining leaf concepts representing the real enumeration values.
- **complete (refined):** Enumeration concepts are always complete. As property definitions are also not allowed, it is ensured that enumeration nodes do not have any outgoing edges.
- **conceptType (refined):** An enumeration must always be treated as an attribute concept.
- **propertyDef (refined):** An enumeration concept must not define any properties. As an enumeration concept is always marked as being complete, no outgoing edges are allowed.
- **additionalConstraint (refined):** Additional constraints cannot be defined for enumeration concepts.

Concrete Syntax. The formal textual concrete syntax for *EnumerationConceptDef* is defined in Listing 9.54.

Listing 9.54: Textual Concrete Syntax for *EnumerationConceptDef*

```
451 EnumerationConceptDef: "enum" _ <Named> (_ "=" _ "{" _
452   || P:enumElement / "," _ || _ "}") ;
```

As most properties from the more general concept *ConceptDef* are restricted to a fixed value, the concrete syntax for enumeration concepts becomes quite simple: The only properties to be encoded are concept name and enumeration values. In order to mark a concept as an enumeration concept, the keyword **enum** will be used at the beginning, followed by the name of the concept. The set of enumeration values will, connected by an equals sign, then be encoded within curly brackets.

Example. An example for the concept *EnumerationConceptDef* will be provided in the following Listing 9.55:

Listing 9.55: Example for *EnumerationConceptDef*

```
enum Boolean = { True, False }
```

This definition for boolean values has been taken from the present meta-metamodel as defined in Section 9.1.2, *Concept* Boolean, p. 174. Other examples can e. g. be found in Section 9.3.11, *Concept* ConceptType, p. 204 or Section 9.3.14, *Concept* PomsetType, p. 208.

9.3.8. Concept *EnumElementConceptDef*

EnumElementConceptDef is a special type of concept definition which is exclusively used within definitions of enumeration concepts as described in Section 9.3.7, *Concept* EnumerationConceptDef, p. 197. Leaf concepts are used to describe dedicated enumeration values. The name comes from the fact that such concepts result in leaf nodes within a M-graph, as it is forbidden for nodes being marked by leaf concepts to have any outgoing edges. Besides all other concepts, leaf concepts only occur within enumeration concepts referenced by the

property *enumerationValue*. They cannot directly be defined within a package. Obviously this is not a strong restriction as the behaviour of a leaf concept can be easily imitated by a basic concept definition as introduced in Section 9.3.5, *Concept* ConceptDef, p. 192. An example for imitating a leaf concept will be provided in the following Listing 9.56:

Listing 9.56: Imitating a leaf concept

```
Leaf!! ::= ;
```

Abstract Syntax. The formal abstract syntax for *EnumElementConceptDef* is defined in Listing 9.57.

Listing 9.57: Abstract Syntax for *EnumElementConceptDef*

```
126 EnumElementConceptDef! refines ConceptDef ::>
127   &refines := ⇆P:enumElement,
128   isAbstract : C:Boolean.False,
129   isComplete : C:Boolean.True,
130   conceptType : C:ConceptType.Attribute,
131   propertyDef[0..0],
132   additionalConstraint[0..0]
133   where ⇆P:composite ∈ C:EnumerationConceptDef ;
```

The properties of the concept *EnumElementConceptDef* are:

- **refines (refined as inferred):** A leaf concept refines exactly that enumeration concept which it defines. Note that this is always exactly one concept.
- **isAbstract (refined):** A leaf concept is not abstract. Otherwise, no node can be labelled by this concept.
- **isComplete (refined):** A leaf concept is marked as being complete although this constraint has already been defined within the refining enumeration concept. As property definitions are also not allowed, it is ensured that leaf nodes do not have any outgoing edges.
- **conceptType (refined):** A leaf concept is always treated as an attribute concept.
- **propertyDef (refined):** A leaf concept must not define any properties. As a leaf concept is always seen as being complete, no outgoing edges are allowed.
- **additionalConstraint (refined):** Additional constraints cannot be defined for enumeration concepts.

Concrete Syntax. The formal textual concrete syntax for *EnumElementConceptDef* is defined in Listing 9.58.

Listing 9.58: Textual Concrete Syntax for *EnumElementConceptDef*

```
453 EnumElementConceptDef: <Named> ;
```

As all properties from the more general concept *ConceptDef* are restricted to a fixed value and no additional properties will be added, the concrete syntax for enumeration values will have to encode the concept name only.

Example. An example for the concept *EnumElementConceptDef* for the enumeration value *True* of the concept *Boolean* will be provided in the following Listing 9.59:

Listing 9.59: Example for *EnumElementConceptDef*

```
True
```

9.3.9. Concept *ExternalConceptDef*

ExternalConceptDef is a special kind of concept definition which is tailored to metamodel modularization. As has been described in Section 9.2.2, *Concept* Metamodel, p. 184, a metamodel may depend on a set of other metamodels. Then, all concepts defined within these other metamodels can be referenced e. g. for refining them. This will, however, result in a tied relationship between those metamodels although such a strong dependency is undesired in many situations.

As described in Section 9.3.5, *Concept* ConceptDef, p. 192, two concepts of two different metamodels are seen to be equal if they have the same qualified name. In this case the constraints of both definitions will be conjuncted, which means that both constraints must hold. This enables another way of combining metamodels: Both metamodels define the same concept by different aspects. A special case evolves, if one of those two definitions only specifies a concept's existence so as to be able to reference it. Then, a concept without any restrictions should formally be defined as a placeholder for an external definition. This can be realised by external concepts. Just as leaf concepts, external concepts can also be easily imitated by a general concept definition. An example for imitating an external concept will be provided in the following Listing 9.60:

Listing 9.60: Imitating an external concept

```
SomeExternalConceptDef ::> ;
```

Nevertheless, a special language construct will be defined in order to show explicitly that a concept is defined in order to be specified externally.

Abstract Syntax. The formal abstract syntax for *ExternalConceptDef* is defined in Listing 9.61.

Listing 9.61: Abstract Syntax for *ExternalConceptDef*

```
134 ExternalConceptDef! refines ConceptDef ::>
135   &refines[0..0],
136   isAbstract : C:Boolean.False,
137   isComplete : C:Boolean.False,
138   conceptType : C:ConceptType.Strong,
139   propertyDef[0..0],
140   additionalConstraint[0..0] ;
```

The properties of the concept *ExternalConceptDef* are:

- **refines (refined):** No refining concept can be defined. Any refinement would result in undesired inherited constraints.
- **isAbstract (refined):** The concept is not abstract as an abstract concept has the constraint that a node must not be marked by this concept.

- **isComplete (refined):** The concept is not complete. As no properties have been defined, every property will be allowed.
- **conceptType (refined):** An external concept is treated as being a strong concept thus resulting in none of the containment constraints (see Section 9.3.11, *Concept* ConceptType, p. 204).
- **propertyDef (refined):** An external concept must not have any property definition as the only reason for specifying said is to restrict the concept.
- **additionalConstraint (refined):** No additional constraint is, of course, desired for external constraints.

Concrete Syntax. The formal textual concrete syntax for *ExternalConceptDef* is defined in Listing 9.62.

Listing 9.62: Textual Concrete Syntax for *ExternalConceptDef*

```
454 ExternalConceptDef: "external" _ <Named> ;
```

As all properties from the more general concept *ConceptDef* are restricted to a fixed value and no additional properties will be added, the concrete syntax for external concepts will have to encode the concept name only. In contrast to leaf concepts, external concepts are marked by a preceding keyword **external**.

Example. An example for the concept *ExternalConceptDef* will be provided in the following Listing 9.63:

Listing 9.63: Example for *ExternalConceptDef*

```
external SomeExternalConceptDef
```

The given example shows the same concept definition as given in Listing 9.60, but in a more explicit way.

9.3.10. Concept *PropertyDef*

A property definition represented by the concept *PropertyDef* is responsible for defining the constraints for a single property. It always occurs within a concept definition referenced by the property *propertyDef* (see Section 9.3.5, *Concept* ConceptDef, p. 192). From a semantic point of view, a property definition can always be translated into a set of constraints expressed as an Edge Algebra statement. According to the definition of *Concept*, these constraints are conjuncted to the overall set of constraints for a dedicated concept.

In general, more than one property definition may be defined for one and the same property. This will particularly be the case when property definitions vary with regard to different assumptions. Note that this way of modelling will result in the fact that property definitions do not have a local key constraint defined for the property *property* as a uniqueness would not allow more than one property definition for a single property. This restriction does, however, not cause any problems as the idea behind properties is that they are not hardly linked to property definitions or even to concept definitions: As has been characterized in Section 9.3.4, *Concept* Property, p. 192, properties are simply referenced by their simple

name defined by the concept *Property*. According to this strategy, the domain of the property *opposite*, which specifies a bidirectional association between concepts, will be *Property* instead of *PropertyDef*.

Abstract Syntax. The formal abstract syntax for *PropertyDef* is defined in Listing 9.64.

Listing 9.64: Abstract Syntax for *PropertyDef*

```
PropertyDef! ::>
  &conceptDef[1..1] ↔ propertyDef : C:ConceptDef,
   assumption[0..1] : C:Predicate,
   property[1..1] : C:Property,
   opposite[0..*](Set) : C:Property,
   keyType[0..1] : C:KeyType,
   linkType[1..1] : C:LinkType,
   multiplicity[0..1] : C:Interval,
   pomsetRestriction[0..1] : C:PomsetType,
   domain[0..1] : C:Edge,
   inferredValue[0..1] : C:Edge,
   &includedContext := P:conceptDef.P:includedContext ;
```

The properties of the concept *PropertyDef* are:

- **concept:** The concept this property definition is associated with. It is defined as the opposite of the property *propertyDef* of *Concept*. Note that a property definition is always modelled within the context of a concept.
- **assumption:** The condition which must hold in order for this property definition to become relevant. The assumption is defined by a predicate statement in terms of Edge Algebra (see Section 9.8.1, *Concept* Predicate, p. 249). If no assumption has been defined, this property definition will always be relevant. Formally, this assumption will result in an implication. The left side of this implication is the assumption. The right side is the resulting constraint for this property definition (without the assumption).
- **property:** The property this property definition is meant for. As multiple property definitions for the same property are allowed, this property is not marked as being a local key. Properties are always referenced by the concept *Property* instead of a reference to *PropertyDef*.
- **opposite:** set of properties which must be set in an inverse direction. This is a more generic way of specifying bidirectional associations. An example is a parent-child-relationship between persons: If one person is a child of another person, this other person is the parent of the child. This more generic specification allows for the definition of multiple opposite properties which are necessary to ensure a structure-preserving conjunction of metamodels. Note that defining an opposite property does not imply a definition of this opposite property within the foreign concept (i. e. the concept specified as the domain of this property). This is important as the domain for a property might be specified by an Edge Algebra statement which is not a pure concept edge in general.
- **keyType:** States whether this property is marked as being the local key. Generally speaking, local keys must be unique within the set of all sister nodes in terms of the composition tree spanned by the *composite*-property. Please refer to Section 7.5.5, *Local keys, namespaces and visibility*, p. 137 for a detailed and formal specification. Note

that multiple keys for different properties with assumptions that hold at the same time will normally result in a contradiction.

- **linkType:** link type for this property. Possible values are *Reference*, *Composition*, and *Instantiation*. Please refer to Section 9.3.13, *Concept* LinkType, p. 207 for details.

- **multiplicity:** If set, it defines the valid multiplicity for the property. If no multiplicity is given, no restrictions are defined which equals to [0..*].

- **pomsetRestriction:** The pomset type allowed for this property. Possible values are *Set*, *Bag*, *List*, *Toset*, *Poset*, and *Pomset*. Please refer to Section 9.3.13, *Concept* LinkType, p. 207 for details. If no pomset type is given, no restrictions are defined which equals to the value *Pomset*.

- **domain:** specifies the set of valid nodes for this property. If no domain has been specified, no constraint concerning the domain will be added. The standard case is to define the domain by a concept; thus a node is valid if its type is of this concept. Nevertheless, a domain of a property will generally be specified by an edge expression in terms of Edge Algebra (see Section 9.7.1, *Concept* Edge, p. 236). Hereupon, so-called context-sensitive domain specifications can be formulated. Please refer to Section 7.5.3, *Context-sensitive domains*, p. 134 for further details. This generic way of modelling a domain also allows for a specification of a domain defined by a set of concepts with the help of an additive union (Section 9.7.5, *Concept* MultiEdgeOperator, p. 241). Formally, a specification of a domain will result in a *consists-of* operator. Thus, order and duplicates of a domain will not influence the set of valid models.

- **inferredValue:** If set, *inferredValue* specifies a property as an inferred one. Thus, the property can be calculated by other properties (which might be inferred properties in turn). The calculation rule is given by *inferredValue* and is specified by an Edge Algebra statement. Note that inferred properties may specify a domain in addition to the inferred value. In various cases this (and other constraints such as multiplicity) may even indirectly result in additional constraints for the properties inspected by the inferred value. Please refer to Section 7.5.4, *Inferred properties*, p. 136 for further details. Although an inferred value has the characteristics of an assignment, it will formally result in an equation constraint: The property is valid if it equals to the evaluation of the edge statement defined by the inferred value. This again leads to the fact that an inferred value will be treated as any property such that the corresponding edges must be modelled explicitly as a part of the M-graph. Such properties do, however, not have to be modelled explicitly within a modelling environment but will be calculated and added automatically. It must be mentioned in this context that inferred values may be defined recursively (directly or indirectly via other inferred properties). This will result in the issue that solving such properties will turn out to be a fix-point problem, so that possibly the entire M-graph will have to be taken into account. In particular, the solution may even be ambiguous. Nevertheless in this context inferred properties are simply seen as an equality constraint. By doing so, none of the above-mentioned problems will occur as the solution for the inferred properties has already been given by the M-graph to be inspected.

- **context:** context included for context-sensitive keys. According to the given definition, all concepts contained by visible meta-packages can be accessed by the simple key as long as it is unique. For further details concerning the formal definition and use of context included please refer to Section 7.5.5, *Local keys, namespaces and visibility*, p. 137.

Concrete Syntax. The formal textual concrete syntax for *PropertyDef* is defined in Listing 9.65.

Listing 9.65: Textual Concrete Syntax for *PropertyDef*

```
456 PropertyDef: ("?" _ | P:assumption | _ "?" _)
457   (P:keyType) (P:linkType) (P:property)
458   ("[" || P:multiplicity / "," || "]")
459   ("(" || P:pomsetRestriction / "," || ")")
460   (_ ("↔" OR "<->") _ || P:opposite / ",")
461   (_ ":" _ || P:domain)
462   (_ ":=" _ || P:inferredValue) ;
```

The concrete syntax for property definitions has been defined such that a compact language evolves which reuses some textual notations from UML class diagrams. First of all, the assumption of the property will be encoded between two question marks. Afterwards, the property name itself will be encoded. Two modifiers can be found in front of the property name: First, a **"(K)"** denotes that the property will be used as the local key; secondly, a **"&"** marks the property as a reference or a **"%"** marks the property as an instantiating property; if both **"&"** and **"%"** are missing, the property will be a compositional property. The property name will be followed by an (optional) comma-separated set of multiplicities within squared brackets. Afterwards, the (optional) pomset restriction will be defined within parentheses, followed by an (also optional) comma-separated set of opposite properties marked by ↔ or the corresponding ascii symbols <−> at the beginning. Then, the domain will be encoded, marked by a preceding colon. Finally, if set, the inferred value identified by **":="** will be given.

Example. An example for the concept *PropertyDef* will be provided in the following Listing 9.66:

Listing 9.66: Example for *PropertyDef*

```
? (bool)P:married ? &spouse[1..1] ↔ spouse : C:Person
```

The given example may be a property definition within a concept *Person* specifying whether a person is married. It must also comprise a property *spouse* which is a person as well. Due to the opposite value defined (which will be spouse again), it is said that if one person is the spouse of another person, that other person is also the spouse of the first one. Note that this example provides an exact representation of a self-connected bidirectional property.

Note that each of the definitions for abstract syntaxes in this Chapter 9, *The overall specification of M2L*, p. 171 that are no enumerations or external concepts represent further possible examples of property definitions.

9.3.11. Concept *ConceptType*

The concept *ConceptType* enumerates the three types of concepts, namely *strong concepts*, *weak concepts*, and *attribute concepts*. A concept type is specified within a concept definition (see Section 9.3.5, *Concept* ConceptDef, p. 192). The three concept types are ordered by an implication: Each attribute concept fulfils the constraints for a weak concept. And each weak concept fulfils the constraints for a strong concept. More precisely, strong concepts do not lead to any additional constraints for the concept concerning the concept type.

Generally speaking, concept types deal with the incoming edges of a node, both compositional as well as general. Compositional relationships between nodes are marked by the special property *composite* within M-graphs. Such a compositional relationship can be forced by setting the link type of a property to *composite* (see Section 9.3.13, *Concept* LinkType, p. 207). As described in Section 9.1.1, *Concept* Any, p. 172, the property *composite* has several restrictions so that the resulting edge for *composite* always describes a rooted forest in terms of graph theory. It can be distinguished between root nodes and child nodes (i. e. those nodes that are no root nodes) within such a forest.

While nodes marked by strong concepts may be both root nodes as well as child nodes (no additional constraints are imposed on strong concepts), nodes marked by weak concepts may only occur as child nodes. In other words, each node marked by a weak concept must be part of another node in terms of a containment relationship. Attribute nodes (i. e. nodes marked by attribute concepts) must not have any additional incoming edges besides the *composite* edge and one edge which triggers the composition (the source node of this edge will therefore be identical to the source node of the *composite* edge). In other words, attribute nodes may only be contained but can not be referenced by other nodes.

Abstract Syntax. The formal abstract syntax for *ConceptType* is defined in Listing 9.67.

Listing 9.67: Abstract Syntax for *ConceptType*

```
155 enum ConceptType = { Strong, Weak, Attribute } ;
```

The enumeration values of the concept *ConceptType* are:

- **Strong:** represents strong concepts. Thus, no additional constraint will be added.
- **Weak:** represents weak concepts which are more restrictive than strong concepts. Nodes marked by weak concepts must always be part of another node in terms of a containment relationship.
- **Attribute:** represents attribute concepts which are again more restrictive than weak concepts. Nodes marked by attribute concepts must not be referenced in any way besides the containing node.

Concrete Syntax. The formal textual concrete syntax for *ConceptType* is defined in Listing 9.68.

Listing 9.68: Textual Concrete Syntax for *ConceptType*

```
464 ConceptType.Strong: ;
465 ConceptType.Weak: "!" ;
466 ConceptType.Attribute: "!!" ;
```

As strong concepts do not lead to any additional constraint, the encoding will be empty. Whereas weak concepts are marked by one exclamation mark, attribute concepts are marked by a double exclamation mark. This notation shall also emphasise the implication relationship between concept types.

Example. An example for the concept *ConceptType* will be provided in the following Listing 9.69:

Listing 9.69: Example for *ConceptType*

```
!!
```

The concept type *Attribute* will be shown.

9.3.12. Concept *KeyType*

This concept enumerates all the valid key types. In detail they are *PrimaryLocalkey* and *AlternativeLocalkey*. For details please refer to Section 7.5.5, *Local keys, namespaces and visibility*, p. 137.

Abstract Syntax. The formal abstract syntax for *KeyType* is defined in Listing 9.70.

Listing 9.70: Abstract Syntax for *KeyType*

```
156 enum KeyType = {
157   PrimaryLocalkey, AlternativeLocalkey
158 };
```

The enumeration values of the concept *KeyType* are:

- **PrimaryLocalkey:** marks a property as the primary local key.
- **AlternativeLocalkey:** marks a property as a alternative local key.

Concrete Syntax. The formal textual concrete syntax for *KeyType* is defined in Listing 9.71.

Listing 9.71: Textual Concrete Syntax for *KeyType*

```
468 KeyType.PrimaryLocalkey: "(PK)" ;
469 KeyType.AlternativeLocalkey: "(K)" ;
```

As strong concepts do not lead to any additional constraint, the encoding will be empty. Whereas weak concepts are marked by one exclamation mark, attribute concepts are marked by a double exclamation mark. This notation shall also emphasise the implication relationship between concept types.

Example. An example for the concept *KeyType* will be provided in the following Listing 9.72:

Listing 9.72: Example for *KeyType*

```
(PK)
```

The key type *PrimaryLocalkey* is shown.

9.3.13. Concept *LinkType*

The concept *LinkType* enumerates the three types of links between nodes, namely *reference links*, *composition links*, and *instantiation links*. A link type will be specified within a property definition (see Section 9.3.10, *Concept* PropertyDef, p. 201). The three link types are ordered by an implication: Each instantiation link fulfils the constraints for a composition link. And each composition link fulfils the constraints for a reference link. More precisely, reference links do not lead to any additional constraints for a property concerning the link type.

Generally speaking, link types deal with special properties, namely *composite* and *template*, within M-graphs. Figure 9.1 provides an example which shows an automaton a_1 defined within a library l_1 and having one state s_1. Then, a component c_1 uses this automaton a_1 by instantiating it by a_2.

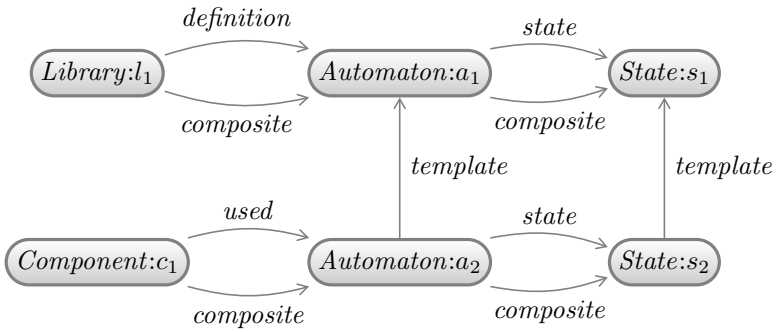


Figure 9.1.: Example for compositions and instantiations

If the link type of a property is defined as being a composition, a second edge must be specified labelled by *composite* for each edge labelled by said property. Hereupon, a composition link between these two nodes will be established. Please refer to Section 7.4.9, *Compositional properties*, p. 131 for further details. Examples of such compositions will be provided in Figure 9.1.

Instantiation links further require for the composed element to be a (deep) copy of a template. The template node will be marked by an edge labelled with *template*. Please refer to Section 7.5.6, *Instantiating Properties*, p. 145 for further details. The automaton a_2 in Figure 9.1 is an instance of automaton a_1 due to the edge labelled by *template*. Thus, the property *used* satisfies the constraint resulting from an instantiation link, and due to that, *used* will be an instantiation link.

Abstract Syntax. The formal abstract syntax for *LinkType* is defined in Listing 9.73.

Listing 9.73: Abstract Syntax for *LinkType*

```
159 enum LinkType = {
160   Reference, Composition, Instantiation
161 };
```

The enumeration values of the concept *LinkType* are:

- **Reference:** represents a reference link. Thus, no additional constraint will be added.

- **Composition:** represents a composition link which is more restrictive than a reference link. Properties marked by a composition link require an additional *composite* edge within the M-graph.
- **Instantiation:** represents an instantiation link which is again more restrictive than composition links. Properties marked by an instantiation link must reference a concept which is a (deep) copy of another concept marked by the *template* edge within the M-graph.

Concrete Syntax. The formal textual concrete syntax for *LinkType* is defined in Listing 9.74.

Listing 9.74: Textual Concrete Syntax for *LinkType*

```
471 LinkType.Reference: "&" ;
472 LinkType.Composition: ;
473 LinkType.Instantiation: "%" ;
```

Although compositions have a stronger constraint as references, compositions are seen as the standard case as they are mainly used within metamodels. Thus, compositions are encoded without any additional marker. Properties referencing attribute concepts may therefore also be written as usual, i. e. without any additional symbol. An ampersand is commonly used for references and hence for encoding references. The percent sign will be used for representing instantiations. It has been chosen as the two 0's may represent the template-copy-relationship: one stands for the template, the other one for the instance copy.

Example. An example for the concept *LinkType* will be provided in the following Listing 9.75:

Listing 9.75: Example for *LinkType*

```
%
```

The link type *Instantiation* is shown.

9.3.14. Concept *PomsetType*

As described in Chapter 5, *Models as Abstract Words*, p. 81, both the theory of Edge Algebra and the metamodelling language *M2L* are based on pomsets instead of the set theory. This approach allows for handling duplicates and order in a formal way. In many situations, properties should be restricted to a subtype of partially ordered multisets which have been introduced in Chapter 4, *Pomsets in the context of metamodelling*, p. 61. The enumeration *PomsetType* defines all possible subtypes, namely *Singleton*, *Set*, *Bag*, *List*, *Toset*, *Poset*, and *Pomset*.

Note that duplicates are not based on identical nodes within the M-graph but on the equivalence relation defined in Chapter 4, *Pomsets in the context of metamodelling*, p. 61 (see Chapter 5, *Models as Abstract Words*, p. 81).

Abstract Syntax. The formal abstract syntax for *PomsetType* is defined in Listing 9.76.

Listing 9.76: Abstract Syntax for *PomsetType*

```
enum PomsetType = {
  Set, Bag, List, Toset, Poset, Pomset
};
```

The enumeration values of the concept *PomsetType* are:

- **Set:** pomset having neither order nor duplicates.
- **Bag:** pomset which has no order but may have duplicates.
- **List:** pomset which has a total order and may have duplicates.
- **Toset:** pomset having a total order but no duplicates.
- **Poset:** pomset having a partial order but no duplicates.
- **Pomset:** Any pomset will be allowed. As this pomset type does not add any additional constraint, it will not be specified explicitly in most cases.

Concrete Syntax. The formal textual concrete syntax for *PomsetType* will be defined in a canonical way by simply encoding the names of the enumeration values themselves. Thus, no custom textual syntax will have to be specified.

Example. An example for the concept *PomsetType* will be provided in the following Listing 9.77:

Listing 9.77: Example for *PomsetType*

```
Bag
```

The pomset type *Bag* is encoded.

9.4. Package *ORG.Metamodels.M2L.ConcreteSyntax*

In order to define a concrete notation for the concepts given by the abstract syntax, one or more concrete syntaxes will have to be specified. The focus of the present thesis is on how models are presented to humans for both browsing as well as editing them within an IDE. As those presentations are usually not used for persisting the models, there is additional freedom: Not all model information within one concrete syntax will have to be encoded. Concrete syntax will instead rather be treated as a view to abstract syntax. Thus, some information may be skipped in dedicated concrete syntaxes. Nevertheless, all model information should be comprised by at least one concrete syntax in order to allow humans to access all parts of the model information.

Concrete syntaxes will always be specified on the basis of abstract syntax within *M2L*. So in order to decide whether a given model is valid with respect to a concrete syntax, abstract syntax will always have to be taken into account. As has already been mentioned, any number of concrete syntaxes may be defined for one abstract syntax. The relationships (especially with regard to consistency) between these concrete syntaxes are in turn established by abstract syntax. Such concrete syntaxes may generally be of different types. It will be differentiated between textual, diagrammatical and tabular concrete syntaxes from a conceptional point of view. Although *M2L* takes this generality into account, it still defines a modelling technique for textual syntaxes up to now. This package provides the common and abstract concepts for defining concrete syntaxes independent of a particular syntax type. Please refer to Section 9.5, *Package* ORG.Metamodels.M2L.ConcreteSyntax.Textual, p. 217 for textual syntax definitions. According to *ConcreteSyntaxPackage*, the package structure within a concrete syntax needs to be structured the same way as given by abstract syntax. Table 9.4 shows the list of all concepts defined:

Concept	Description
SyntaxIdentifier	the identifier for concrete syntaxes
ConcreteSyntax	the main concept encapsulating concrete syntax
ConcreteSyntaxPackage	builds the package structure within concrete syntaxes
ConcreteSyntaxDef	a syntax definition for a single concept

Table 9.4.: List of concepts defined in *ORG.Metamodels.M2L.ConcreteSyntax*

As more than one concrete syntax is generally feasible, identifiers must be defined for each syntax. These identifiers must be unique among all syntax types. One concrete syntax may be marked as default syntax for each syntax type. For textual syntaxes, for example, the default syntax can be accessed by the reserved syntax identifier *"TEXTUAL DEFAULT"*. The default syntax will be used whenever the desired syntax has not been defined or if no syntax identifier has been given. In addition to the default syntax, a canonical syntax will also be predefined for each syntax type. For textual syntaxes, for example, the canonical syntax can be accessed by the reserved syntax identifier *"TEXTUAL CANONIC"*. The canonical syntax will be used whenever the default syntax is requested but no default syntax has been defined. A canonical syntax can be derived from an abstract syntax without any additional information about concrete syntax. This allows a language engineer to browse and edit models in early language development stages (e. g. for validating a modelling language), as no concrete syntax definition will be necessary.

9.4.1. Concept *SyntaxIdentifier*

In contrast to abstract syntax, concrete syntaxes have a unique identifier. This results from the fact that a metamodel consists of exactly one abstract syntax but may have several concrete syntaxes. The concept *SyntaxIdentifier* represents – as the name suggests – such an identifier for concrete syntaxes. A syntax identifier itself is represented by an arbitrary case-sensitive and non-empty string. Note that syntax identifiers must be unique even over different syntax types. Hence, there must not be a textual and a diagrammatical syntax with the same syntax identifier. With regard to default and canonical syntaxes there are two reserved syntax identifiers:

- *"TEXTUAL DEFAULT"* representing the default textual syntax, and
- *"TEXTUAL CANONIC"* representing the canonical textual syntax.

Abstract Syntax. The formal abstract syntax for *SyntaxIdentifier* is defined in Listing 9.78.

Listing 9.78: Abstract Syntax for *SyntaxIdentifier*

```
168 SyntaxIdentifier!! refines Identifier ;
```

The concept *SyntaxIdentifier* refines the concept *Identifier* without any additional restrictions or properties. As abstract syntax is not further restricted, syntax identifiers may also contain any kind of special character.

Concrete Syntax. Concrete syntax is (indirectly) inherited (via *Identifier*) from *String* (see Section 9.1.6, *Concept* String, p. 177).

Example. An example for the concept *SyntaxIdentifier*, representing the syntax identifier *M2L/Text* specified in the meta-metamodel itself will be provided in the following Listing 9.79:

Listing 9.79: Example for *SyntaxIdentifier*

```
"M2L/Text"
```

Quotes are necessary in this example as the special character *'/'* will be used within the syntax identifier.

9.4.2. Concept *ConcreteSyntax*

The concept *ConcreteSyntax* encapsulates the definition of one concrete syntax within a metamodel. Structured by syntax packages (see Section 9.4.3, *Concept* ConcreteSyntaxPackage, p. 213) it contains all syntax definitions (indirectly via syntax packages) (see Section 9.4.4, *Concept* ConcreteSyntaxDef, p. 215) that are defined within this concrete syntax.

This concept is abstract as a dedicated refined non-abstract concept will be defined for the different syntax types just as in Section 9.5.1, *Concept* TextualSyntax, p. 218. These refined concepts should then ensure that they only contain syntax definitions of the corresponding syntax type (for textual syntaxes this will e. g. be Section 9.5.3, *Concept* TextualSyntaxDef, p. 220).

Abstract Syntax. The formal abstract syntax for *ConcreteSyntax* is defined in Listing 9.80.

Listing 9.80: Abstract Syntax for *ConcreteSyntax*

```
[ConcreteSyntax!] refines Named ::=
  name : C:SyntaxIdentifier,
  alternativeName : C:SyntaxIdentifier,
  isDefault : C:Boolean,
  syntaxPackage[0..*](Set) : C:ConcreteSyntaxPackage ;
```

The properties of the concept *ConcreteSyntax* are:

- **name (refined):** identifier of this concrete syntax.
- **alternativeName (refined):** alternative identifiers of this concrete syntax.
- **default:** indicates whether this concrete syntax is marked as being a default syntax. If more than one concrete syntax is marked as being a default syntax for the same concept, they are seen as being alternatives. In textual syntaxes this will be achieved by a switch (see Section 9.5.12, *Concept* Switch, p. 230).
- **syntaxPackage:** set of all (root) syntax packages this concrete syntax consists of. The structure of the syntax packages will have to match the meta-package structure of the abstract syntax. Note that the transitively reachable sub-packages will not be included herein.

Concrete Syntax. The formal textual concrete syntax for *ConcreteSyntax* is defined in Listing 9.81.

Listing 9.81: Textual Concrete Syntax for *ConcreteSyntax*

```
ConcreteSyntax: ((bool)P:default ? "default" _)
  "concrete" "syntax" _ <Named> _ "{"
  (nl | P:syntaxPackage)
  nl "}" ;
```

Each concrete syntax is marked by the composed keyword **"concrete syntax"** followed by the syntax identifier thereof including the optional alternative identifiers thereof (see Section 9.1.8, *Concept* Named, p. 179). The keyword **"default"** in front of this composed keyword indicates whether the concrete syntax is marked as being the default syntax or not. The syntax packages will finally be listed within curly brackets.

As the concept *ConcreteSyntax* is abstract, this syntax definition will be defined for the use within refining concepts such as in Section 9.5.1, *Concept* TextualSyntax, p. 218 by adding an additional keyword (e. g. **"textual"** for textual syntaxes).

Example. An example for the concept *ConcreteSyntax* will be provided in the following Listing 9.82:

Listing 9.82: Example for *ConcreteSyntax*

```
default concrete syntax "M2L/Text" ("Textual M2L") {
  syntaxpackage ORG {
    ...
  }
```

```
}
```

This example shows the encoding of concrete syntax of the metamodelling language *M2L* itself as defined in the meta-metamodel thereof. Note that the keyword **"textual"** will be skipped here, as only the more abstract version will be encoded herein. Details within syntax packages will be skipped here. Please refer to Section 9.4.3, *Concept* ConcreteSyntaxPackage, p. 213 for details. In addition, an alternative name *Textual M2L* will be added for illustration.

9.4.3. Concept *ConcreteSyntaxPackage*

The package structure for the concrete syntax will be built by the concept *ConcreteSyntaxPackage*. To shorten the illustration, the term *concrete* will often be omitted and the concept will just be named *syntax package*. Concrete syntax packages may again contain concrete syntax packages as sub-packages. The resulting tree structure must match the structure defined in abstract syntax. Hence, each concrete syntax package must correspond to a meta-package from the abstract syntax which belongs to the same metamodel. In addition, corresponding packages must be contained by corresponding packages in turn. Note that not every meta-package must, however, have a corresponding concrete syntax package. The local key will indirectly be defined by the local key of the meta-package. Thus, the resulting tree structure of the concrete syntax packages forms an appropriate namespace for the contained concrete syntax definitions.

Abstract Syntax. The formal abstract syntax for *ConcreteSyntaxPackage* is defined in Listing 9.83.

Listing 9.83: Abstract Syntax for *ConcreteSyntaxPackage*

```
174 [ConcreteSyntaxPackage!] ::=
175   &metapackage[1..1] : P:includedContext.
176     (P:metapackage ⊎ P:subpackage),
177    subpackage[0..*](Set) : C:ConcreteSyntaxPackage,
178    concreteSyntaxDef[0..*](Set) : C:ConcreteSyntaxDef,
179   &includedContext := ϵ(
180         (⇆(P:concreteSyntax.P:syntaxPackage).
181        P:abstractSyntax)
182       ⊎
183         (⇆P:subpackage.P:metapackage)
184       ),
185    lkey := P:metapackage.P:lkey ;
```

The properties of the concept *ConcreteSyntaxPackage* are:

- **metapackage:** corresponding meta-package for this syntax package. If this syntax package is a root package, all root meta-packages from the abstract syntax are valid; if this syntax package is a sub-package (i. e. there is a superordinate syntax package as a parent), all sub-packages of the meta-package corresponding to the superordinate syntax package are valid. Formally, this will be expressed via the *context*. Note that it is not allowed to model more than one syntax package referencing one and the same meta-package. This is formally forbidden due to the lkey constraint, as such a scenario will always result in duplicate lkeys.

- **subpackage:** set of all sub-packages contained in this syntax package. Note that the transitively reachable sub-packages will not be included herein.
- **concreteSyntaxDef:** set of concrete syntax definitions defined within this syntax package.
- **context (inferred):** context included for context-sensitive keys. According to the given definition, either all composed nodes (in particular the root meta-packages) of the abstract syntax or all composed nodes of the meta-package, associated with the superordinate syntax package, can be referenced by the simple key as long as it is unique. For further details concerning the formal definition and use of context included please refer to Section 7.5.5, *Local keys, namespaces and visibility*, p. 137.
- **lkey (inferred):** the local key for the concrete syntax package will be inferred from the lkey of the corresponding meta-package. Hereupon it will also be ensured that two syntax packages do not have the same corresponding meta-package.

Concrete Syntax. The formal textual concrete syntax for *ConcreteSyntaxPackage* is defined in Listing 9.84.

Listing 9.84: Textual Concrete Syntax for *ConcreteSyntaxPackage*

```
481 ConcreteSyntaxPackage: "syntaxpackage" _
482   (&P:metapackage) _ "{"
483   (nl | P:concreteSyntaxDef)
484   (nl | P:subpackage)
485   nl "}" ;
```

A concrete syntax package is encoded quite similar to meta-packages: Instead of **"metapackage"**, a concrete syntax package starts with **"syntaxpackage"**. Whereas said is followed by the package name for meta-packages, the reference to the corresponding meta-package will be encoded for concrete syntaxes. Due to the fact that the included context always contains the referenced meta-package, the local key will always be sufficient. All contained concrete syntax definitions followed by the defined sub-packages will be encoded within the curly brackets.

Example. An example for the concept *ConcreteSyntaxPackage* will be provided in the following Listing 9.85:

Listing 9.85: Example for *ConcreteSyntaxPackage*

```
syntaxpackage M2L {
  Metamodel: ... ;
  syntaxpackage AbstractSyntax {
    Concept: ... ;
  }
}
```

The example illustrates a simplified concrete syntax package taken from the metamodel of *M2L*. It references the meta-package *M2L* and contains a concrete syntax definition for the concept *Metamodel* and a sub-package referring to the meta-package *AbstractSyntax* (which will be possible as the meta-package *AbstractSyntax* is a sub-package of the meta-package *M2L*). The latter concrete syntax package will in turn contain a concrete syntax definition

for the concept *Concept*. Details concerning the concept definitions will be omitted herein. Please refer to Section 9.4.4, *Concept* ConcreteSyntaxDef, p. 215.

9.4.4. Concept *ConcreteSyntaxDef*

The concept *ConcreteSyntaxDef* represents a syntax definition for a single concept. This concept is abstract as it forms the basis for various syntax types such as the textual one. It mainly defines the link to the concept from abstract syntax. All other details are specific to a dedicated syntax type.

Abstract Syntax. The formal abstract syntax for *ConcreteSyntaxDef* is defined in Listing 9.86.

Listing 9.86: Abstract Syntax for *ConcreteSyntaxDef*

```
186 [ConcreteSyntaxDef!] ::>
187   &conceptDef[1..1] : P:includedContext.P:conceptDef,
188   &includedContext :=
189     ⇆P:concreteSyntaxDef.P:metapackage,
190   lkey := P:conceptDef.P:lkey ;
```

The properties of the concept *ConcreteSyntaxDef* are:

- **concept:** corresponding concept from abstract syntax for this concrete syntax definition. Only the concepts defined within the associated meta-package can be referenced. Formally, this will be expressed via the *context*. Note that it is not allowed to model more than one syntax definition referencing one and the same concept. This is formally forbidden due to the lkey constraint, as such a scenario will always result in duplicate lkeys.
- **context (inferred):** context included for context-sensitive keys. According to the given definition, all composed nodes (in particular the defined concepts) of the associated meta-package can be referenced by the simple key as long as it is unique. For further details concerning the formal definition and use of context included please refer to Section 7.5.5, *Local keys, namespaces and visibility*, p. 137.
- **lkey (inferred):** the local key for the concrete syntax definition will be inferred from the lkey of the corresponding concept. Hereupon it will also be ensured that two syntax definitions do not have the same corresponding concept.

Concrete Syntax. The formal textual concrete syntax for *ConcreteSyntaxDef* is defined in Listing 9.87.

Listing 9.87: Textual Concrete Syntax for *ConcreteSyntaxDef*

```
486 ConcreteSyntaxDef: (&P:conceptDef) ":" _ ;
```

According to the fact that this concept is an abstract one, it only defines a prefix which should be used within all refining concepts. That concept for which the syntax shall be defined will be referenced first. Due to the context included this may always be realised by the simple lkey. The referenced concept will be followed by a colon. Afterwards, the dedicated syntax definition for the corresponding syntax type will follow.

Example. An example for the concept *ConcreteSyntaxDef* will be provided in the following Listing 9.88:

Listing 9.88: Example for *ConcreteSyntaxDef*

```
Person :
```

The present example illustrates a concrete syntax definition for the concept *Person*. As *ConcreteSyntaxDef* is an abstract concept, the example only shows the common prefix of all types of concrete syntax definitions. After the colon, the specific part of syntax definition will begin. Please refer to Section 9.5.3, *Concept* TextualSyntaxDef, p. 220 for a textual syntax specification.

9.5. Package *ORG.Metamodels.M2L.ConcreteSyntax.Textual*

This package defines the concepts necessary for describing the textual concrete syntax of a modelling language. A textual concrete syntax generally defines a *textual concrete language* – also known as a *formal language* and thus a concrete syntactical notation – based on an abstract modelling language as defined in Section 9.3, *Package* ORG.Metamodels.M2L.AbstractSyntax, p. 188. As a concrete syntax definition – as described in the present section – is always based on an abstract syntax, a complete language specification always needs both, an abstract and a concrete syntax definition.

Textual syntaxes in *M2L* are described by a template-based approach which is suitable for both pretty-printing as well as parsing. The style of the resulting syntax specification language shall empathize established grammar specification languages such as EBNF in order to provide a comprehensible specification technique for language engineers in particular. Due to the combination of abstract syntax and concrete syntax described herein, context-sensitive conditions can easily be expressed.

In this context, the M-graph (wherein the structure thereof has been described by the abstract syntax) takes over the role of the abstract syntax tree (AST). In contrast to an AST the M-graph is – as the name suggests – a graph instead of a tree. All rooted trees which are subgraphs of the given M-graph are possible ASTs for a textual concrete language. In general, the tree does not need to span the M-graph as it may describe only a part of the model. These additional cross-references rendering the tree a graph can be seen as the context-sensitive part of the language.

Up to now, the exact expressiveness of textual languages described by *M2L* has not been analysed in detail. Nevertheless is an expressiveness close to context-sensitive grammars assumed. At this point it must be considered that all freedom in defining abstract syntax will be required for building arbitrary textual languages. In other words, the possible concrete syntaxes will be limited for a given abstract syntax as the abstract syntax already restricts possible abstract syntax trees to the spanning trees or at least trees which are subgraphs of the given M-graph.

The challenge of textual syntax definitions in *M2L* was being able to describe textual syntaxes from a documentation point of view but not from an implementation point of view. Tokens or defining the language in the form of a (mostly difficult to understand) LL grammar just for optimizing the parser shall therefore not be mentioned. Nevertheless should it be possible to generate both pretty-printer and parser without any additional information.

Important issues when designing *M2L* were also extensibility and the combination of modelling languages. The present thesis therefore decided to use a scannerless approach in order to skip tokens which allows for the use of keywords as identifies as well. This is important upon the extension or combination of languages, as here, new keywords evolve in many cases which might render existing models invalid. In other words, such an approach avoids that concrete syntaxes will influence the abstract syntax. When, for example, Java introduced enumerations, the new keyword *enum* was introduced. Since then, *enum* could no longer be used as an identifier name resulting in a downwards incompatibility.

Parsing performance on the other hand has a lower priority when designing *M2L*. The reason for that is that textual syntaxes are mainly used for a presentation of models to humans. Instead of persisting models into text files, models are e. g. stored in a model database using abstract syntax in a direct way, thus rendering them available in an already parsed way. If the model is accessed that way – e. g. for model analysis or code generation – no high performance parser will be necessary so as to parse the entire model each time. Parsing will only be necessary when humans are manually altering the model which will naturally

affect only a very small part of the model. Note that this approach necessitates incremental parsing mechanisms instead.

TextualSyntax, *TextualSyntaxPackage*, and *TextualSyntaxDef* are refinements of the predefined concepts from the generic package for concrete syntaxes defined in Section 9.4, *Package* ORG.Metamodels.M2L.ConcreteSyntax, p. 210. A textual syntax definition of a concept consists of a *SyntaxTemplate* which describes the textual syntax by concatenation of several *TemplateElements*. *TemplateElements* are the basic modelling elements for textual syntaxes, such as *Terminal* or *NonTerminal*. All in all, the defined concepts are listed in Table 9.5:

Concept	Description
TextualSyntax	the main concept encapsulating textual syntax
TextualSyntaxPackage	builds the package structure within textual syntaxes
TextualSyntaxDef	a textual syntax definition for a single concept
SyntaxTemplate	describes a textual syntax by concatenating template elements
TemplateElement	abstract concept for one element of a syntax template
Terminal	template element for terminal symbols
ProperTerminal	terminal symbols not treated as whitespaces
WhitespaceTerminal	terminal symbols treated as whitespaces
Whitespace	enumerates possible whitespaces
NonTerminal	models a non-terminal by referring to abstract syntax
Option	models an option depending on an Edge Algebra predicate
Switch	models a set of alternative textual syntaxes
IncludeSyntaxDef	includes another textual syntax definition

Table 9.5.: List of concepts defined in *ORG.Metamodels.M2L.ConcreteSyntax.Textual*

9.5.1. Concept *TextualSyntax*

The concept *TextualSyntax* encapsulates the definition of one textual concrete syntax within a metamodel by refining the concept *ConcreteSyntax* (see Section 9.4.2, *Concept* ConcreteSyntax, p. 211). This concept ensures that only textual syntax packages (see Section 9.5.2, *Concept* TextualSyntaxPackage, p. 219) and textual syntax definitions (see Section 9.5.3, *Concept* TextualSyntaxDef, p. 220) will be contained.

Abstract Syntax. The formal abstract syntax for *TextualSyntax* is defined in Listing 9.89.

Listing 9.89: Abstract Syntax for *TextualSyntax*

```
193 TextualSyntax! refines ConcreteSyntax ::=
194   syntaxPackage : C:TextualSyntaxPackage ;
```

The properties of the concept *TextualSyntax* are:

- **syntaxPackage (refined):** set of all (root) syntax packages this textual syntax consists of. Due to the refinement, this property may only contain textual syntax packages.

Concrete Syntax. The formal textual concrete syntax for *TextualSyntax* is defined in Listing 9.90.

Listing 9.90: Textual Concrete Syntax for *TextualSyntax*

```
489 TextualSyntax: "textual" _ <ConcreteSyntax> ;
```

As the refining abstract concept already specifies a concrete syntax, only the prefix **"textual"** will be added.

Example. An example for the concept *TextualSyntax* will be provided in the following Listing 9.91:

Listing 9.91: Example for *TextualSyntax*

```
textual default concrete syntax "M2L/Text" ("Textual M2L") {
  syntaxpackage ORG {
    ...
  }
}
```

This example shows the encoding of the textual syntax of the metamodelling language *M2L* itself as defined in the meta-metamodel thereof. In contrast to the example for the abstract concept for *ConcreteSyntax*, (see Section 9.4.2, *Concept* ConcreteSyntax, p. 211), the keyword **"textual"** will now be encoded. Details within syntax packages will be skipped here. Please refer to Section 9.5.2, *Concept* TextualSyntaxPackage, p. 219 for details. In addition, an alternative name *Textual M2L* will be added for illustration.

9.5.2. Concept *TextualSyntaxPackage*

The package structure for the textual concrete syntax will be built by the concept *TextualSyntaxPackage*. It refines the concept *ConcreteSyntax* and ensures that a textual syntax package will again contain (maybe indirectly) only textual syntax definitions (see Section 9.4.4, *Concept* ConcreteSyntaxDef, p. 215). Please refer to Section 9.4.2, *Concept* ConcreteSyntax, p. 211 for further details.

Abstract Syntax. The formal abstract syntax for *TextualSyntaxPackage* is defined in Listing 9.92.

Listing 9.92: Abstract Syntax for *TextualSyntaxPackage*

```
195 TextualSyntaxPackage!
196     refines ConcreteSyntaxPackage ::=
197   subpackage : C:TextualSyntaxPackage,
198   concreteSyntaxDef : C:TextualSyntaxDef ;
```

The properties of the concept *TextualSyntaxPackage* are:

- **subpackage (inferred):** set of all sub-packages contained in this concrete syntax package. Note that the transitively reachable sub-packages will not be included herein. Due to the refinement, this property may only contain textual syntax packages.
- **concreteSyntaxDef (inferred):** set of concrete syntax definitions defined within this syntax package. Due to the refinement, this property may only contain textual syntax packages.

Concrete Syntax. Concrete syntax is inherited from *ConcreteSyntaxPackage* (see Section 9.4.3, *Concept* ConcreteSyntaxPackage, p. 213).

Example. As the syntax definition is similar to that of the refined concept *ConcreteSyntaxPackage*, please refer to the example in Listing 9.85 from Section 9.4.3, *Concept* ConcreteSyntaxPackage, p. 213.

9.5.3. Concept *TextualSyntaxDef*

According to the refined concept *ConcreteSyntaxDef*, the concept *TextualSyntaxDef* represents a textual syntax definition for a single concept. It consists of exactly one template specifying the textual concrete syntax.

Abstract Syntax. The formal abstract syntax for *TextualSyntaxDef* is defined in Listing 9.93.

Listing 9.93: Abstract Syntax for *TextualSyntaxDef*

```
199 TextualSyntaxDef! refines ConcreteSyntaxDef ::>
200   mainSyntaxTemplate[1..1] : C:SyntaxTemplate ;
```

The properties of the concept *TextualSyntaxDef* are:

- **mainSyntaxTemplate:** template specifying the textual concrete syntax. Note that this property must always exist even though this syntax template does not contain any template elements (see Section 9.5.5, *Concept* TemplateElement, p. 222).

Concrete Syntax. The formal textual concrete syntax for *TextualSyntaxDef* is defined in Listing 9.94.

Listing 9.94: Textual Concrete Syntax for *TextualSyntaxDef*

```
490 TextualSyntaxDef: <ConcreteSyntaxDef>
491   (P:mainSyntaxTemplate) _ ";" ;
```

The textual syntax for the textual syntax definition itself simply encodes the main syntax template after the prefix which has already been specified for the refining concept *ConcreteSyntaxDef* (see Section 9.4.4, *Concept* ConcreteSyntaxDef, p. 215) and which will be finalized by a semicolon.

Example. An example for the concept *TextualSyntaxDef* will be provided in the following Listing 9.95:

Listing 9.95: Example for *TextualSyntaxDef*

```
Person: "person" _ (name) ;
```

The given example encodes a concept *Person* by a terminal **"person"**, followed by a whitespace, followed by the non-terminal encoding the property *name*. Note that each listing in the sections named *Concrete Syntax* within this chapter represent additional examples of textual syntax definitions.

9.5.4. Concept *SyntaxTemplate*

The concept *SyntaxTemplate* encodes the concatenation of a set of template elements such as terminals or non-terminals (see Section 9.5.5, *Concept* TemplateElement, p. 222 for details). Besides the occurrence as main syntax template within the textual syntax definition (see Section 9.5.3, *Concept* TextualSyntaxDef, p. 220), it is also referenced within *Option* to model both then- and else-case (see Section 9.5.11, *Concept* Option, p. 229), within *Switch* to model alternatives (see Section 9.5.12, *Concept* Switch, p. 230), and within *NonTerminal* to model prefixes, suffixes etc. (see Section 9.5.10, *Concept* NonTerminal, p. 226). A syntax template may also be empty, thus it does not contain any template elements.

The concatenation of template elements means that they are expected within the textual representation in the same order. Note that between every two consecutive template elements whitespaces may occur in the textual representation to be parsed. Thus there is a difference between **"Hello" "World"** and **"HelloWorld"** as the former one allows an arbitrary amount of whitespaces. Note that this approach renders the explicit modelling of whitespaces dispensable for parsing: **"Hello"_"World"** excludes the same textual representations as **"Hello" "World"**. Even a representation without any whitespace will be excluded in both cases as an explicit whitespace within a syntax template does not call for it. The difference comes up upon pretty-printing the model: A whitespace will then only be encoded when it is modelled explicitly.

Abstract Syntax. The formal abstract syntax for *SyntaxTemplate* is defined in Listing 9.96.

Listing 9.96: Abstract Syntax for *SyntaxTemplate*

```
202 SyntaxTemplate!! ::=
203   templateElement[0..*](List) : C:TemplateElement ;
```

The properties of the concept *SyntaxTemplate* are:

- **templateElement:** list of template elements that are concatenated within this syntax template.

Concrete Syntax. The formal textual concrete syntax for *SyntaxTemplate* is defined in Listing 9.97.

Listing 9.97: Textual Concrete Syntax for *SyntaxTemplate*

```
493 SyntaxTemplate: (P:templateElement / _) ;
```

Syntax templates are simply encoded by concatenating the contained template elements with a whitespace therebetween.

Example. An example for the concept *SyntaxTemplate* will be provided in the following Listing 9.98:

Listing 9.98: Example for *SyntaxTemplate*

```
"person" _ (name)
```

The given example shows a syntax template with tree template elements. The first one (**"person"**) is a proper terminal; the second one (_) is a whitespace terminal; and the third one ((name)) is a non-terminal. For a detailed information about available template elements please refer to Section 9.5.5, *Concept* TemplateElement, p. 222.

9.5.5. Concept *TemplateElement*

Template elements are those elements a syntax template is build of by concatenation (see Section 9.5.4, *Concept* SyntaxTemplate, p. 221). The concept *TemplateElement* represents an abstract concept so as to be refined by the concrete concepts provided for template elements by *M2L*. A complete list of template elements will be shown in Table 9.6:

Template element	Description
ProperTerminal	terminal symbols not treated as whitespaces
WhitespaceTerminal	terminal symbols treated as whitespaces
NonTerminal	models a non-terminal by referring to abstract syntax
Option	models an option depending on an Edge Algebra predicate
Switch	models a set of alternative textual syntaxes
IncludeSyntaxDef	includes another textual syntax definition

Table 9.6.: Complete list of template elements for describing textual syntax in *M2L*

Abstract Syntax. The formal abstract syntax for *TemplateElement* is defined in Listing 9.99.

Listing 9.99: Abstract Syntax for *TemplateElement*

```
204 [TemplateElement!!] ;
```

TemplateElement is an abstract concept without any properties that have been additionally defined.

Concrete Syntax. No concrete syntax has to be defined for this abstract concept. Please refer to the (non-abstract) refining concepts.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts.

9.5.6. Concept *Terminal*

M2L distinguishes between two types of terminal symbols: proper terminals (see Section 9.5.7, *Concept* ProperTerminal, p. 223) and whitespace terminals (see Section 9.5.8, *Concept* WhitespaceTerminal, p. 224). These two types are grouped by this abstract concept *Terminal*. Please refer to those concepts for further details.

Abstract Syntax. The formal abstract syntax for *Terminal* is defined in Listing 9.100.

Listing 9.100: Abstract Syntax for *Terminal*

```
205 [Terminal!!] refines TemplateElement ;
```

Terminal is an abstract concept without any properties that have been additionally defined.

Concrete Syntax. No concrete syntax has to be defined for this abstract concept. Please refer to the (non-abstract) refining concepts, namely *ProperTerminal* and *WhitespaceTerminal*.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts, namely *ProperTerminal* and *WhitespaceTerminal*.

9.5.7. Concept *ProperTerminal*

The terminal symbols will be modelled by the concept *ProperTerminal*. Thus, each character must be encoded within a valid textual representation one by one. Terminal symbols may consist of arbitrary unicode characters but must contain at least one character. Keywords, but also braces etc., of a textual representation will be modelled by proper terminals.

Even whitespace characters, such as space or carriage return, will be allowed. Note that if such whitespace characters are used in proper terminals they are treated as any other terminal symbol: The (whitespace) characters must occur in exactly that form – i. e. such a whitespace must not be replaced by another whitespace or by a sequence of whitespaces. Whitespaces within proper terminals should therefore be rarely used. In order to use whitespaces in their traditional way, *WhitespaceTerminal* (see Section 9.5.8, *Concept* WhitespaceTerminal, p. 224) should be used instead.

Abstract Syntax. The formal abstract syntax for *ProperTerminal* is defined in Listing 9.101.

Listing 9.101: Abstract Syntax for *ProperTerminal*

```
206 ProperTerminal!! refines Terminal ::=
207   symbols[1..1] : C:String
208   where |P:symbols.P:character| > 0 ;
```

The properties of the concept *ProperTerminal* are:

- **symbols:** non-empty string of terminal symbols. Whitespace characters are generally allowed, but should be used rarely in this context as they are not treated as whitespaces. *WhitespaceTerminal* should be used instead.

Concrete Syntax. The formal textual concrete syntax for *ProperTerminal* is defined in Listing 9.102.

Listing 9.102: Textual Concrete Syntax for *ProperTerminal*

```
494 ProperTerminal: (P:symbols[QUOTED]) ;
```

Proper terminals are always encoded by a double-quoted string. The quote sign itself as well as the backslash are avoided by a preceding backslash: \" and \\.

Example. An example for the concept *ProperTerminal* will be provided in the following Listing 9.103:

Listing 9.103: Example for *ProperTerminal*

```
"person"
```

The example shows a proper terminal which models the keyword person, which has already been shown in example Listing 9.98.

9.5.8. Concept *WhitespaceTerminal*

An explicit whitespace can be modelled within a syntax template by the concept *WhitespaceTermial*. As described in Section 9.5.4, *Concept* SyntaxTemplate, p. 221, arbitrary whitespaces (namely spaces, carriage returns, and tabs) are allowed between each template element. Note that even an explicitly modelled whitespace does neither call for a particular whitespace nor for one at all. Thus an explicitly modelled whitespace terminal has no influence on the textual representations accepted by the specified language. Instead, it will define the behaviour when pretty-printing a model by the specified textual syntax, as only explicitly modelled whitespaces will be encoded.

Two types of whitespaces can be modelled: *Space* and *Newline*. While a *Space* will simply be encoded by a space, a more sophisticated behaviour will be defined for *Newline* as it supports an automatically correct indent. Note that although only space and newline can be explicitly defined, also tabs will be allowed as whitespaces during parsing.

Abstract Syntax. The formal abstract syntax for *WhitespaceTerminal* is defined in Listing 9.104.

Listing 9.104: Abstract Syntax for *WhitespaceTerminal*

```
209 WhitespaceTerminal!! refines Terminal ::=
210   whitespace[1..1] : C:Whitespace ;
```

The properties of the concept *WhitespaceTerminal* are:

- **whitespace:** whitespace modelled. Please refer to Section 9.5.9, *Concept* Whitespace, p. 225 for further details.

Concrete Syntax. The formal textual concrete syntax for *WhitespaceTerminal* is defined in Listing 9.105.

Listing 9.105: Textual Concrete Syntax for *WhitespaceTerminal*

```
495 WhitespaceTerminal: (P:whitespace) ;
```

A whitespace terminal will simply be encoded by its whitespace. Please refer to Section 9.5.9, *Concept* Whitespace, p. 225 for further details.

Example. An example for the concept *WhitespaceTerminal* will be provided in the following Listing 9.106:

Listing 9.106: Example for *WhitespaceTerminal*

```
_
```

The example shows a whitespace terminal which models a space that has already been shown in example Listing 9.98.

9.5.9. Concept *Whitespace*

The concept *Whitespace* enumerates the supported whitespaces which can be modelled within an explicit whitespace (see Section 9.5.8, *Concept* WhitespaceTerminal, p. 224).

Abstract Syntax. The formal abstract syntax for *Whitespace* is defined in Listing 9.107.

Listing 9.107: Abstract Syntax for *Whitespace*

```
211 enum Whitespace = { Space, Newline } ;
```

The enumeration values of the concept *ConceptType* are:

- **Space:** represents a space. A space will simply be encoded by a space.
- **Newline:** represents a newline. A newline will be encoded by a carriage return including the necessary tabs for realizing a correct indent.

Concrete Syntax. The formal textual concrete syntax for *Whitespace* is defined in Listing 9.108.

Listing 9.108: Textual Concrete Syntax for *Whitespace*

```
496 Whitespace.Space: "_" ;
497 Whitespace.Newline: "nl" ;
```

A space will be encoded by an underscore (_), whereas a newline will be encoded by **nl**.

Example. An example for the concept *Whitespace* will be provided in the following Listing 9.109:

Listing 9.109: Example for *Whitespace*

```
_
```

The example shows a whitespace representing a space which has already been used to encode a whitespace terminal in example Listing 9.106.

9.5.10. Concept *NonTerminal*

The concept *NonTerminal* models a non-terminal in terms of *M2L*, which is closely related to non-terminals in the context of production rules for grammars. In contrast to terminals, a non-terminal has to be replaced by another syntax rule which will, in turn, be defined by a textual syntax definition. That part of the model that has to be placed at the location of the non-terminal will be specified by an edge expression from Edge Algebra (see Section 9.7, *Package* ORG.Metamodels.EdgeAlgebra.EdgeExpressions, p. 235). The resulting nodes will then be encoded by the corresponding textual syntax definition depending on which concept is defined as the type for each node.

By default, a syntax definition having the same syntax identifier as the syntax definition this non-terminal is specified in, will be used. In case a syntax definition with another syntax identifier should be used, this could be modelled by a differing syntax (property *differingSyntax*).

Besides encoding the nodes referenced by an edge, it could also be indicated to encode a reference instead, by using a context-sensitive key. A context-sensitive key is a shortened canonical key depending on which node the referencing one is. Please refer to Section 7.5.5, *Local keys, namespaces and visibility*, p. 137 for a detailed description of context-sensitive keys.

The most usual expression is a *PropertyEdge* simply referencing a dedicated property of the currently encoded concept. Note that here, any kind of property can be referenced – independent on whether it is specified within a concept or not. This way of modelling corresponds to the fact that every property can be accessed – in the worst case, the evaluation of a property will result in an empty set. It can, however, be checked whether a referenced property is explicitly modelled for a concept or not; if not, a warning occurs.

Properties (and edges, generally spoken) are generally multi-valued. Thus, a non-terminal always encodes all elements one after the other. This way of defining can be seen as an implicit Kleene star for each non-terminal. Note that for defining concrete textual syntaxes in the context of *M2L* this will be sufficient as the valid multiplicity has already been defined by the abstract syntax.

In many cases multi-valued properties will be encoded by a special separator therebetween (such as a semicolon or comma), or a keyword will be encoded at the beginning. Non-terminals can therefore be configured by a set of additional syntax templates, namely:

- **Starting/ending.** The syntax templates for starting and ending will be encoded at the very beginning and the very ending of the encoded set of elements respectively, if, and only if, the property/edge consists of at least one element.
- **Prefix/suffix.** The syntax templates for prefix and suffix will be encoded at the beginning and ending of each encoded element respectively.
- **Infix.** The syntax template for the infix will be encoded inbetween each encoded element (including prefix and suffix).

Let value_of_element be the encoding for a single element of the non-terminal's property/edge. Then will the encoding scheme up to now be specified using EBNF statements in Listing 9.110 by coding_schema:

Listing 9.110: Simplified coding schema for non-terminals in EBNF

```
element       = prefix value_of_element suffix ;
coding_schema = [ starting element {infix element} ending ] ;
```

In the most general case, a non-terminal's property/edge equals a pomset – which may in particular contain duplicates and is partially ordered. Whereas an encoding of duplicates does not cause any problems, an additional extension for encoding a partial order will be required for encoding non-terminals. Depending on how the abstract syntax restricts the property/edge, three major cases will have to be distinguished:

1. **Property/edge must be totally ordered.** In this first case, no additional information will have to be encoded. The order in which the elements will occur in the textual representation will be interpreted as the order for the property/edge.
2. **Property/edge must be unordered.** In this second case, again no additional information will have to be encoded. The order of the elements will simply be ignored.
3. **Property/edge do not have to be totally ordered/unordered.** In this third case, additional information encoding the ordering will have to be encoded. Each element will therefore be marked by a numeric identifier within parentheses placed between the value of the element and the suffix. The identifier is a natural number beginning at 1 and will be assigned in the order in which the elements will occur within the textual representation. At the end of the enumeration of all elements, the (potentially partial) ordering will explicitly be encoded after a separating semicolon before the ending will be encoded. The partial ordering will be encoded in a transitively reduced, non reflexive way as already presented in Section 4.3, *Notations for pomsets*, p. 63. Two abbreviations will be provided for convenience: First, identifiers of elements that are not necessary within the explicit encoding of the ordering may be skipped; then the following elements will be assigned by decremented identifiers. Secondly, in case of a total ordering (also including singletons, hence sets with a cardinality less than or equal to 1) elements will be encoded without additional order information; the order will instead result from the order of the elements' occurrence as described in the first case. Totally unordered (multi-)sets of the order information in contrast cannot be omitted as otherwise it could not be distinguished between totally ordered and unordered (multi-)sets. The difference is, however, only an additional semicolon before the ending will be encoded, as the order is empty and thus no identifiers will be needed as well.

All in all can the encoding scheme be specified using EBNF statements in Listing 9.111 by coding_schema:

Listing 9.111: Full coding schema for non-terminals

```
element       = prefix value_of_element [ "(" id ")" ] suffix ;
coding_schema = [ starting element {infix element} ending
                  [ ";" { id "<" id } ] ] ;
```

Abstract Syntax. The formal abstract syntax for *NonTerminal* is defined in Listing 9.112.

Listing 9.112: Abstract Syntax for *NonTerminal*

```
212 NonTerminal!! refines TemplateElement ::=
213   edge[1..1] : C:Edge,
214   linkType[1..1] : C:LinkType,
215   differingSyntax[0..1] : C:SyntaxIdentifier,
216   starting[0..1] : C:SyntaxTemplate,
217   prefix[0..1] : C:SyntaxTemplate,
218   infix[0..1] : C:SyntaxTemplate,
219   suffix[0..1] : C:SyntaxTemplate,
```

```
220   ending[0..1] : C:SyntaxTemplate
221   where P:linkType ∈ C:LinkType.Reference ⇒
222     |P:differingSyntax| = 0 ;
```

The properties of the concept *NonTerminal* are:

- **edge:** describes the values which should be encoded by this non-terminal. It is recommended to reference properties directly by *PropertyEdging* (see Section 9.7.2, *Concept* ConstantEdge, p. 236) in order to accomplish parsability.
- **linkType:** indicates whether the elements of this non-terminal will be encoded inline or as a reference by a context-sensitive key (see Section 7.5.5, *Local keys, namespaces and visibility*, p. 137).
- **differingSyntax:** if set, another syntax identifier should be used than the one this non-terminal is defined for.
- **starting:** template element representing the *starting* for this non-terminal. The starting will only be encoded if the property/edge consists of at least one element.
- **prefix:** template element representing the *prefix* for this non-terminal. The prefix will be encoded preceding each element of the property/edge.
- **infix:** template element representing the *infix* for this non-terminal. The infix will be encoded between each of two consecutive elements of the property/edge.
- **suffix:** template element representing the *suffix* for this non-terminal. The suffix will be encoded after each element of the property/edge.
- **ending:** template element representing the *ending* for this non-terminal. The ending will only be encoded if the property/edge consists of at least one element.

Concrete Syntax. The formal textual concrete syntax for *NonTerminal* is defined in Listing 9.113.

Listing 9.113: Textual Concrete Syntax for *NonTerminal*

```
498 NonTerminal: "("
499   (P:starting | _ "||" _) (P:prefix | _ "|" _)
500   (P:linkType) (P:edge)
501   ("[" | P:differingSyntax | "]")
502   (_ "/" _ | P:infix)
503   (_ "|" _ | P:suffix) (_ "||" _ | P:ending)
504   ")" ;
```

Non-terminals will always be encoded within parentheses and will have at least one edge expression therein. As such an edge will normally be represented by a property edging as defined in Section 9.7.2, *Concept* ConstantEdge, p. 236, it will be encoded by a P: followed by the name of the property. A preceding ampersand (&) indicates that this non-terminal is encoded as a reference. If another syntax identifier should be used, this identifier will be encoded after the edge expression in squared brackets. The (optional) infix will then be encoded afterwards, separated by a slash (/). The (optional) prefix and suffix will then be encoded before and afterwards respectively, separated by a single pipe symbol (|). The (optional) starting and ending will finally be encoded before and afterwards respectively, separated by a double pipe symbol (||).

Example. An example for the concept *NonTerminal* will be provided in the following Listing 9.114:

Listing 9.114: Example for *NonTerminal*

```
("[" || "–" | &P:person / "," _ | "–" || "]")
```

The given example encodes a property *person* as a reference with an infix "," _, a prefix and suffix as "–", a starting "[", and an ending "]". Different exemplary encodings for different situations shall be provided in the following:

- If the abstract syntax restricts the property *person* to be totally ordered or unordered, an example will be [–Julian–, –Steve–, –Bill–].
- If the abstract syntax does not restrict the property *person* in such a way, an example for a partially ordered set will be [–Julian(1)–, –Steve(2)–, –Bill(3)– ; 3<1 3<2].
- According to abstract syntax, an example for a unordered set for the same situation will be [–Julian–, –Steve–, –Bill– ;]. Note that herein no numeric identifiers will be associated with the elements as the order is empty and thus no identifiers will be needed. Eventually, only the semicolon will remain at the end of the elements.
- According to abstract syntax, an example for a totally ordered set for the same situation will be [–Julian–, –Steve–, –Bill–]. Here, the abbreviation for totally ordered sets may be used.

9.5.11. Concept *Option*

The concept *Option* allows to model an alternative encoding depending on a predicate specified by an Edge Algebra statement. One option is perfectly suitable for representing boolean properties of a concept to be encoded. If the predicate equals *true*, the syntax template specified as the *thenCase* will have to be used; if the predicate equals *false*, the syntax template specified as the *elseCase* will have to be used. Whereas the then-case is mandatory, the else-case is optional. In case the else-case is missing, an empty syntax template will be assumed. Thus, if the predicate equals *false*, nothing will be encoded.

Abstract Syntax. The formal abstract syntax for *Option* is defined in Listing 9.115.

Listing 9.115: Abstract Syntax for *Option*

```
223 Option!! refines TemplateElement ::=
224   predicate[1..1] : C:Predicate,
225   thenCase[1..1] : C:SyntaxTemplate,
226   elseCase[0..1] : C:SyntaxTemplate ;
```

The properties of the concept *Option* are:

- **predicate:** describes that predicate which should be evaluated for this option. It is recommended to reference a boolean property directly by casting a *PropertyEdging* (see Section 9.7.2, *Concept* ConstantEdge, p. 236) to a boolean value by *CastEdgeToPredicate* (see Section 9.8.7, *Concept* SingleEdgeToPredicate, p. 255) in order to accomplish parsability.
- **thenCase:** template element representing the *thenCase* for this option. The then-case will be encoded if the predicate equals *true*.

- **elseCase:** template element representing the *elseCase* for this option. The else-case will be encoded if the predicate equals *false*.

Concrete Syntax. The formal textual concrete syntax for *Option* is defined in Listing 9.116.

Listing 9.116: Textual Concrete Syntax for *Option*

```
505 Option: "(" (P:predicate) _ "?" _ (P:thenCase)
506   (_ ":" _ | P:elseCase) ")" ;
```

Options will always be encoded within parentheses and will start with the Edge Algebra predicate followed by a question mark. The then-case will be encoded afterwards. If an else-case will be given, it will finally be encoded by a preceding colon.

Example. An example for the concept *Option* will be provided in the following Listing 9.117:

Listing 9.117: Example for *Option*

```
((bool)P:male ? "male" : "female")
```

This example models an option stating that the property *male* equals *true*; the syntax template **"male"** will be used for encoding, otherwise the syntax template **"female"** will be used for encoding.

9.5.12. Concept *Switch*

The concept *Switch* allows to model any set of alternative encodings, thus stating that a concept (or a dedicated part of a syntax template) may be represented in different ways. When a concept is encoded, that alternative resulting in the shortest representation will be used by default. (Note that this may even be non-deterministic as there may be two different alternatives resulting in representations with the same length.)

The concept *Switch* will also be used when syntax definitions are combined, e. g. when metamodels are combined or when multiple concepts are refined (see Section 9.4, *Package* ORG.Metamodels.M2L.ConcreteSyntax, p. 210).

Abstract Syntax. The formal abstract syntax for *Switch* is defined in Listing 9.118.

Listing 9.118: Abstract Syntax for *Switch*

```
227 Switch!! refines TemplateElement ::=
228   alternative[2..*](Set) : C:SyntaxTemplate ;
```

The properties of the concept *Switch* are:

- **alternative:** template elements representing alternative encodings for this switch. Note that at least two alternatives must be specified for one switch.

Concrete Syntax. The formal textual concrete syntax for *Switch* is defined in Listing 9.119.

Listing 9.119: Textual Concrete Syntax for *Switch*

```
507 Switch: "(" (P:alternative / _ "OR" _) ")" ;
```

Switches will always be encoded within parentheses and will simply enumerate the set of alternatives separated by an **OR** inbetween.

Example. An example for the concept *Switch* will be provided in the following Listing 9.120:

Listing 9.120: Example for *Switch*

```
( _ "add" _ OR "+")
```

The given example models a switch which allows two ways of encoding a summing operator: The first syntax template is _ "add" _, whereas the second syntax template is "+".

9.5.13. Concept *IncludeSyntaxDef*

Other syntax definitions are desired in many situations, so as to be re-used for defining new definitions. The concept *IncludeSyntaxDef* will therefore be introduced. There are two major use-cases for including another syntax definition:

1. **Including a syntax definition from *another* concept having *the same* syntax identifier.** This is often the case upon the introduction of the syntax definition of a refined concept. As multiple refinements are generally possible it will be necessary to define the concept name explicitly.
2. **Including a syntax definition from *the same* concept having *another* syntax identifier.** Upon the implementation of a correct bracketing, different syntax definitions for one concept will be defined. The only difference is that one concept will be bracketed, whereas the one will be not. In such a situation, the concept with the brackets can include the other one.

Both use-cases may even be combined by the concept *IncludeSyntaxDef*. It will therefore be possible to include a syntax definition with a syntax differing from another concept.

Abstract Syntax. The formal abstract syntax for *IncludeSyntaxDef* is defined in Listing 9.121.

Listing 9.121: Abstract Syntax for *IncludeSyntaxDef*

```
229 IncludeSyntaxDef!! refines TemplateElement ::=
230   concept[0..1] : C:Concept,
231   differingSyntax[0..1] : C:SyntaxIdentifier
232   where |P:concept| + |P:differingSyntax| ≥ 1 ;
```

The properties of the concept *IncludeSyntaxDef* are:

- **concept:** if set, a syntax definition of another concept than the one this include is defined for should be included.
- **differingSyntax:** if set, a syntax definition of a differing syntax identifier than the one this include is defined for should be included.

Concrete Syntax. The formal textual concrete syntax for *IncludeSyntaxDef* is defined in Listing 9.122.

Listing 9.122: Textual Concrete Syntax for *IncludeSyntaxDef*

```
508 IncludeSyntaxDef: "<" (&P:concept)
509   ("[" | P:differingSyntax | "]") ">" ;
```

Includes will always be encoded within angle brackets. Inbetween, the concept name will be denoted at first if a concept will be specified. Then – if specified – the differing syntax will be given within squared brackets. Note that at least either the concept name or the differing syntax must be specified.

Example. An example for the concept *IncludeSyntaxDef* will be provided in the following Listing 9.123:

Listing 9.123: Example for *IncludeSyntaxDef*

```
<Person["TEXTUAL_DEFAULT"]>
```

The given example models an include of the textual default syntax specification from the concept *Person*.

9.6. Package *ORG.Metamodels.EdgeAlgebra*

The package *ORG.Metamodels.EdgeAlgebra* including the three sub-packages thereof will provide the exact syntactical definition (including both abstract and concrete syntax) of the Edge Algebra as defined in Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 and used within *M2L* as described in this Chapter 9, *The overall specification of M2L*, p. 171. In particular the Edge Algebra will be used in the following six situations within *M2L*:

1. When specifying abstract syntaxes within the concept *Concept*, a predicate statement of the Edge Algebra will be used to specify additional constraints for the concept to be defined (see property *additionalConstraint* in Section 9.3.5, *Concept* ConceptDef, p. 192).
2. When specifying abstract syntaxes within the concept *PropertyDef*, an edge statement of the Edge Algebra will be used to specify the (context-sensitive) domain for a property (see property *domain* in Section 9.3.10, *Concept* PropertyDef, p. 201).
3. When specifying abstract syntaxes within the concept *PropertyDef*, an edge statement of the Edge Algebra will be used to specify an inferred value for a property (see property *inferredValue* in Section 9.3.10, *Concept* PropertyDef, p. 201).
4. When specifying abstract syntaxes within the concept *PropertyDef*, a predicate statement of the Edge Algebra will be used to specify the condition which must hold for this property definition to become relevant (see property *assumption* in Section 9.3.10, *Concept* PropertyDef, p. 201).
5. When specifying textual concrete syntaxes within the concept *NonTerminal*, an edge statement of the Edge Algebra will be used to describe the values which should be encoded by a non-terminal (see property *edge* in Section 9.5.10, *Concept* NonTerminal, p. 226).
6. When specifying textual concrete syntaxes within the concept *Option*, a predicate statement of the Edge Algebra will be used to describe the predicate which should be evaluated for an option (see property *predicate* in Section 9.5.11, *Concept* Option, p. 229).

This package contains its concepts indirectly via three sub-packages. These sub-packages will classify the operators according to their resulting type (thus the co-domain), namely edges, node predicates, and node valuations as listed in Table 9.7.

Section	Sub-package	Codomain
Section 9.7	EdgeExpressions	edges $\mathcal{E}_V = V \to \mathcal{P}_{pomset}(V)$
Section 9.8	PredicateExpressions	node predicates $\mathcal{B}_V = V \to \mathbb{B}$
Section 9.9	NumericalExpressions	node valuations $\mathcal{N}_V = V \to \mathbb{N}$

Table 9.7.: The sub-packages of *ORG.Metamodels.EdgeAlgebra*

All introduced operators have already formally been defined in Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95, except that functions defined for two operands will now be generalized to any number of at least two operands. This allows for many abbreviations, such as $a \leq b \leq c$. Please refer to the corresponding sections for the detailed meaning.

Remember that many edge operators are inferred from those for pomsets in Chapter 4, *Pomsets in the context of metamodelling*, p. 61. The operators herein are, however, defined for edges instead for pomsets which even allows for the use of those operators as an operand of

e. g. an edge inverse. As Edge Algebra is based on pomsets instead of sets, it is much more expressive than a set-based approach as it can handle both (partial) orders and duplicates. But pomset operators may also cause risks: Although the operators known for union ($\cup$), intersection ($\cap$) and subset ($\subset$) have been generalized for pomsets, these operators will now take care of the (partial) order by operating on connected components. This will in many cases result in an undesired behaviour and thus it is recommended to use the operators additive union ($\uplus$), projection ($\downarrow$), and consists-of ($\Subset$) instead.

In order to support a correct bracketing, an additional textual helper syntax, identified by *bracketed*, will be defined for the three Edge Algebra sub-packages. Whenever a sub-expression will have to be placed within brackets as the surrounding expression does not clarify the order of the operators (one example would be the concept *Addition*, see Section 9.9.4, *Concept* MultiNumericalOperator, p. 265), the syntax *bracketed* will be used. This helping syntax may be encoded in two different ways for the concepts: The first way allows an optional encoding within brackets as the dedicated sub-expression has already been encoded in a kind of bracketing way (such as the concept *Minimum*, see Section 9.9.4, *Concept* MultiNumericalOperator, p. 265). The second way calls for an encoding within brackets as the dedicated sub-expression has not been encoded within any kind of brackets (such as the concept *Addition*, see Section 9.9.4, *Concept* MultiNumericalOperator, p. 265). The definition of this helping syntax *Bracketed* will not be expressed explicitly in the following sections. Please refer to Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291 for the formal definition. Instead, a sentence such as the following will be found in the dedicated sections for the concrete syntax definition: *"In the helping syntax* Bracketed*, the parentheses will be mandatory/optional."* The Edge Algebra provides a single rule for operator priorities by the defined bracketing: Operators having a single operand (which will always be denoted in a prefix form) are stronger than operators having multiple operands (which will mostly be denoted in an infix form). For example: $(^\wedge a).b = {}^\wedge a.b \neq {}^\wedge(a.b)$ or $(\neg a) \vee b = \neg a \vee b \neq \neg(a \vee b)$.

The following three sections will be structured in an equal way: First of all, the abstract main concept will be given, representing all expressions evaluating to one of the three possible co-domains. An example will be the concept *Edge*. Then, the constant functions will be listed (such as *Self*), followed by the operators. Operators are those functions wherein the domain is of the same type as the co-domain (such as *Closure*). Finally, those functions will be enumerated wherein the domain of which is not of the same type as the co-domain. As in each case two other types will be possible for the domain, this final section will again be divided into two parts (such as represented by the concepts *PredicateToEdge* and *NumericalToEdge*). Except for the constant functions, it will additionally be distinguished between functions having a single operand (such as represented by the concept *SingleEdgeOperator*) and those functions having a number of operands (such as represented by the concept *MultiEdgeOperator*). In contrast to the previous sections of this chapter, the description of closely related concepts such as *Addition* and *Multiplication* will be pooled together in the following sections and described in a single subsection.

Note that the following sections define a huge set of operators which can even be expressed by each other. Both operators $<$ as well as $>$ and also $\leq$ and $\geq$ will, for example, be explicitly defined. Otherwise these operators cannot be used within the language which would result in a cumbersome way of writing down Edge Algebra expressions.

9.7. Package *ORG.Metamodels.EdgeAlgebra.EdgeExpressions*

This package includes all functions of Edge Algebra the co-domain thereof being an edge $\mathcal{E}_V = V \to \mathcal{P}_{pomset}(V)$. The constant edge functions together with the edge operators, except for *SubPomset*, will form the *Fundamental Edge Algebra*. These concepts will be marked by a $(*)$. All other functions including those of the two other packages for the Edge Algebra will extend the Fundamental Edge Algebra to form the *Propositional Edge Algebra*. Table 9.8 shows the list of all concepts defined:

Concept	Description
Edge	expression evaluating to an edge
ConstantEdge	expression without any parameters
EdgeValue$^{(*)}$	a dedicated pomset with concrete values
ConceptEdging$^{(*)}$	all nodes of which the type is a given (or sub-) concept
TypeEdging$^{(*)}$	all nodes of which the type is exactly a given concept
PropertyEdging$^{(*)}$	the edge for a specific property
Self$^{(*)}$	edge where each node is mapped to itself
Equality$^{(*)}$	edge where each node is mapped to all equal nodes
Successor$^{(*)}$	successors for a node independent of a property
BoundedEdgeVariable	bounded variable e. g. within a universal quantifier
EdgeOperator	operator having at least one edge operand
SubPomset	extracting elements of a pomset with a certain depth
SingleEdgeOperator	operator having exactly one edge operand
First$^{(*)}$	set of all smallest elements out of a pomset
Closure$^{(*)}$	transitive reflexive closure of a given edge
EdgeInverse$^{(*)}$	inverse edge of a given edge
OrderInverse$^{(*)}$	the order of a pomset is inverted
OrderDestroy$^{(*)}$	the order of a pomset is eliminated
DuplicateDestroy$^{(*)}$	the duplicate elements of a pomset are eliminated
MultiEdgeOperator	operator having at least two edge operands
Navigation$^{(*)}$	navigation over a list of edges
AdditiveUnion$^{(*)}$	additive union over (pomset-valued) edges
Concatenation$^{(*)}$	concatenation over (pomset-valued) edges
Projection$^{(*)}$	projection over (pomset-valued) edges
Difference$^{(*)}$	difference over (pomset-valued) edges
Union$^{(*)}$	union over (pomset-valued) edges
Intersection$^{(*)}$	intersection over (pomset-valued) edges
PredicateToEdge	function having at least one predicate operand
SinglePredicateToEdge	function having exactly one predicate operand
CastPredicateToEdge	cast a node predicate to a boolean valued edge
PredicateSelection	select all nodes for which the given predicate holds
NumericalToEdge	function having at least one numerical operand
SingleNumericalToEdge	function having exactly one numerical operand
CastNumericalToEdge	cast a node valuation to a natural number valued edge
NumericalSelection	select the nodes in the quantity of node valuation

Table 9.8.: List of concepts defined in *ORG.Metamodels.EdgeAlgebra.EdgeExpressions*

9.7.1. Concept *Edge*

The abstract concept *Edge* represents all Edge Algebra expressions evaluating to an edge and will thus be refined by each of the dedicated concepts. Hence, the concept *Edge* stands for all functions and operators the co-domain thereof being $\mathcal{E}_V = V \rightarrow \mathcal{P}_{pomset}(V)$. Note that according to this, the co-domain of such an Edge Algebra expression will formally again be a function mapping each node of an M-graph to a pomset of nodes from the same M-graph.

Abstract Syntax. The formal abstract syntax for *Edge* is defined in Listing 9.124.

Listing 9.124: Abstract Syntax for *Edge*

```
234 [Edge!!] ;
```

Edge is an abstract concept without any properties that have been additionally defined. It is prepared to be refined by concepts representing dedicated Edge Algebra statements.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *Edge*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will nonetheless be defined as a default in which the parentheses will be optional. Please refer to Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291 for a detailed definition of the helping syntax *Bracketed*.

Example. An example for the concept *Edge* will be provided in the following Listing 9.125:

Listing 9.125: Example for *Edge*

```
≳^⇆P:composite.1 P:lkey
```

The given example illustrates an edge calculating the canonical key for each node as defined in Section 7.5.5, *Local keys, namespaces and visibility*, p. 137. Note that due to the stronger binding of single operand operators the bracketing will have to be read as (≳(^(⇆(P:composite)))).(1(P:lkey)). When keywords are used instead of symbols, this results in (**orderInv**(**closure**(**edgeInv**(P:composite)))).(**first**(P:lkey)). The brackets can, of course, be omitted in this version anyway: **orderInv closure edgeInv** P:composite.**first** P:lkey. Please refer to the dedicated sections for a detailed definition of the several operators.

9.7.2. Concept *ConstantEdge*

The concept *ConstantEdge* represents all Edge Algebra expressions evaluating to an edge but having no operands as operands are elements from the three Edge Algebra carrier sets. Constant edges may, however, comprise a parameter. Five constant edges will be defined in total:

- **EdgeValue** allows for the definition of a constant pomset.
- **ConceptEdging** defines a set of all nodes having the type of a given concept (including refined concepts).

- **TypeEdging** defines a set of all nodes having the type of a given concept (excluding refined concepts).
- **PropertyEdging** defines an edge according to a given property.
- **Self** defines an edge mapping each node to itself.
- **Equality** defines an edge mapping each node to all equal nodes. For details please refer to Section 5.4, *Node equivalence*, p. 86.
- **Successor** defines an edge resulting from an additive union of all properties.
- **BoundedEdgeVariable** represents the bounded variable within the quantified edging and the selection operator. Please refer to concept *QuantifiedEdging* (in Section 9.8.3, *Concept* PredicateOperator, p. 250), *PredicateSelection* (in Section 9.8.1, *Concept* Predicate, p. 249), and *NumericalSelection* (in Section 9.9.1, *Concept* Numerical, p. 263) as well.

Abstract Syntax. The formal abstract syntax for *ConstantEdge* is defined in Listing 9.126.

Listing 9.126: Abstract Syntax for *ConstantEdge*

```
241 [ConstantEdge!!] refines Edge ;
242 EdgeValue!! refines ConstantEdge ::=
243   &value[0..*] ;
244 ConceptEdging!! refines ConstantEdge ::=
245   concept[1..1] : C:Concept ;
246 TypeEdging!! refines ConstantEdge ::=
247   concept[1..1] : C:Concept ;
248 PropertyEdging!! refines ConstantEdge ::=
249   property[1..1] : C:Property ;
250 Self!! refines ConstantEdge ::= ;
251 Equality!! refines ConstantEdge ::= ;
252 Successor!! refines ConstantEdge ::= ;
253 BoundedEdgeVariable!! refines ConstantEdge ::=
254   identifier[1..1] : C:Identifier ;
```

The properties of the concept *EdgeValue* are:

- **value:** value defining the constant pomset which may even be an empty set.

The properties of the concept *ConceptEdging* and *TypeEdging* are:

- **concept:** concept for which the nodes should be selected.

The properties of the concept *PropertyEdging* are:

- **property:** property for which the edge should be returned.

The properties of the concept *BoundedEdgeVariable* are:

- **identifier:** identifier of the bounded edge variable.

Concrete Syntax. The formal textual concrete syntax for *ConstantEdge* is defined in Listing 9.127.

Listing 9.127: Textual Concrete Syntax for *ConstantEdge*

```
EdgeValue: "{{" _
  ((P:value["TEXTUAL_UNIQUE"] / "," _)
   OR (&P:value / "," _)) _ "}}" ;
ConceptEdging: ("C:" | P:concept) ;
TypeEdging: ("T:" | P:concept) ;
PropertyEdging: ("P:" | P:property) ;
Self: ("↺" OR "self") ;
Equality: ("↪" OR "equality") ;
Successor: "*" ;
BoundedEdgeVariable: (P:identifier) ;
```

An edge value will be encoded by double curly brackets {{ }}. Values may be coded as references or inline. If the values are encoded inline, the syntax *UNIQUE* will be used to ensure that each and every concept can be encoded in a unique way. The syntax *UNIQUE* encodes the default syntax within parentheses with the concept name as a prefix. As the value may be a real partially ordered pomset, refer to Section 9.5.10, *Concept* NonTerminal, p. 226 for the detailed encoding.

The concept edging will be encoded by a preceding C: ; the type edging by a preceding T: ; the property edging in contrast will be encoded by a preceding P: . The reflexive operator will be encoded by the symbol ↺ or the keyword self. The equality operator will be encoded by the symbol ↪ or the keyword equality. The successor will be encoded by a star ∗. The bounded edge variable e. g. within a universal quantifier will simply be encoded by the identifier itself.

Example. An example for the concept *ConstantEdge* will be provided in the following Listing 9.128:

Listing 9.128: Example for *ConstantEdge*

```
{{ Identifier(id)(1), String("str")(2),
   Natural(10)(3) ; 1<2 1<3 }}
```

The example shows an edge value representing the following pomset: $\left\{ \begin{array}{l} \text{id} \nearrow \text{"str"} \\ \text{id} \searrow 10 \end{array} \right\}$

9.7.3. Concept *EdgeOperator*

The concept *EdgeOperator* represents all Edge Algebra expressions both operating on at least one edge and evaluating to an edge. The edge operators will be categorised into three classes: those which operate on exactly one operand (see Section 9.7.4, *Concept* SingleEdgeOperator, p. 240), those which operate on at least two operands (see Section 9.7.5, *Concept* MultiEdgeOperator, p. 241), and the special case *SubPomset*, as it comprises two numerical values in addition to the edge operand and will thus be described directly in this section. Please refer to Chapter 4, *Pomsets in the context of metamodelling*, p. 61 and Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition of the sub-pomset operator.

Abstract Syntax. The formal abstract syntax for *EdgeOperator* is defined in Listing 9.129.

Listing 9.129: Abstract Syntax for *EdgeOperator*

```
256 [EdgeOperator!!] refines Edge ::>
257   edgeOperand[1..*](List) : C:Edge ;
258 SubPomset!! refines EdgeOperator ::=
259   edgeOperand[1..1],
260   minDepth[1..1] : C:Numerical,
261   maxDepth[0..1] : C:Numerical ;
```

The properties of the concept *EdgeOperator* are:

- **edgeOperand:** list of edge operands for an edge operator. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands.

The properties of the concept *SubPomset* are:

- **edgeOperand (refined):** edge operand for the sub-pomset operator.
- **minDepth:** minimal depth required for an element of the given pomset in order to be included in the result. Note that the depth of the smallest elements is zero.
- **maxDepth:** maximal depth required for an element of the given pomset in order to be included in the result. This operand is optional. If the maximum depth is missing, the depth of the resulting elements will have no upper limit. Note that the depth of the smallest elements is zero.

Concrete Syntax. The formal textual concrete syntax for *EdgeOperator* is defined in Listing 9.130.

Listing 9.130: Textual Concrete Syntax for *EdgeOperator*

```
527 SubPomset: (
528     "[" (P:minDepth) ("," | P:maxDepth) "]"
529     (P:edgeOperand[Bracketed]) ;
530   OR
531     "subpomset" "(" (P:edgeOperand)
532     ("," _ | P:minDepth) ("," _ | P:maxDepth) ")"
533   ) ;
```

No concrete syntax has to be defined for the abstract concept *EdgeOperator*. Please refer to the (non-abstract) refining concepts.

Two alternative encodings will be provided for the concept *SubPomset*: The first alternative encodes the sub-pomset operator like a parametrized prefix operator: In front of the edge operand both the minimal and the optional maximal depth will be notated within square brackets and are separated by a colon. The second alternative encodes the sub-pomset operator in a standard functional notation: The keyword **subpomset** will be encoded at first, followed by the operators within parentheses separated by colons starting with the edge operand, then the minimal depth and finally the optional maximum depth.

The helping syntax *Bracketed* for both concepts *EdgeOperator* and *SubPomset* will remain unchanged and thus the parentheses are optional.

Example. An example for the concept *EdgeOperator* will be provided in the following Listing 9.131:

Listing 9.131: Example for *EdgeOperator*

```
[1,2]^P:child
```

The given example will return all children and grand children. Although the closure operator is reflexive, the *self* node will not be returned, as the minimum depth has been set to 1 instead of 0.

9.7.4. Concept *SingleEdgeOperator*

The concept *SingleEdgeOperator* represents all edge operators both operating on and evaluating to a single edge. Six edge operators operating on one edge will be defined in total:

- **First** returns the smallest elements of a (pomset-valued) edge.
- **Closure** returns the ordered reflexive-transitive closure of the given edge. Note that this operator preserves both order and duplicates. The descend will additionally be recorded within the order. By a special loop cutting definition, infinite pomsets will be avoided.
- **EdgeInverse** returns the inverse edge of the given one. Duplicates will be preserved. Note that inverse edges will always be unordered and thus be multi-sets.
- **OrderInverse** returns an edge in which the order of the pomsets is inverted.
- **OrderDestroy** returns an edge in which the order of the pomsets is eliminated.
- **DuplicateDestroy** returns an edge in which the duplicates of the pomsets are eliminated.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 and Chapter 4, *Pomsets in the context of metamodelling*, p. 61 for a detailed definition.

Abstract Syntax. The formal abstract syntax for *SingleEdgeOperator* is defined in Listing 9.132.

Listing 9.132: Abstract Syntax for *SingleEdgeOperator*

```
263 [SingleEdgeOperator!!] refines EdgeOperator ::=
264   edgeOperand[1..1] ;
265 First!! refines SingleEdgeOperator ;
266 Closure!! refines SingleEdgeOperator ;
267 EdgeInverse!! refines SingleEdgeOperator ;
268 OrderInverse!! refines SingleEdgeOperator ;
269 OrderDestroy!! refines SingleEdgeOperator ;
270 DuplicateDestroy!! refines SingleEdgeOperator ;
```

The properties of the concept *SingleEdgeOperator* are:

- **edgeOperand (refined):** edge operand for a single-valued edge operator.

All refinements of the concept *SingleEdgeOperators* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *SingleEdgeOperator* is defined in Listing 9.133.

Listing 9.133: Textual Concrete Syntax for *SingleEdgeOperator*

```
First: ("1" OR "first") _ (P:edgeOperand[Bracketed]) ;
Closure: ("^" OR "closure" _)
  (P:edgeOperand[Bracketed]) ;
EdgeInverse: ("⇆" OR "edgeInv" _)
  (P:edgeOperand[Bracketed]) ;
OrderInverse: ("⩾" OR "orderInv" _)
  (P:edgeOperand[Bracketed]) ;
OrderDestroy: ("μ" OR "orderDest" _)
  (P:edgeOperand[Bracketed]) ;
DuplicateDestroy: ("ϵ" OR "dupDest" _)
  (P:edgeOperand[Bracketed]) ;
```

All single-valued edge operators will be denoted in a prefix form. There are two ways of encoding for each operator. One is a short symbolic way, the other one uses a keyword.

- For the **first** operator, the symbol will be 1 and the keyword will be **first**.
- For the **closure** operator, the symbol will be ^ and the keyword will be **closure**.
- For the **edge-inverse** operator, the symbol will be ⇆ and the keyword will be **edgeInv**.
- For the **order-inverse** operator, the symbol will be ⩾ and the keyword will be **orderInv**.
- For the **order-destroy** operator, the symbol will be μ and the keyword will be **orderDest**.
- For the **duplicate-destroy** operator, the symbol will be ϵ and the keyword will be **dupDest**.

Example. An example for the concept *SingleEdgeOperator* will be provided in the following Listing 9.134:

Listing 9.134: Example for *SingleEdgeOperator*

```
⇆P: composite
```

The given example forms the edge-inverse of the property *composite* and thus calculates the parent of a node with respect to compositions.

9.7.5. Concept *MultiEdgeOperator*

The concept *MultiEdgeOperator* represents all edge operators operating on a list of edges and evaluating to a single edge. Seven edge operators operating on multiple edges will be defined in total:

- **Navigation** navigates over the given list of edges. The navigation operator preserves both order and duplicates and may thus result in a really partially ordered pomset.

- **AdditiveUnion** returns an edge in which the pomsets are additively united. Thus, no additional order will be added between elements of the different pomsets from the operands.
- **Concatenation** returns an edge in which the pomsets are concatenated. Thus, the elements from the different operands will be ordered according to the order of the operands.
- **Difference** returns an edge in which the difference of the pomsets is built. This results in the edge given by the first operator, except for those pomset elements occurring in one of the other operands.
- **Union** returns an edge in which the pomsets are united, based on the connected components of the pomsets.
- **Intersection** returns an edge in which the pomsets are intersected, based on the connected components of the pomsets.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 and Chapter 4, *Pomsets in the context of metamodelling*, p. 61 for a detailed definition. In contrast to the definitions in the sections herein, the operators will be defined on any number greater than or equal to two instead of on exactly two operands. The present extension can be reduced to the basic definition by the following equation 9.4, wherein op will be one of the operators mentioned above:

$$x_1 \text{ op } x_2 \text{ op } x_3 \text{ op } \ldots \text{ op } x_n = (\ldots((x_1 \text{ op } x_2) \text{ op } x_3) \text{ op } \ldots) \text{ op } x_n \tag{9.4}$$

Abstract Syntax. The formal abstract syntax for *MultiEdgeOperator* is defined in Listing 9.135.

Listing 9.135: Abstract Syntax for *MultiEdgeOperator*

```
272 [MultiEdgeOperator!!] refines EdgeOperator ::=
273   edgeOperand[2..*] ;
274 Navigation!! refines MultiEdgeOperator ;
275 AdditiveUnion!! refines MultiEdgeOperator ;
276 Concatenation!! refines MultiEdgeOperator ;
277 Projection!! refines MultiEdgeOperator ;
278 Difference!! refines MultiEdgeOperator ;
279 Union!! refines MultiEdgeOperator ;
280 Intersection!! refines MultiEdgeOperator ;
```

The properties of the concept *MultiEdgeOperator* are:

- **edgeOperand (refined):** edge operands for a multi-valued edge operator which must be at least two.

All refinements of the concept *MultiEdgeOperator* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *MultiEdgeOperator* is defined in Listing 9.136.

Listing 9.136: Textual Concrete Syntax for *MultiEdgeOperator*

```
547 Navigation: (P:edgeOperand[Bracketed] / ".") ;
548 AdditiveUnion: (P:edgeOperand[Bracketed]
549   / _ ("⊎" OR "addUnion") _) ;
550 Concatenation: (P:edgeOperand[Bracketed]
551   / _ ("⊕" OR "concat") _) ;
552 Projection: (P:edgeOperand[Bracketed]
553   / _ ("↓" OR "projectOn") _ ) ;
554 Difference: (P:edgeOperand[Bracketed] / _ "\\" _ ) ;
555 Union: (P:edgeOperand[Bracketed]
556   / _ ("∪" OR "union") _) ;
557 Intersection: (P:edgeOperand[Bracketed]
558   / _ ("∩" OR "intersect") _) ;
```

All multi-valued edge operators will be denoted in an infix form. There are two ways of encoding for most of the operators. One is a short symbolic way, the other one uses a keyword.

- For the **navigation** operator, the symbol will be a dot (.). An alternative keyword will not be defined.
- For the **additive-union** operator, the symbol will be ⊎ and the keyword will be **addUnion**.
- For the **concatenation** operator, the symbol will be ⊕ and the keyword will be **concat**.
- For the **projection** operator, the symbol will be ↓ and the keyword will be **projectOn**.
- For the **difference** operator, the symbol will be a backslash ($\downarrow^*$). An alternative keyword will not be defined. Note that the backslash will be doubled within the formal syntax definition as a backslash must be escaped when encoding a string and thus a terminal symbol.
- For the **union** operator, the symbol will be ∪ and the keyword will be **union**.
- For the **intersection** operator, the symbol will be ∩ and the keyword will be **intersect**.

Example. An example for the concept *MultiEdgeOperator* will be provided in the following Listing 9.137:

Listing 9.137: Example for *MultiEdgeOperator*

```
^P:composite ↓ C:Person ↓ (P:employee ⊎ P:retired)
```

The given example illustrates the projection operator having three operands: The first one is the closure of the property *composite*; the second one is the concept edge for *Person* which results in the set of all persons; and the third one is an additive union of the properties *employee* and *retired*. In total the expression returns all nodes that are transitively reachable via compositions, that are persons and that are referenced as an employee or a retired person.

9.7.6. Concept *PredicateToEdge*

The concept *PredicateToEdge* represents all Edge Algebra expressions evaluating to an edge but operating on (at least) one node predicate instead of edges. The available edge expres-

sions operating on node predicates are all of the same kind: They all operate on a single node predicate (see Section 9.7.7, *Concept* SinglePredicateToEdge, p. 244).

Abstract Syntax. The formal abstract syntax for *PredicateToEdge* is defined in Listing 9.138.

Listing 9.138: Abstract Syntax for *PredicateToEdge*

```
[PredicateToEdge!!] refines Edge ::>
  predicateOperand[1..*](List) : C:Predicate ;
```

The properties of the concept *PredicateToEdge* are:

- **predicateOperand:** list of node predicate operands for an edge expression operating on node predicates. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands. Note that the second variant has never been used yet.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *PredicateToEdge*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will remain unchanged and thus the parentheses will be optional.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts.

9.7.7. Concept *SinglePredicateToEdge*

The concept *SinglePredicateToEdge* represents all Edge Algebra expressions evaluating to an edge and operating on exactly one node predicate. Two edge expressions operating on one node predicate will be defined in total:

- **CastPredicateToEdge** returns an edge with a single-valued pomset directing towards either *true* or *false* according to the node predicate given.
- **PredicateSelection** returns an edge directing towards all nodes for which the node predicate evaluates to *true*.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition.

Abstract Syntax. The formal abstract syntax for *SinglePredicateToEdge* is defined in Listing 9.139.

Listing 9.139: Abstract Syntax for *SinglePredicateToEdge*

```
[SinglePredicateToEdge!!] refines PredicateToEdge ::=
  predicateOperand[1..1] ;
CastPredicateToEdge!! refines SinglePredicateToEdge ;
PredicateSelection!! refines SinglePredicateToEdge ::=
  boundedVariable[0..1] : C:Identifier ;
```

The properties of the concept *SinglePredicateToEdge* are:

- **predicateOperand (refined):** node predicate operand for a single-valued edge expression operating on node predicates.

The additional properties of the concept *PredicateSelection* are:

- **boundedVariable:** the identifier for the bounded edge variable.

All other refinements of the concept *SinglePredicateToEdge* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *SinglePredicateToEdge* is defined in Listing 9.140.

Listing 9.140: Textual Concrete Syntax for *SinglePredicateToEdge*

```
560 CastPredicateToEdge:
561   "(edge)" (P:predicateOperand[Bracketed]) ;
562 PredicateSelection: ("σ" OR "select")
563   (_ | P:boundedVariable | ":")
564   (_ | P:predicateOperand[Bracketed]) ;
```

All single-valued edge expressions operating on node predicates will be denoted in a prefix form. There are two ways of encoding for some operators. One is a short symbolic way, the other one uses a keyword.

- For the **cast** operator, the keyword will be (**edge**). An alternative symbol will not be defined.
- For the **select** operator, the symbol will be σ and the keyword will be **select**.

Example. An example for the concept *SinglePredicateToEdge* will be provided in the following Listing 9.141:

Listing 9.141: Example for *SinglePredicateToEdge*

```
σ|⇆P:composite|=0
```

The given example shows the predicate select operator. The expression will result in all nodes that do not have a compositional parent and will thus be root nodes. Please refer to Section 9.8.10, *Concept* MultiNumericalToPredicate, p. 260 and Section 9.9.6, *Concept* SingleEdgeToNumerical, p. 268 respectively for the equation and cardinality operator.

9.7.8. Concept *NumericalToEdge*

The concept *NumericalToEdge* represents all Edge Algebra expressions evaluating to an edge but operating on (at least) one node valuation instead of edges. The available edge expressions operating on node valuations are all of the same kind: They all operate on a single node valuation (see Section 9.7.9, *Concept* SingleNumericalToEdge, p. 246).

Abstract Syntax. The formal abstract syntax for *NumericalToEdge* is defined in Listing 9.142.

Listing 9.142: Abstract Syntax for *NumericalToEdge*

```
[NumericalToEdge!!] refines Edge ::>
  numericalOperand[1..*](List) : C:Numerical ;
```

The properties of the concept *NumericalToEdge* are:

- **numericalOperand:** list of node valuation operands for an edge expression operating on node valuations. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands. Note that the second variant has never been used yet.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *NumericalToEdge*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will remain unchanged and thus the parentheses will be optional.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts.

9.7.9. Concept *SingleNumericalToEdge*

The concept *SingleNumericalToEdge* represents all Edge Algebra expressions evaluating to an edge and operating on exactly one node valuation. Two edge expressions operating on one node valuation will be defined in total:

- **CastNumericalToEdge** returns an edge with a single-valued pomset directing towards a natural number according to the node valuation given.
- **NumericalSelection** returns an edge with a multi-set containing the nodes in the quantity of node valuation.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition.

Abstract Syntax. The formal abstract syntax for *SingleNumericalToEdge* is defined in Listing 9.143.

Listing 9.143: Abstract Syntax for *SingleNumericalToEdge*

```
[SingleNumericalToEdge!!] refines NumericalToEdge ::=
  numericalOperand[1..1] ;
CastNumericalToEdge!! refines SingleNumericalToEdge ;
NumericalSelection!! refines SingleNumericalToEdge ::=
  boundedVariable[0..1] : C:Identifier ;
```

The properties of the concept *SingleNumericalToEdge* are:

- **numericalOperand (refined):** node valuation operand for a single-valued edge expression operating on node valuations.

The additional properties of the concept *NumericalSelection* are:

- **boundedVariable:** the identifier for the bounded edge variable.

All other refinements of the concept *SingleNumericalToEdge* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *SingleNumericalToEdge* is defined in Listing 9.144.

Listing 9.144: Textual Concrete Syntax for *SingleNumericalToEdge*

```
566 CastNumericalToEdge:
567   "(edge)" (P:numericalOperand[Bracketed]) ;
568 NumericalSelection: ("σ" OR "select" _)
569   (P:boundedVariable | ":")
570   (P:numericalOperand[Bracketed]) ;
```

All single-valued edge expressions operating on node valuations will be denoted in a prefix form. There are two ways of encoding for some operators. One is a short symbolic way, the other one uses a keyword.

- For the **cast** operator, the keyword will be (**edge**). An alternative symbol will not be defined.
- For the **select** operator, the symbol will be σ and the keyword will be **select**.

Example. In the following Listing 9.145 we give an example for the concept *SingleNumericalToEdge*:

Listing 9.145: Example for *SingleNumericalToEdge*

```
σ(number)P:quantity
```

The given example shows the numerical select operator. The expression will result in a multiset containing the nodes with a multiplicity according to the property *quantity*. Please refer to Section 9.9.6, *Concept* SingleEdgeToNumerical, p. 268 for the cast to a numerical.

9.8. Package *ORG.Metamodels.EdgeAlgebra.PredicateExpressions*

This package includes all functions of Edge Algebra the co-domain thereof being a node predicate $\mathcal{B}_V = V \to \mathbb{B}$. All functions defined herein are part of the *Propositional Edge Algebra*. Table 9.9 and Table 10 show the list of all concepts defined:

Concept	Description
Predicate	expression evaluating to a node predicate
ConstantPredicate	expression without any parameters
PredicateValue	a boolean value *true* or *false*
IsRoot	holds for nodes which do not have a parent node
PredicateOperator	operator having at least one node predicate operand
QuantifiedConceptEdging	the universal quantification over (sub-)concepts
QuantifiedTypeEdging	the universal quantification over the exact type
QuantifiedPropertyEdging	the universal quantification over properties
SinglePredicateOperator	operator having exactly one node predicate operand
Not	negates the given predicate
MultiPredicateOperator	operator having at least two node predicate operands
And	conjunction of the given set of predicates
Or	disjunction of the given set of predicates
Xor	exclusive disjunction of the given set of predicates
Iff	if-and-only-if for the given set of predicates
Implies	implication for the given list of predicates
EdgeToPredicate	predicate having at least one edge operand
SingleEdgeToPredicate	predicate having exactly one edge operand
CastEdgeToPredicate	cast a boolean valued edge to a node predicate
IsEmpty	holds for nodes wherein the edge is empty
IsSingleton	holds for nodes wherein the edge has one element at most
IsSet	holds for nodes wherein the edge is a unordered set
IsBag	holds for nodes wherein the edge is unordered
IsList	holds for nodes wherein the edge is totally ordered
IsToset	holds for nodes wherein the edge is a totally ordered set
IsPoset	holds for nodes wherein the edge has no duplicates
MultiEdgeToPredicate	predicate having at least two edge operands
EdgeEqual	holds for nodes wherein the edges are equal
EdgeNotEqual	holds for nodes wherein the edges are pairwise unequal
ConsistsOf	node predicate according to the $\Subset$ operator
NotConsistsOf	node predicate according to the $\Subsetneq$ operator
Subset	node predicate according to the $\subset$ operator
NotSubset	node predicate according to the $\not\subset$ operator
SubsetOrEqual	node predicate according to the $\subseteq$ operator
NotSubsetOrEqual	node predicate according to the $\not\subseteq$ operator

Table 9.9.: List of concepts defined in *ORG.Metamodels.EdgeAlgebra.PredicateExpressions* (part 1)

Concept	Description
NumericalToPredicate	predicate having at least one node valuation operand
MultiNumericalToPredicate	predicate having at least two node valuation operands
NumericalEqual	holds for nodes wherein the node valuations are equal
NumericalNotEqual	holds for nodes wherein the node val. are pairwise unequal
LessOrEqual	node predicate according to the $\leq$ operator
LessThan	node predicate according to the $<$ operator
GreaterOrEqual	node predicate according to the $\geq$ operator
GreaterThan	node predicate according to the $>$ operator

Table 9.10.: List of concepts defined in *ORG.Metamodels.EdgeAlgebra.PredicateExpressions* (part2)

9.8.1. Concept *Predicate*

The abstract concept *Predicate* represents all Edge Algebra expressions evaluating to a node predicate and thus being refined by each of the dedicated concepts. The concept *Predicate* thus represents all functions and operators the co-domain thereof being $\mathcal{B}_V = V \to \mathbb{B}$. Note that according to this, the co-domain of such an Edge Algebra expression will formally again be a function mapping each node of an M-graph to a boolean value.

Abstract Syntax. The formal abstract syntax for *Predicate* is defined in Listing 9.146.

Listing 9.146: Abstract Syntax for *Predicate*

```
302 [Predicate!!] ;
```

Predicate is an abstract concept without any properties that have been additionally defined. It is prepared to be refined by concepts representing dedicated Edge Algebra statements.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *Predicate*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will nonetheless be defined as a default in which the parentheses will be optional. Please refer to Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291 for a detailed definition of the helping syntax *Bracketed*.

Example. An example for the concept *Predicate* will be provided in the following Listing 9.147:

Listing 9.147: Example for *Predicate*

```
(root? ⇔ |⇆P:composite|=0) ∧ ↻ ∉ P:composite.^P:composite
```

The given example illustrates a node predicate that holds for every valid M-graph. The left side of the conjunction provides an exact definition of the node predicate root?: For every node *IsRoot* will evaluate to *true*, if, and only if, the node has no compositional parent. The right side of the conjunction states that the property *composite* must not contain cycles, i. e. a node itself must not be part of the (non-reflexive) transitive closure of the property *composite*. Please refer to the dedicated sections for a detailed definition of the several operators.

9.8.2. Concept *ConstantPredicate*

The concept *ConstantPredicate* represents all Edge Algebra expressions evaluating to a node predicate but having no operands as operands are elements from the three Edge Algebra carrier sets. Constant edges may, however, comprise a parameter. Two constant node predicates will be defined in total:

- **PredicateValue** allows for a definition of a constant node predicate by a boolean value.
- **IsRoot** evaluates to *true* if the node does not have a compositional parent.

Abstract Syntax. The formal abstract syntax for *ConstantPredicate* is defined in Listing 9.148.

Listing 9.148: Abstract Syntax for *ConstantPredicate*

```
304 [ConstantPredicate!!] refines Predicate ;
305 PredicateValue!! refines ConstantPredicate ::=
306   value[1..1] : C:Boolean ;
307 IsRoot!! refines ConstantPredicate ::= ;
```

The properties of the concept *PredicateValue* are:

- **value:** constant boolean value assigned to each node.

Concrete Syntax. The formal textual concrete syntax for *ConstantPredicate* is defined in Listing 9.149.

Listing 9.149: Textual Concrete Syntax for *ConstantPredicate*

```
574 PredicateValue: (P:value) ;
575 Root: "root?" ;
```

An node predicate value will be encoded by simply encoding a boolean value, e. g. **true** or **false**. The symbols $\top$ and $\bot$ may also be used instead. The concept *IsRoot* will be encoded by **root**?.

Example. An example for the concept *ConstantPredicate* will be provided in the following Listing 9.150:

Listing 9.150: Example for *ConstantPredicate*

```
root?
```

The example illustrates the concept *IsRoot*.

9.8.3. Concept *PredicateOperator*

The concept *PredicateOperator* represents all Edge Algebra expressions both operating on at least one node predicate and evaluating to a node predicate. The node predicate operators will be categorised into two classes: those which operate on exactly one operand (see

Section 9.8.4, *Concept* SinglePredicateOperator, p. 252), those which operate on at least two operands (see Section 9.8.5, *Concept* MultiPredicateOperator, p. 253). Further, there are two special cases *ForAllConcepts* and *ForAllProperties*, as they comprise a value in addition to the predicate operand and will thus be described directly in this section.

Abstract Syntax. The formal abstract syntax for *PredicateOperator* is defined in Listing 9.151.

Listing 9.151: Abstract Syntax for *PredicateOperator*

```
[PredicateOperator!!] refines Predicate ::>
  predicateOperand[1..*](List) : C:Predicate ;

[QuantifiedEdging!!] refines PredicateOperator ::>
  boundedVariable[1..1] : C:Identifier ,
  predicateOperand[1..1] ;
QuantifiedConceptEdging!! refines QuantifiedEdging ::=
  excludedConcept[0..*](Set) : C:Concept ;
QuantifiedTypeEdging!! refines QuantifiedEdging ::=
  excludedConcept[0..*](Set) : C:Concept ;
QuantifiedPropertyEdging!! refines QuantifiedEdging ::=
  excludedProperty[0..*](Set) : C:Property ;
```

The properties of the concept *PredicateOperator* are:

- **predicateOperand:** list of node predicate operands for a node predicate operator. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands.

The properties of the concept *QuantifiedEdging* are:

- **predicateOperand:** bounded edge variable for the quantification.
- **predicateOperand (refined):** node predicate which should be universally quantified over concepts.

The additional properties of the concepts *QuantifiedConceptEdging* and *QuantifiedTypeEdging* are:

- **excludedConcept:** set of concept which should not be included within the universal quantification.

The properties of the concept *QuantifiedPropertyEdging* are:

- **excludedProperty:** set of properties which should not be included within the universal quantification.

Concrete Syntax. The formal textual concrete syntax for *PredicateOperator* is defined in Listing 9.152.

Listing 9.152: Textual Concrete Syntax for *PredicateOperator*

```
QuantifiedConceptEdging: ("∀C:" OR "forallC:")
  (P:boundedVariable)
```

```
579   ("\\" "{" || &P:excludedConcept / "," _ || "}")
580   ":" _ (P:predicateOperand[Bracketed]) ;
581 QuantifiedTypeEdging: ("∀T:" OR "forallT:")
582   (P:boundedVariable)
583   ("\\" "{" || &P:excludedConcept / "," _ || "}")
584   ":" _ (P:predicateOperand[Bracketed]) ;
585 QuantifiedPropertyEdging: ("∀P:" OR "forallP:")
586   (P:boundedVariable)
587   ("\\" "{" || &P:excludedProperty / "," _ || "}")
588   ":" _ (P:predicateOperand[Bracketed]) ;
```

No concrete syntax has to be defined for the abstract concept *PredicateOperator*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will remain unchanged and thus the parentheses will be optional.

The three universal quantifiers will be denoted by the common symbol ∀ or the keyword **forall**, followed by the name of the bounded variable. If some of the concepts and properties respectively shall be excluded from universal quantification, they will be written within curly brackets with a preceding backslash. Note that the doubling of the backslash will result from the encoding of strings in M2L.

Example. An example for the concept *PredicateOperator* will be provided in the following Listing 9.153:

Listing 9.153: Example for *PredicateOperator*

```
∀p\{composite} |p| = 0
```

The given example shows the universal quantifier over properties. It states that each property, except for *composite*, must be empty for every node.

9.8.4. Concept *SinglePredicateOperator*

The concept *SinglePredicateOperator* represents all node predicate operators both operating on and evaluating to a single node predicate. Only one node predicate operator operating on one node predicate will be defined:

- **Not** returns the negation of the boolean values for each node.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition.

Abstract Syntax. The formal abstract syntax for *SinglePredicateOperator* is defined in Listing 9.154.

Listing 9.154: Abstract Syntax for *SinglePredicateOperator*

```
322 [SinglePredicateOperator!!]
323   refines PredicateOperator ::=
324   predicateOperand[1..1] ;
325 Not!! refines SinglePredicateOperator ;
```

The properties of the concept *SinglePredicateOperator* are:

- **predicateOperand (refined):** node predicate operand for a single-valued node predicate operator.

The refining concept *Not* does not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *SinglePredicateOperator* is defined in Listing 9.155.

Listing 9.155: Textual Concrete Syntax for *SinglePredicateOperator*

```
590 Not: ("¬" OR "!") (P: predicateOperand [Bracketed]) ;
```

The single-valued node predicate operator *Not* will be denoted in a prefix form. The prefix symbol may be ¬ or an exclamation mark (!).

Example. An example for the concept *SinglePredicateOperator* will be provided in the following Listing 9.156:

Listing 9.156: Example for *SinglePredicateOperator*

```
¬root?
```

The given example evaluates to *true* if the node is not a root node.

9.8.5. Concept *MultiPredicateOperator*

The concept *MultiPredicateOperator* represents all node predicate operators operating on a list of node predicates and evaluating to a single node predicate. Five node predicate operators operating on multiple node predicates will be defined in total:

- **And** returns a node predicate conjuncting the boolean values.
- **Or** returns a node predicate disjuncting the boolean values.
- **Xor** returns a node predicate exclusively disjuncting the boolean values.
- **Iff** returns a node predicate that holds if all boolean values are *true* or all boolean values are *false*. This if-and-only-if operator can be seen as a boolean equation operator.
- **Implies** returns a node predicate that holds if the boolean values have the form that once a *true* occurs, a *false* will never occur afterwards.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition. In contrast to the definitions in the sections herein, the operators will be defined on any number of operands greater than or equal to two instead of on exactly two operands. The present extension can be reduced to the basic definition by the following equations in 9.5, wherein $\mathrm{op}^1 \in \{\wedge, \vee\}$ and $\mathrm{op}^2 \in \{\oplus, \Leftrightarrow, \Rightarrow\}$:

$$\begin{aligned} & x_1 \ \mathrm{op}^1 \ x_2 \ \mathrm{op}^1 \ x_3 \ \mathrm{op}^1 \ \ldots \ \mathrm{op}^1 \ x_n \\ & = (\ldots((x_1 \ \mathrm{op}^1 \ x_2) \ \mathrm{op}^1 \ x_3) \ \mathrm{op}^1 \ \ldots) \ \mathrm{op}^1 \ x_n \\ & \\ & x_1 \ \mathrm{op}^2 \ x_2 \ \mathrm{op}^2 \ x_3 \ \mathrm{op}^2 \ \ldots \ \mathrm{op}^2 \ x_n \\ & = (x_1 \ \mathrm{op}^2 \ x_2) \wedge \ldots \wedge (x_1 \ \mathrm{op}^2 \ x_n) \wedge (x_2 \ \mathrm{op}^2 \ x_3) \wedge (x_2 \ \mathrm{op}^2 \ x_n) \wedge \ldots \end{aligned} \tag{9.5}$$

For *And* and *Or*, multiple operands will be construed by an implicit bracketing from the left side, e. g. $a \vee b \vee c = (a \vee b) \vee c$. For *Xor*, *Iff* and *Implies*, multiple operands will be construed by conjuncting the pairwise application of the operator while preserving the order of the operands, e. g. $a \Rightarrow b \Rightarrow c = (a \Rightarrow b) \wedge (a \Rightarrow c) \wedge (b \Rightarrow c)$. Note that according to this definition $a \Rightarrow b \Rightarrow c \neq (a \Rightarrow b) \Rightarrow c$.

Abstract Syntax. The formal abstract syntax for *MultiPredicateOperator* is defined in Listing 9.157.

Listing 9.157: Abstract Syntax for *MultiPredicateOperator*

```
[MultiPredicateOperator!!]
  refines PredicateOperator ::=
  predicateOperand[2..*] ;
And!! refines MultiPredicateOperator ;
Or!! refines MultiPredicateOperator ;
Xor!! refines MultiPredicateOperator ;
Iff!! refines MultiPredicateOperator ;
Implies!! refines MultiPredicateOperator ;
```

The properties of the concept *MultiPredicateOperator* are:

- **predicateOperand (refined):** node predicate operands for a multi-valued node predicate operator which must be at least two.

All refinements of the concept *MultiPredicateOperator* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *MultiPredicateOperator* is defined in Listing 9.158.

Listing 9.158: Textual Concrete Syntax for *MultiPredicateOperator*

```
And: (P:predicateOperand[Bracketed]
  / _ ("∧" OR "&") _) ;
Or:  (P:predicateOperand[Bracketed]
  / _ ("∨" OR "v") _) ;
Xor: (P:predicateOperand[Bracketed]
  / _ ("⊕" OR "xor") _) ;
Iff: (P:predicateOperand[Bracketed]
  / _ ("⇔" OR "<=>") _) ;
Implies: (P:predicateOperand[Bracketed]
  / _ ("⇒" OR "=>") _) ;
```

All multi-valued node predicate operators will be denoted in an infix form. There are two ways of encoding all operators. One is a short symbolic way, the other one uses a keyword or an encoding by ascii characters.

- For the **and** operator, the symbol will be $\wedge$ and the ascii encoding will be &.
- For the **or** operator, the symbol will be $\vee$ and the keyword will be **v**.
- For the **xor** operator, the symbol will be $\oplus$ and the keyword will be **xor**.
- For the **iff** operator, the symbol will be $\Leftrightarrow$ and the ascii encoding will be <=>.

- For the **implies** operator, the symbol will be ⇒ and the ascii encoding will be =>.

Example. An example for the concept *MultiPredicateOperator* will be provided in the following Listing 9.159:

Listing 9.159: Example for *MultiPredicateOperator*

```
root? ⇒ |⇆P:composite|=0 ⇒ root?
```

The given example illustrates the *implies* operator having three operands: The first and the third one are the *is-root* operators. The operator in the middle is an equation. All in all, the expression provides the definition of the *is-root* operator without using the *if-and-only-if* operator.

9.8.6. Concept *EdgeToPredicate*

The concept *EdgeToPredicate* represents all Edge Algebra expressions operating on at least one edge and evaluating to a node predicate. The predicate expressions operating on edges will be categorised into two classes: those which operate on exactly one operand (see Section 9.8.7, *Concept* SingleEdgeToPredicate, p. 255), and those which operate on at least two operands (see Section 9.8.8, *Concept* MultiEdgeToPredicate, p. 257).

Abstract Syntax. The formal abstract syntax for *EdgeToPredicate* is defined in Listing 9.160.

Listing 9.160: Abstract Syntax for *EdgeToPredicate*

```
336 [EdgeToPredicate!!] refines Predicate ::>
337   edgeOperand[1..*](List) : C:Edge ;
```

The properties of the concept *EdgeToPredicate* are:

- **edgeOperand:** list of edge operands for a predicate expression operating on edges. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *EdgeToPredicate*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will remain unchanged and thus the parentheses will be optional.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts.

9.8.7. Concept *SingleEdgeToPredicate*

The concept *SingleEdgeToPredicate* represents all Edge Algebra expressions evaluating to a node predicate and operating on exactly one edge. Eight predicate expressions operating on one edge will be defined in total:

- **CastEdgeToPredicate** returns a node predicate that holds for a node if the given edge evaluates to a single-valued pomset directing towards *true*.
- **IsEmpty** returns a node predicate that holds for a node if the given edge evaluates to an empty pomset.
- **IsSingleton** returns a node predicate that holds for a node if the given edge evaluates to a pomset having one element at most.
- **IsSet** returns a node predicate that holds for a node if the given edge evaluates to a set, i. e. the pomset will be unordered and will not contain duplicates.
- **IsBag** returns a node predicate that holds for a node if the given edge evaluates to a bag, i. e. the pomset will be unordered.
- **IsList** returns a node predicate that holds for a node if the given edge evaluates to a list, i. e. the pomset will be totally ordered.
- **IsToset** returns a node predicate that holds for a node if the given edge evaluates to a totally ordered set, i. e. the pomset will be totally ordered and will not contain duplicates.
- **IsPoset** returns a node predicate that holds for a node if the given edge evaluates to a partially ordered set, i. e. the pomset will not contain duplicates.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition.

Abstract Syntax. The formal abstract syntax for *SingleEdgeToPredicate* is defined in Listing 9.161.

Listing 9.161: Abstract Syntax for *SingleEdgeToPredicate*

```
331 [SingleEdgeToPredicate!!] refines EdgeToPredicate ::=
332   edgeOperand[1..1] ;
333 CastEdgeToPredicate!! refines SingleEdgeToPredicate ;
334 IsEmpty!! refines SingleEdgeToPredicate ;
335 IsSingleton!! refines SingleEdgeToPredicate ;
336 IsSet!! refines SingleEdgeToPredicate ;
337 IsBag!! refines SingleEdgeToPredicate ;
338 IsList!! refines SingleEdgeToPredicate ;
339 IsToset!! refines SingleEdgeToPredicate ;
340 IsPoset!! refines SingleEdgeToPredicate ;
```

The properties of the concept *SingleEdgeToPredicate* are:

- **edgeOperand (refined):** edge operand for a single-valued predicate expression operating on edges.

All refinements of the concept *SingleEdgeToPredicate* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *SingleEdgeToPredicate* is defined in Listing 9.162.

Listing 9.162: Textual Concrete Syntax for *SingleEdgeToPredicate*

```
603 CastEdgeToPredicate:
604   "(bool)" (P:edgeOperand[Bracketed]) ;
605 IsEmpty: "empty?" _ (P:edgeOperand[Bracketed]) ;
606 IsSingleton:
607   "singleton?" _ (P:edgeOperand[Bracketed]) ;
608 IsSet: "set?" _ (P:edgeOperand[Bracketed]) ;
609 IsBag: "bag?" _ (P:edgeOperand[Bracketed]) ;
610 IsList: "list?" _ (P:edgeOperand[Bracketed]) ;
611 IsToset: "toset?" _ (P:edgeOperand[Bracketed]) ;
612 IsPoset: "poset?" _ (P:edgeOperand[Bracketed]) ;
```

All single-valued edge expressions operating on node predicates will be denoted in a prefix form.

- For the **is-empty** operator, the keyword will be **empty**?.
- For the **is-singleton** operator, the keyword will be **singleton**?.
- For the **is-set** operator, the keyword will be **set**?.
- For the **is-bag** operator, the keyword will be **bag**?.
- For the **is-list** operator, the keyword will be **list**?.
- For the **is-toset** operator, the keyword will be **toset**?.
- For the **is-poset** operator, the keyword will be **poset**?.

Example. An example for the concept *SingleEdgeToPredicate* will be provided in the following Listing 9.163:

Listing 9.163: Example for *SingleEdgeToPredicate*

```
list? P:parameter
```

The given example checks whether the property *parameter* will be totally ordered.

9.8.8. Concept *MultiEdgeToPredicate*

The concept *MultiEdgeToPredicate* represents all Edge Algebra expressions evaluating to a node predicate and operating on at least two edges. Eight predicate expressions operating on multiple edges will be defined in total:

- **EdgeEqual** returns a node predicate that holds for each node the given edges thereof evaluating to equal pomsets.
- **EdgeNotEqual** returns a node predicate that holds for each node the given edges thereof evaluating to pairwise unequal pomsets.
- **ConsistsOf** returns a node predicate that holds for each node the given edges thereof evaluating to pomsets such that the more left ones will pairwise consist of the more right ones.
- **NotConsistsOf** returns a node predicate that holds for each node the given edges thereof evaluating to pomsets such that the more left ones will pairwise not consist of the more right ones.

- **Subset** returns a node predicate that holds for each node the given edges thereof evaluating to pomsets such that the more left ones will pairwise be real subsets of the more right ones. Note that this subset operator will be based on connected components.
- **NotSubset** returns a node predicate that holds for each node the given edges thereof evaluating to pomsets such that the more left ones will pairwise be no real subsets of the more right ones. Note that this subset operator will be based on connected components.
- **SubsetOrEqual** returns a node predicate that holds for each node the given edges thereof evaluating to pomsets such that the more left ones will pairwise be subsets of or equal to the more right ones. Note that this subset operator will be based on connected components.
- **NotSubsetOrEqual** returns a node predicate that holds for each node the given edges thereof evaluating to pomsets such that the more left ones will pairwise be no subsets of and not be equal to the more right ones. Note that this subset operator will be based on connected components.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition. In contrast to the definitions in the sections herein, the operators will be defined on any number of operands greater than or equal to two instead of on exactly two operands. The present extension can be reduced to the basic definition by the following equations in 9.6, wherein op will be one of the operators mentioned above:

$$\begin{aligned} &x_1 \text{ op } x_2 \text{ op } x_3 \text{ op } \ldots \text{ op } x_n \\ &= (x_1 \text{ op } x_2) \wedge \ldots \wedge (x_1 \text{ op } x_n) \wedge (x_2 \text{ op } x_3) \wedge (x_2 \text{ op } x_n) \wedge \ldots \end{aligned} \tag{9.6}$$

Multiple operands will be construed by conjuncting the pairwise application of the operator while preserving the order of the operands, e. g. $a \notin b \notin c = (a \notin b) \wedge (a \notin c) \wedge (b \notin c)$. Note that according to this definition $a \notin b \notin c \neq \neg(a \in b \in c)$.

Abstract Syntax. The formal abstract syntax for *MultiEdgeToPredicate* is defined in Listing 9.164.

Listing 9.164: Abstract Syntax for *MultiEdgeToPredicate*

```
350 [MultiEdgeToPredicate!!] refines EdgeToPredicate ::=
351   edgeOperand[2..*] ;
352 EdgeEqual!! refines MultiEdgeToPredicate ;
353 EdgeNotEqual!! refines MultiEdgeToPredicate ;
354 ConsistsOf!! refines MultiEdgeToPredicate ;
355 NotConsistsOf!! refines MultiEdgeToPredicate ;
356 Subset!! refines MultiEdgeToPredicate ;
357 NotSubset!! refines MultiEdgeToPredicate ;
358 SubsetOrEqual!! refines MultiEdgeToPredicate ;
359 NotSubsetOrEqual!! refines MultiEdgeToPredicate ;
```

The properties of the concept *MultiEdgeToPredicate* are:

- **edgeOperand (refined):** edge operand for a multi-valued predicate expression operating on edges.

All refinements of the concept *MultiEdgeToPredicate* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *MultiEdgeToPredicate* is defined in Listing 9.165.

Listing 9.165: Textual Concrete Syntax for *MultiEdgeToPredicate*

```
614 EdgeEqual: (P:edgeOperand[Bracketed]
615   / _ "=" _) ;
616 EdgeNotEqual: (P:edgeOperand[Bracketed]
617   / _ ("≠" OR "!=") _) ;
618 ConsistsOf: (P:edgeOperand[Bracketed]
619   / _ ("∈" OR "consistsOf") _) ;
620 NotConsistsOf: (P:edgeOperand[Bracketed]
621   / _ ("∉" OR "!consistsOf") _) ;
622 Subset: (P:edgeOperand[Bracketed]
623   / _ ("⊂" OR "subset") _) ;
624 NotSubset: (P:edgeOperand[Bracketed]
625   / _ ("⊄" OR "!subset") _) ;
626 SubsetOrEqual: (P:edgeOperand[Bracketed]
627   / _ ("⊆" OR "subsetEq") _) ;
628 NotSubsetOrEqual: (P:edgeOperand[Bracketed]
629   / _ ("⊈" OR "!subsetEq") _) ;
```

All multi-valued predicate expressions operating on edges will be denoted in an infix form. There are two ways of encoding for most of the operators. One is a short symbolic way, the other one uses a keyword or an encoding by ascii characters.

- For the **equality** operator, the symbol will be =. An alternative keyword will not be defined.
- For the **inequality** operator, the symbol will be ≠ and the ascii encoding will be !=.
- For the **consists-of** operator, the symbol will be ∈ and the keyword will be consistsOf.
- For the **not-consists-of** operator, the symbol will be ∉ and the keyword will be !consistsOf.
- For the **subset-of** operator, the symbol will be ⊂ and the keyword will be subset.
- For the **not-subset-of** operator, the symbol will be ⊄ and the keyword will be !subset.
- For the **subset-or-equal** operator, the symbol will be ⊆ and the keyword will be subsetEq.
- For the **not-subset-or-equal** operator, the symbol will be ⊈ and the keyword will be !subsetEq.

Example. An example for the concept *MultiEdgeToPredicate* will be provided in the following Listing 9.166:

Listing 9.166: Example for *MultiEdgeToPredicate*

```
P: child ∈ C: Person
```

The given example checks whether the property *child* will only consist of nodes of the concept *Person*.

9.8.9. Concept *NumericalToPredicate*

The concept *NumericalToPredicate* represents all Edge Algebra expressions operating on at least one node valuation and evaluating to a node predicate. The available predicate expressions operating on node valuations are all of the same kind: They all operate on multiple node valuations (see Section 9.8.10, *Concept* MultiNumericalToPredicate, p. 260).

Abstract Syntax. The formal abstract syntax for *NumericalToPredicate* is defined in Listing 9.167.

Listing 9.167: Abstract Syntax for *NumericalToPredicate*

```
361 [NumericalToPredicate!!] refines Predicate ::>
362   numericalOperand[1..*](List) : C:Numerical ;
```

The properties of the concept *NumericalToPredicate* are:

- **numericalOperand:** list of node valuation operands for a predicate expression operating on node valuations. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands. Note that the first variant has never been used yet.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *NumericalToPredicate*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will remain unchanged and thus the parentheses will be optional.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts.

9.8.10. Concept *MultiNumericalToPredicate*

The concept *MultiNumericalToPredicate* represents all Edge Algebra expressions evaluating to a node predicate and operating on at least two node valuations. Six predicate expressions operating on multiple node valuations will be defined in total:

- **NumericalEqual** returns a node predicate that holds for each node the given node valuations thereof evaluating to equal natural numbers.
- **NumericalNotEqual** returns a node predicate that holds for each node the given node valuations thereof evaluating to pairwise unequal natural numbers.
- **LessOrEqual** returns a node predicate that holds for each node the given node valuations thereof evaluating to natural numbers such that the more left ones will pairwise be less than or equal to the more right ones.
- **LessThan** returns a node predicate that holds for each node the given node valuations thereof evaluating to natural numbers such that the more left ones will pairwise be less than the more right ones.
- **GreaterOrEqual** returns a node predicate that holds for each node the given node valuations thereof evaluating to natural numbers such that the more left ones will pairwise be greater than or equal to the more right ones.

- **GreaterThan** returns a node predicate that holds for each node the given node valuations thereof evaluating to natural numbers such that the more left ones will pairwise be greater than the more right ones.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition. In contrast to the definitions in the sections herein, the operators will be defined on any number of operands greater than or equal to two instead of on exactly two operands. The present extension can be reduced to the basic definition by the following equations in 9.7, wherein op will be one of the operators mentioned above:

$$\begin{aligned} & x_1 \text{ op } x_2 \text{ op } x_3 \text{ op } \ldots \text{ op } x_n \\ & = (x_1 \text{ op } x_2) \wedge \ldots \wedge (x_1 \text{ op } x_n) \wedge (x_2 \text{ op } x_3) \wedge (x_2 \text{ op } x_n) \wedge \ldots \end{aligned} \tag{9.7}$$

Multiple operands will be construed by conjuncting the pairwise application of the operator while preserving the order of the operands, e. g. $a \leq b \leq c = (a \leq b) \wedge (a \leq c) \wedge (b \leq c)$.

Abstract Syntax. The formal abstract syntax for *MultiNumericalToPredicate* is defined in Listing 9.168.

Listing 9.168: Abstract Syntax for *MultiNumericalToPredicate*

```
[MultiNumericalToPredicate!!]
  refines NumericalToPredicate ::=
  numericalOperand[2..*] ;
NumericalEqual!! refines MultiNumericalToPredicate ;
NumericalNotEqual!! refines MultiNumericalToPredicate ;
LessOrEqual!! refines MultiNumericalToPredicate ;
LessThan!! refines MultiNumericalToPredicate ;
GreaterOrEqual!! refines MultiNumericalToPredicate ;
GreaterThan!! refines MultiNumericalToPredicate ;
```

The properties of the concept *MultiNumericalToPredicate* are:

- **numericalOperand (refined):** node valuation operands for a multi-valued predicate expression operating on node valuations.

All refinements of the concept *MultiEdgeToPredicate* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *MultiNumericalToPredicate* is defined in Listing 9.169.

Listing 9.169: Textual Concrete Syntax for *MultiNumericalToPredicate*

```
NumericalEqual: (P:numericalOperand[Bracketed]
  / _ "=" _) ;
NumericalNotEqual: (P:numericalOperand[Bracketed]
  / _ ("≠" OR "!=") _) ;
LessOrEqual: (P:numericalOperand[Bracketed]
  / _ ("≤" OR "<=") _) ;
LessThan: (P:numericalOperand[Bracketed]
  / _ "<" _) ;
GreaterOrEqual: (P:numericalOperand[Bracketed]
```

```
640   / _ ("≥" OR ">=") _) ;
641 GreaterThan: (P:numericalOperand[Bracketed]
642   / _ ">" _) ;
```

All multi-valued predicate expressions operating on node valuations will be denoted in an infix form. There are two ways of encoding for most of the operators. One is a short symbolic way, the other one uses an encoding by ascii characters.

- For the **equality** operator, the symbol will be =. An alternative keyword will not be defined.
- For the **inequality** operator, the symbol will be ≠ and the ascii encoding will be !=.
- For the **less-or-equal** operator, the symbol will be ≤ and the ascii encoding will be <=.
- For the **less-than** operator, the symbol will be <. An additional ascii encoding will not be necessary.
- For the **greater-or-equal** operator, the symbol will be ≥ and the ascii encoding will be >=.
- For the **greater-than** operator, the symbol will be >. An additional ascii encoding will not be necessary.

Example. An example for the concept *MultiNumericalToPredicate* will be provided in the following Listing 9.170:

Listing 9.170: Example for *MultiNumericalToPredicate*

```
(number)P:age ≥ 18
```

The given example checks whether a property *age* will be greater than or equal to 18.

9.9. Package *ORG.Metamodels.EdgeAlgebra.NumericalExpressions*

This package includes all functions of Edge Algebra the co-domain thereof being a node valuation $\mathcal{N}_V = V \to \mathbb{N}$. All functions defined herein are part of the *Propositional Edge Algebra*. Table 9.11 shows the list of all concepts defined:

Concept	Description
Numerical	expression evaluating to a node valuation
ConstantNumerical	expression without any parameters
NumericalValue	concrete value of a natural number
NumericalOperator	operator having at least one node valuation operand
MultiNumericalOperator	operator having at least two node valuation operands
Addition	node valuation operator according to the $+$ operator
Subtraction	node valuation operator according to the $-$ operator
Multiplication	node valuation operator according to the $\cdot$ operator
IntegerDivision	node valuation operator according to the $\div$ operator
Modulo	node valuation operator according to the $\%$ operator
Minimum	returns the minimum for each node
Maximum	returns the maximum for each node
EdgeToNumerical	function having at least one edge operand
SingleEdgeToNumerical	function having exactly one edge operand
CastEdgeToNumerical	cast a natural number valued edge to a node valuation
Cardinality	returns the cardinality of a pomset for each node
Depth	returns the depth of a pomset for each node

Table 9.11.: List of concepts defined in *ORG.Metamodels.EdgeAlgebra.NumericalExpressions*

9.9.1. Concept *Numerical*

The abstract concept *Numerical* represents all Edge Algebra expressions evaluating to a node valuation and will thus be refined by each of the dedicated concepts. The concept *Numerical* thus represents all functions and operators the co-domain thereof being $\mathcal{N}_V = V \to \mathbb{N}$. Note that according to this, the co-domain of such an Edge Algebra expression will formally again be a function mapping each node of an M-graph to a natural number.

Abstract Syntax. The formal abstract syntax for *Numerical* is defined in Listing 9.171.

Listing 9.171: Abstract Syntax for *Numerical*

```
376 [Numerical!!] ;
```

Numerical is an abstract concept without any properties that have been additionally defined. It is prepared to be refined by concepts representing dedicated Edge Algebra statements.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *Numerical*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will nonetheless be defined as a default in which the parentheses will be optional. Please refer to Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291 for a detailed definition of the helping syntax *Bracketed*.

Example. An example for the concept *Numerical* will be provided in the following Listing 9.172:

Listing 9.172: Example for *Numerical*

```
(|P:propertyA| + |P:propertyB| + |P:propertyC|) ÷ 3
```

The given example illustrates a node valuation calculating an average size of the three properties, namely *propertyA*, *propertyB*, and *propertyC*. Please refer to the dedicated sections for a detailed definition of the several operators.

9.9.2. Concept *ConstantNumerical*

The concept *ConstantNumerical* represents all Edge Algebra expressions evaluating to a node valuation but having no operands as operands are elements from the three Edge Algebra carrier sets. Constant edges may, however, comprise a parameter. A single constant node valuation will be defined:

- **NumericalValue** allows for the definition of a constant node valuation by a natural number.

Abstract Syntax. The formal abstract syntax for *ConstantNumerical* is defined in Listing 9.173.

Listing 9.173: Abstract Syntax for *ConstantNumerical*

```
378 [ConstantNumerical!!] refines Numerical ;
379 NumericalValue!! refines ConstantNumerical ::=
380   value[1..1] : C:Natural ;
```

The properties of the concept *NumericalValue* are:

- **value:** constant natural number assigned to each node.

Concrete Syntax. The formal textual concrete syntax for *ConstantNumerical* is defined in Listing 9.174.

Listing 9.174: Textual Concrete Syntax for *ConstantNumerical*

```
646 NumericalValue: (P:value) ;
```

A numerical value will be encoded by simply encoding a natural number.

Example. An example for the concept *ConstantNumerical* will be provided in the following Listing 9.175:

Listing 9.175: Example for *ConstantNumerical*

```
123
```

The example illustrates the concept *NumercialValue* by encoding the number 123.

9.9.3. Concept *NumericalOperator*

The concept *NumericalOperator* represents all Edge Algebra expressions both operating on at least one node valuation and evaluating to a node valuation. The available node valuation operators are all of the same kind: They all operate on multiple node valuations (see Section 9.9.4, *Concept* MultiNumericalOperator, p. 265).

Abstract Syntax. The formal abstract syntax for *NumericalOperator* is defined in Listing 9.176.

Listing 9.176: Abstract Syntax for *NumericalOperator*

```
382 [NumericalOperator!!] refines Numerical ::>
383   numericalOperand[1..*](List) : C:Numerical ;
```

The properties of the concept *NumericalOperator* are:

- **numericalOperand:** list of node valuation operands for a node valuation operator. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands. Note that the first variant has never been used yet.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *NumericalOperator*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will remain unchanged and thus the parentheses will be optional.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts.

9.9.4. Concept *MultiNumericalOperator*

The concept *MultiNumericalOperator* represents all node valuation operators operating on a list of node valuations and evaluating to a single node valuation. Seven node valuation operators operating on multiple node valuations will be defined in total:

- **Addition** returns a node predicate adding the natural numbers for each node.
- **Subtraction** returns a node predicate subtracting the natural numbers for each node. As this operator operates on natural numbers, a negative result will be substituted by zero.
- **Multiplication** returns a node predicate multiplying the natural numbers for each node.
- **IntegerDivision** returns a node predicate dividing the natural numbers for each node by integers.
- **Modulo** returns a node predicate returning the rest of an integer division for each node.
- **Minimum** returns a node predicate resulting in the minimum of natural numbers for each node.

- **Maximum** returns a node predicate resulting in the maximum of natural numbers for each node.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 for a detailed definition. In contrast to the definitions in the sections herein, the operators will be defined on any number of operands greater than or equal to two instead of on exactly two operands. The present extension can be reduced to the basic definition by the following equations in 9.8, wherein op will be one of the operators mentioned above:

$$\begin{aligned} & x_1 \text{ op } x_2 \text{ op } x_3 \text{ op } \ldots \text{ op } x_n \\ & = (\ldots((x_1 \text{ op } x_2) \text{ op } x_3) \text{ op } \ldots) \text{ op } x_n \end{aligned} \tag{9.8}$$

Multiple operands will be construed by an implicit bracketing from the left, e. g. $a - b - c = (a - b) - c$.

Abstract Syntax. The formal abstract syntax for *MultiNumericalOperator* is defined in Listing 9.177.

Listing 9.177: Abstract Syntax for *MultiNumericalOperator*

```
385 [MultiNumericalOperator!!]
386   refines NumericalOperator ::=
387   numericalOperand[2..*] ;
388 Addition!! refines MultiNumericalOperator ;
389 Subtraction!! refines MultiNumericalOperator ;
390 Multiplication!! refines MultiNumericalOperator ;
391 IntegerDivision!! refines MultiNumericalOperator ;
392 Modulo!! refines MultiNumericalOperator ;
393 Minimum!! refines MultiNumericalOperator ;
394 Maximum!! refines MultiNumericalOperator ;
```

The properties of the concept *MultiNumericalOperator* are:

- **numericalOperand (refined):** node valuation operands for a multi-valued node valuation operator which must be at least two.

All refinements of the concept *MultiNumericalOperator* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *MultiNumericalOperator* is defined in Listing 9.178.

Listing 9.178: Textual Concrete Syntax for *MultiNumericalOperator*

```
648 Addition: (P:numericalOperand[Bracketed]
649   / _ "+" _) ;
650 Subtraction: (P:numericalOperand[Bracketed]
651   / _ "-" _) ;
652 Multiplication: (P:numericalOperand[Bracketed]
653   / _ ("·" OR "*") _) ;
654 IntegerDivison: (P:numericalOperand[Bracketed]
655   / _ ("÷" OR "div") _) ;
656 Modulo: (P:numericalOperand[Bracketed]
```

```
657   / _ ("%" OR "mod") _) ;
658 Minimum: "min" "(" (P:numericalOperand / "," _) ")" ;
659 Maximum: "max" "(" (P:numericalOperand / "," _) ")" ;
```

All multi-valued node valuation operators, except for *Minimum* and *Maximum*, will be denoted in an infix form. There are two ways of encoding some of the operators. One is a short symbolic way, the other one uses a keyword or an encoding by ASCII characters.

- For the **addition** operator, the symbol will be +. An alternative encoding will not be necessary.
- For the **subtraction** operator, the symbol will be −. An alternative encoding will not be necessary.
- For the **multiplication** operator, the symbol will be · and the alternative ascii encoding will be ∗.
- For the **integer-division** operator, the symbol will be ÷ and the alternative keyword will be **div**.
- For the **modulo** operator, the symbol will be % and the alternative keyword will be **mod**.

Minimum and *Maximum* will be encoded in functional prefix notation. The corresponding keywords will be **min** and **max**.

Example. An example for the concept *MultiNumericalOperator* will be provided in the following Listing 9.179:

Listing 9.179: Example for *MultiNumericalOperator*

```
min(2, 2 · |P:propertyA|, 1) + (2 · 3) + (|P:propertyB| % 2)
```

The given example illustrates an arbitrary expression of node valuation operators. Note that the second addend – i. e. the multiplication – will have to be written within parentheses, as a prioritization of operators will consciously and consequently be avoided – even in this particular case.

9.9.5. Concept *EdgeToNumerical*

The concept *EdgeToNumerical* represents all Edge Algebra expressions operating on at least one edge and evaluating to a node valuation. The available node valuation expressions operating on edges are all of the same kind: They all operate on a single edge (see Section 9.9.6, *Concept* SingleEdgeToNumerical, p. 268).

Abstract Syntax. The formal abstract syntax for *EdgeToNumerical* is defined in Listing 9.180.

Listing 9.180: Abstract Syntax for *EdgeToNumerical*

```
396 [EdgeToNumerical!!] refines Numerical ::=
397   edgeOperand[1..*](List) : Edge ;
```

The properties of the concept *EdgeToNumerical* are:

- **edgeOperand:** list of edge operands for a predicate expression operating on edges. At least one operand will be required. This property will normally be refined to exactly one operand or at least two operands. Note that the first variant has never been used yet.

Concrete Syntax. No concrete syntax has to be defined for the abstract concept *EdgeToNumerical*. Please refer to the (non-abstract) refining concepts. The helping syntax *Bracketed* will remain unchanged and thus the parentheses will be optional.

Example. As no concrete syntax is defined for this abstract concept, please refer to the (non-abstract) refining concepts.

9.9.6. Concept *SingleEdgeToNumerical*

The concept *SingleEdgeToNumerical* represents all Edge Algebra expressions evaluating to a node valuation and operating on exactly one edge. Three numerical expressions operating on one edge will be defined in total:

- **CastEdgeToNumerical** returns a node valuation that will return a natural number for each node according to the evaluation of the given edge: If this edge evaluates to a single-valued pomset directing towards a node having the concept *Natural*, this value will be returned. In all other cases, zero will be returned.
- **Cardinality** returns a node valuation that will return the cardinality (i. e. the size of the pomset) for each node according to the evaluation of the given edge.
- **Depth** returns a node valuation that will return the depth for each node according to the evaluation of the given edge.

Please refer to Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 and Chapter 4, *Pomsets in the context of metamodelling*, p. 61 for a detailed definition.

Abstract Syntax. The formal abstract syntax for *SingleEdgeToNumerical* is defined in Listing 9.181.

Listing 9.181: Abstract Syntax for *SingleEdgeToNumerical*

```
399 [SingleEdgeToNumerical!!] refines EdgeToNumerical ::=
400   edgeOperand[1..1] ;
401 CastEdgeToNumerical!! refines SingleEdgeToNumerical ;
402 Cardinality!! refines SingleEdgeToNumerical ;
403 Depth!! refines SingleEdgeToNumerical ;
```

The properties of the concept *SingleEdgeToNumerical* are:

- **edgeOperand (refined):** edge operand for a single-valued node valuation expression operating on edges.

All refinements of the concept *SingleEdgeToNumerical* do not have any additional property restrictions.

Concrete Syntax. The formal textual concrete syntax for *SingleEdgeToNumerical* is defined in Listing 9.182.

Listing 9.182: Textual Concrete Syntax for *SingleEdgeToNumerical*

```
CastEdgeToNumerical:
  "(number)" (P:edgeOperand[Bracketed]) ;
Cardinality: ("size" _ (P:edgeOperand[Bracketed])
  OR "|" (P:edgeOperand) "|") ;
Depth: ("depth" _ (P:edgeOperand[Bracketed]) ;
  OR "||" (P:edgeOperand) "||") ;
```

The **cast** operator will be denoted by a preceding (number). The **cardinality** operator will be denoted by a single pipe symbol (|) on each side. The **depth** operator will be denoted by a doubled pipe symbol (||) on each side.

Example. An example for the concept *SingleEdgeToNumerical* will be provided in the following Listing 9.183:

Listing 9.183: Example for *SingleEdgeToNumerical*

```
(number)P:age
```

The given example returns a node valuation according to the value of the property *age*. If this property does not evaluate to a single-valued pomset containing a natural number, zero will be returned.

Chapter 10

Summary, evaluation, and outlook

With Chapter 9, *The overall specification of M2L*, p. 171, the main goal of the work has been achieved: A formal, but also appropriate metamodelling language allowing for fully specifying all syntactical issues of modelling languages has been introduced. The present chapter shall now give a short summary of the present work by reflecting all steps that have been passed. After a discussion of the advantages and disadvantages, an outlook on the future work will be provided.

Contents

10.1. Summary

In order to be able to define new modelling languages in an adequate way, a formal metamodelling approach leading to the metamodelling language *M2L* has been established in the present work.

First of all, a set of crucial requirements relevant for such a metamodelling language has been collected in Section 2.3, *Requirements to a metamodelling language*, p. 34. In detail, these are *Simplicity*, *Formality*, *Homogeneity*, *Expressiveness*, *Appropriateness*, and *Autonomy*.

One of the critical issues upon the definition of such a metamodelling language is the procedure of how to specify such a language. Due to its bootstrapping characteristics, of course, it has been defined by itself and thus a meta-metamodel in the end. This is, however, not sufficient from a formal point of view. Because of that, the detailed procedure has initially been described in Section 2.4, *Procedure specifying the (self-describing) metamodelling language* M2L , p. 42.

Due to the complexity of this topic, a detailed running example has been necessary. The difficulty here was to find the right domain for it: On the one hand it should not be too simple as then it would not have been possible to illustrate most things. On the other

hand it should not be too complicate as then nobody would have understood the example at all. The criteria has been described in detail in Section 3.1, *Criteria for selecting a suitable running example*, p. 45. Finally we have decided to model dataflow algorithms as the running example. Although this running example is a nearly complete language, for a proof of concept it still seems to be too small. Hence, the usual proceeding for metamodelling languages had been followed: The proof of concept has been the metamodelling language itself.

Before we had been going into details in terms of defining the language engineering concepts, partially ordered multi-sets (pomsets) have been introduced in Chapter 4, *Pomsets in the context of metamodelling*, p. 61. These pomsets are crucial for the present work as they represent the only sound way to allow both sets and lists within one metamodelling approach.

Abstract words have been introduced based on pomsets. As defined in Definition 30 within Section 5.1, *M-graphs (Model-graphs)*, p. 81, abstract words have been formalised by *M-graphs* which are a Σ-labelled, directed, partially-ordered multi-graph. Similar to a *word* in formal languages, which can be defined independently of any grammar definition, an abstract word has also been defined independent of any metamodel. Thus, model and metamodel are strictly separated in the present approach.

Having M-graphs, an algebra has been defined on edges in Chapter 6, *Queries on abstract words - the Edge Algebra*, p. 95 to be able to formulate complex queries and predicates over abstract words. Due to that, *abstract languages* can be specified by a node predicate formulated as an Edge Algebra statement. Then, the set of all abstract words fulfilling the node predicate globally – i. e. for all nodes of the M-graph – are valid to the defined abstract language.

Based on the principle ability of formulating abstract languages, a more comfortable language for specifying abstract syntaxes has been defined in Chapter 7, *Abstract Syntaxes in M2L*, p. 119. It basically orients itself on common metamodelling approaches such as UML class diagrams, but besides a formal structural semantics it also comprises a set of additional concepts introduced. In particular, these are *conditional properties*, *context-sensitive domains*, *local keys*, *namespaces*, and *instantiating properties*.

Chapter 8, *Textual Concrete Syntaxes in M2L*, p. 153 will finally explain of how to define textual concrete syntaxes. Basically they are defined as templates for each concept out of the abstract syntax. Hereupon, a tight relationship between abstract and concrete syntax with any redundant definitions has been achieved. As soon as the concrete textual syntax has been defined, the metamodelling language *M2L* can be explained by itself in a textual and thus human-readable way. The resulting language will be defined in Chapter 9, *The overall specification of M2L*, p. 171 including all details thereof. Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291 shows the meta-metamodel on its own.

10.2. Evaluation

Before beginning to discuss future work, the advantages of the metamodelling language *M2L* introduced shall be summarised. The requirements having been defined in Section 2.3, *Requirements to a metamodelling language*, p. 34, shall therefore be reviewed. In detail, these are *Simplicity*, *Formality*, *Homogeneity*, *Expressiveness*, *Appropriateness*, and *Autonomy*.

10.2.1. Simplicity

"Using the metamodelling language should be as simple as possible. Especially as it should not only specify a language for implementation but should also be used as a *readable* documentation."

Common issues are easy to express. Besides the commonly known metamodelling concepts such multiplicities, a set of specific metamodelling constructs has been introduced to simplify creating new languages. In particular, these are *conditional properties*, *context-sensitive domains*, *local keys*, *namespaces*, and *instantiating properties*.

Rare issues may be more complex to express. Besides the additional amount of specific constructs is it possible to add any Edge Algebra statement within each metamodel. In detail there are four ways of adding an Edge Algebra statement: Firstly, any node predicate can be added as additional constraint for each concept; secondly, any node predicate can be added as an assumption for each property; thirdly, the domain of a property is defined by any edge function; and fourthly, arbitrary inferred properties, wherein the inferred value will again be defined by an edge function, may be defined.

Homogeneity of resulting languages. Although arbitrary languages can be created, there are some important aspects rendering the resulting languages homogeneous. In the present context, the most important aspect is the local key concept including namespaces, context-sensitive keys, etc. The way of encoding identifiers will always be the same as well. Besides that, canonical syntax will, of course, also always have the same structure.

No redundancy. The way of defining abstract and textual concrete syntaxes will be realised such that there will be no redundant specification. One example is the definition of multiplicities: Nothing will be mentioned about that in concrete syntax. Whereas it can be defined in e. g. EBNF whether a non-terminal will be repeatable or optional, such a restriction cannot be defined within the concrete syntax in the present approach as it can already be defined by abstract syntax.

Multiple views. Due to the strict separation of abstract and concrete syntaxes is it possible to define multiple concrete syntaxes based on the same abstract syntax. The metamodelling language *M2L* as well allows for a definition of multiple abstract syntaxes.

10.2.2. Formality

"The metamodelling language has to provide a clear semantics for all syntactical elements of the metamodelling language. This forms the basis for using this language for formal specifications as could be done by a formal grammar."

Formal syntactical definition. This is the real main topic of the present work. The syntax of *M2L* will, of course, be defined in a formal way. This will finally be realised by the meta-metamodel itself.

Formal semantics. The semantics of *M2L* will also be defined in a formal way. Note that in this context, only a structural semantics shall be discussed as within the present work focus shall be on the syntactical aspects of a metamodel.

Bootstrapping. It has been shown that the metamodelling language *M2L* has been defined in a bootstrapping way. Note that this approach does not only take abstract syntax into account but also textual concrete syntax. Thus, also the concrete syntax will be incorporated into the bootstrapping mechanism. For this purpose, it will be required to understand both

the abstract and concrete syntax definition in order to be able to read the meta-metamodel in Appendix A, *Meta-Metamodel – The Metamodel of M2L*, p. 291.

10.2.3. Homogeneity

"All four metamodel aspects (abstract syntax, concrete syntax, process definition, and semantics) must fit together in a formal way. A fully automated tool generation is impossible without a homogeneity like this."

The present work focuses on the syntactical aspects of metamodelling. Thus, this requirement will only be fulfilled for the first two aspects, namely abstract and concrete syntax: When using *M2L*, concrete syntax can smoothly be integrated in abstract syntax. Thus, no additional glue code will be necessary when defining a language.

Abstract syntax in the centre. Both abstract syntax as well as abstract words are in the very centre of the present approach.

Concrete syntaxes based on abstract syntax. Whereas abstract syntax stands for its own, a concrete syntax will always be formed on an abstract one.

Process definition based on abstract syntax. As the present approach concentrates on the syntactical parts of a language, this requirement will not be relevant.

Semantics based on abstract syntax. As the present approach concentrates on the syntactical parts of a language, this requirement will not be relevant.

10.2.4. Expressiveness

"Informally, each language to be described can be described by this metamodelling language. If a metamodelling language does not fulfil such a requirement, every use of this metamodelling language holds the risk that it will be impossible to express desired future extensions."

Although the expressive power of neither Edge Algebra nor the metamodelling language *M2L* will explicitly be discussed in a formal way, there were no constraints to be defined that could not be expressed - neither in the running example nor in the meta-metamodel of *M2L*. This is, however, not a proof but an indication that the expressiveness of *M2L* will be suitable for relevant languages.

10.2.5. Appropriateness

"Those metamodelling language support mechanisms necessary from a methodological point of view. Here, for example metamodel evolution and metamodel modularisation are important issues."

Intuitive for language and product model engineers. Due to the fact that most commonly known concepts of UML class diagrams are also available within *M2L*, is it easy to start modelling with the help of *M2L*. Due to pomsets, the huge number of special operators and Edge Algebra itself, however, a detailed study of the language is required.

Suitable for creating models prior to a metamodel. It is possible to define models without a metamodel due to the semi-structured approach. When talking about an abstract word, an M-graph will be defined, which can, of course, be written down by using the

canonical textual representation. When talking about a textual representation, this can be realised by simply writing text.

No "contamination" of technical details. The most important construct herein are *instantiating properties*. The metamodel can easily define a complex re-use mechanism without introducing additional concepts or constraints by using these properties. It can simply be realised by a modifier for properties. The second construct helping to simplify the metamodel is the concept for local keys. This may appear as having no consequence because in most cases such consistency constraints are totally skipped within a metamodel due to complexity.

From rough to detailed specification. Herein, many aspects play a role. First of all, the possibility of defining models before metamodels helps during a language engineering process. Secondly, abstract syntaxes can be defined before having to consider concrete syntaxes. Finally, complex consistency constraints can be added later on without any problems. The next point of customising canonical concrete syntax will also help herein.

Customising of canonical concrete syntaxes. Models can be written down in a suitable way without having to define a concrete syntax definition due to canonical concrete syntax. Afterwards, a special textual syntax can stepwise be developed for the language by replacing the canonical syntax definition.

Language modularisation. As the abstract syntax definition is entirely based on restricting abstract words, a language modularisation can easily be realised. When combining two metamodels, just the resulting constraints of both metamodels must hold. In order to render the relationship more explicit, external concepts can be defined in addition.

Supporting language evolution. If a strong relationship between model and metamodel exists, one of the major issues of language evolution will arise: How will a model be described which is currently being migrated? In such a situation it is neither part of the original nor the new metamodel. In our formal approach a language, which is the union of both the old and the new metamodel, can easily be described. Complex migration rules can additionally be defined by using the Edge Algebra. This approach does, however, not support language evolution in an explicit way.

Suitable for generating comprehensive development tools. Although the present work discusses an implementation of a framework based on *M2L*, a framework called *OOMEGA* already exists, implementing most of the functionally introduced by *M2L*. Please refer to Section 10.3.1, OOMEGA *and* metaMODELS.org, p. 276 for further details.

10.2.6. Autonomy

"The metamodelling language is independent of a concrete tool implementation platform. EMF for example is closely related to Java and Eclipse which makes it difficult to use EMF in other environments such as C#."

Although an implementation of *M2L* in the form of *OOMEGA* exists, all concepts are made such that there are no relationships to any programming languages. *M2L* is instead totally based on a mathematical formalism.

10.3. Outlook

Although the requirements defined in Section 2.3, *Requirements to a metamodelling language*, p. 34 have been achieved, there is still a lot of work to do. This very final section shall give an idea of what the issues are that are still open. Besides this, the implementation *OOMEGA* [OOMEGA, 2010] as well as the metamodelling platform *metaMODELS.org* [Metamodels.org, 2010], which already allows a professional use of the metamodelling language *M2L*, shall be introduced as well.

10.3.1. *OOMEGA* and *metaMODELS.org*

OOMEGA's modelling environment is an extension to the Eclipse platform. It provides an implementation of the presented metamodelling language *M2L* and allows you to define arbitrary modelling languages. Based on such language definitions a full featured Eclipse plug-in is generated in order to create according models. As described in [OOMEGA, 2010], in detail the features are:

- **IDE/Eclipse Integration** MDE is most often applied to software projects. Then, models and traditional source code complement one another and together they form the software product. Models actually become a part of the working software within MDE - in contrast to the original CASE methodology. It is therefore important that models can be edited in the standard IDE. Thanks to OOMEGA there's no tool barrier. Both, models and code can be edited.
- **Textual Modelling** OOMEGA instantly provides textual model editors for your domain-specific languages. Those Eclipse editors realise both, the background parsing strategy as well as the MVC pattern. Hence, your textual DSL can be complemented with graphical or form-based editors and the very same model can be edited with different editors concurrently. At the same time, the textual editors will remain comfortable, i. e. any word can be typed just as in a standard Java editor. Obviously it comprises features such as syntax highlighting and hyper-linking.
- **Database Support** Especially when a large system is to be designed and models are becoming increasingly larger, clients can rely on OOMEGA's database support. OOMEGA provides a common API to db4objects, Hibernate (thus any SQL database) as well as the object database Versant. Moreover, an in-memory ODBMS is implemented. This is why the database back-end can be exchanged any time and the appropriate technology for storing your metamodels and models can be chosen.
- **Team Collaboration** OOMEGA technically supports teamwork in two alternative, but complementary ways: interactive modelling and local repositories. The former is a highly interactive modelling environment based on a central model database. Whenever committing your changes, others will immediately notice your work as their editors will be notified by OOMEGA's Client/Server protocol. Local repositories comprising the well-known Update/Commit/Merge operations are supported as well. CVS or SVN may simply be used and your models can be merged on the basis of appropriate textual representations.
- **Model-to-Text Transformations** Model-to-Text (M2T) transformations are mainly required in MDE projects. OOMEGA offers a built-in code generation engine that is based on Java Server Pages (JSP) technology. As an alternative, clients are very welcome to use openArchitectureWare. OOMEGA's Java Model Access API and query-language are used to explore models in the context of JSP or Xpand templates. Hence,

whatever code generator will be preferred, it will provide more features than simple template processing.

- **Model-to-Model Transformations** OOMEGA offers Model-to-Model (M2M) transformations by supporting the ATLAS Transformation Language (ATL). ATL is a prominent open source project providing a M2M transformation language and an interpreter therefore.
- **Java Model Access API / Query Language** One of OOMEGA's major strengths is an object persistence solution that provides an easy-to-use, powerful and standardised Java API to db4objects, Hibernate, Versant and OOMEGA's in-memory ODBMS. The Java Model Access API does not only offer a flexible query-language based on Edge Algebra but also well-engineered features such as transaction management, nested transactions, event notifications, cascading deletions, and dynamic access via reflection.
- **Automated Builds** Modern software and MDE projects typically rely on automated builds. Apache Ant and Maven are wide-spread build tools that offer fully-automated, repeatable and customisable software builds. M2M/M2T transformations are typically part of a build cycle, thus Ant tasks and Maven Mojos for ATL, oAW and OOMEGA's generator are offered.
- **XML Support** In particular for exchanging information with other software systems it is quite important to have an XML binding for metamodels and models. XML Schema (XSD) documents can be transformed to metamodels and vice versa. XML documents can be transformed to models and vice versa.

Besides OOMEGA, a metamodelling platform called *metaMODELS.org* has already been established. It is possible to create your own metamodels based on *M2L*. As all metamodels are available within one repository, these metamodels may include all others. The basic idea behind *metaMODELS.org* is to create a set of common-sense metamodels thus facilitating an exchange of models.

All in all, a picture as illustrated in Figure 10.1 will form: Edge Algebra forms the theoretical foundation for metamodelling. Based on Edge Algebra, the metamodelling language *M2L* will be defined. The open-source project *OOMEGA* provides both a tool for defining your own metamodel in *M2L* as well as a platform for creating corresponding models in a database-centric and multi-user tooling environment. Finally, *metaMODELS.org* has been established in the Internet as a platform for a collaborative development of metamodels. It had been implemented by using the tooling environment OOMEGA itself for creating metamodels. These metamodels may again be instantiated by way of OOMEGA.

10.3.2. Future work

In particular due to the fact that the present approach provides a powerful formal basis for metamodelling, many questions arise that are still open. The most important issues shall be discussed in the following:

1. **Process definition and semantics.** The term *metamodel* has been established at the beginning of the present work as being the aggregation of the four aspects of *abstract syntax*, *concrete syntax*, *process definition*, and *semantics*. The former two have been defined within the present work. The latter two are still open.
2. **Abstract views.** Views are a very important concept in databases. Views make it possible to create abstractions of the complete model such that only the necessary

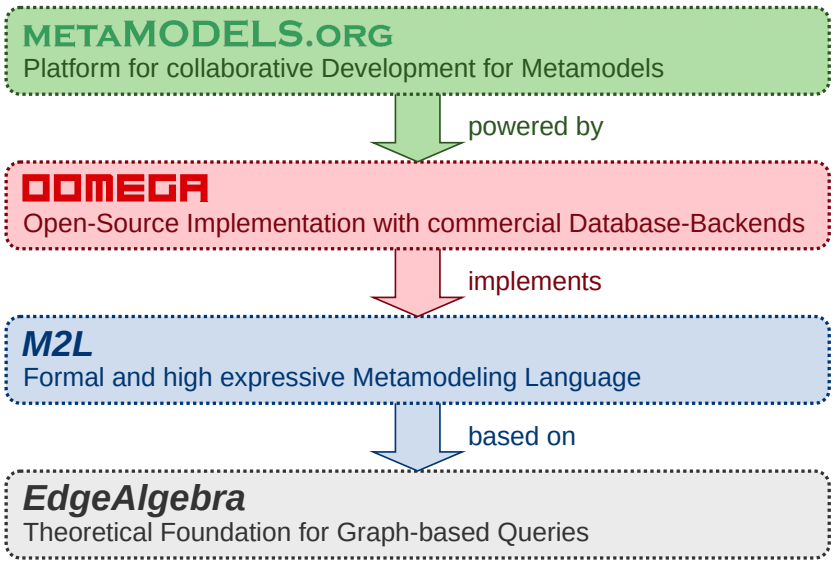


Figure 10.1.: Overview of technologies.

information will be presented – even in a specific way if necessary. In the domain of language engineering this is a very important issue. This aspect has, however, been omitted in the present work.

3. **Comparing different metamodelling approaches** It is now possible to represent a model independent of a specific metamodelling approach due to M-graphs. This makes facilitates a comparison of different metamodelling approaches in a formal way. Nonetheless will such a comparison – even with a single approach – go far beyond the scope of the present work.

4. **Behavioural semantics for abstract and concrete syntaxes.** The present approach concentrates entirely on a static view of abstract and concrete syntaxes. Thus, this approach states nothing about the influence of such a metamodel definition on the behaviour of metamodel frameworks when creating a corresponding model. Open questions are e. g. what is going to happen when an instance of a template is being altered by a setter.

5. **Formal expressiveness of *M2L*.** As mentioned above, the expressiveness of *M2L* or even the Edge Algebra will not be discussed within the present work. Interesting is, what kind of textual languages can be described. It can be assumed that at least all context-sensitive languages can be described. A formal proof is still open.

6. **Unambiguous textual syntaxes.** When defining textual concrete syntaxes is it important for the resulting language to be unambiguous. Thus, one textual representation will only have one valid abstract syntax. In particular, when combining partial languages to a mega-language, this aspect will turn out to be crucial.

7. **Efficient parser.** Even if a textual language definition is unambiguous, it is still open, how to write an efficient parser for such a language definition. In this context is it also interesting, what the minimal complexity of such a parsing process is.

8. **Analysis of languages.** For both abstract as well as textual concrete languages is it important to know about decidability and complexity of common problems such as the

word problem or equality and emptiness of languages. Another interesting question in this context is, how to find a minimal abstract word for a language, if any.

9. **Theorems for pomset and Edge Algebra operators.** Although a huge set of operators has been introduced for both pomsets and edge functions, a set of theorems for restructuring formulas is still missing.

10. **Query optimisation.** It is important for a query – expressed by an Edge Algebra statement – to be optimised before being executed in the context of databases containing a large abstract word. For relational algebra, for example, such optimisation rules are already available. It has also been verified in this context whether Edge Algebra is suitable for such an optimisation.

11. **Type system for Edge Algebra.** It is useful in many situations to know what a possible type of an Edge Algebra statement is. Thus, what the concepts are, the resulting vertices are labelled with. This is e. g. important in case of a context-sensitive domain: In general, there is no explicit definition of the resulting type.

12. **Language modularisation.** Although language definitions can be modularised by the present approach, this mechanism is a very basic solution, however. A more sophisticated way of decomposing a language definition into language modules is still open.

13. **Language evolution.** As mentioned above, Edge Algebra will help when defining a migration of models in order to evolve a language. An explicit approach allowing for the creation of a language definition – especially if a huge number of models already exists – is, however, still open.

Bibliography

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Number ISBN: 0-20153-771-0. Addison Wesley.

[Agrawal et al., 2006] Agrawal, A., Karsai, G., Neema, S., Shi, F., and Vizhanyo, A. (2006). The design of a language for model transformations. *Journal on Software and System Modeling*, 5(3):261–288. Introduction of GReAT.

[Alanen et al., 2007] Alanen, M., Lundkvist, T., and Porres, I. (2007). Creating and reconciling diagrams after executing model transformations. *Science of Computer Programming*, 68(3):128–151.

[Alanen and Porres, 2003] Alanen, M. and Porres, I. (2003). A relation between context-free grammars and meta object facility metamodels. Technical Report 606, Turku Centre for Computer Science.

[Ambler, 1999] Ambler, S. W. (1999). Mapping objects to relational databases. Technical report, AmbySoft Inc.

[Aquintos, 2010] Aquintos (2010). Website of aquintos. http://www.aquintos.info.

[Artwork, 2002] Artwork (2002). Website of the artwork project. http://artwork.in.tum.de.

[Artwork, 2003a] Artwork (2003a). Architektur und implementierung der engineering workbench. abschlussbericht des projekts artwork.

[Artwork, 2003b] Artwork (2003b). Das produkt- und prozessmodell der engineering workbench. abschlussbericht des projekts artwork.

[AutoFocus, 2006] AutoFocus (2006). Autofocus model and model api.

[Autosar, 2006] Autosar (2006). Autosar template uml profile and modeling guide. http://www.autosar.org/download/AUTOSAR_TemplateModelingGuide.pdf.

[Baar, 2006] Baar, T. (2006). Correctly defined concrete syntax for visual modeling languages. In *MoDELS*, pages 111–125.

[Balasubramanian et al., 2007] Balasubramanian, K., Schmidt, D. C., Molnar, Z., and Ledeczi, A. (2007). Component-based system integration via (meta)model composition. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 93–102, Washington, DC, USA. IEEE Computer Society.

[Becker et al., 2005] Becker, S. M., Haase, T., and Westfechtel, B. (2005). Model-based a-posteriori integration of engineering tools for incremental development processes. *Software and System Modeling*, 4(2):123–140.

[Becker et al., 2007] Becker, S. M., Herold, S., Lohmann, S., and Westfechtel, B. (2007). A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and System Modeling*, 6(3):287–315.

[Berry et al., 2000] Berry, G., Bouali, A., Fornari, X., Ledinot, E., Nassor, E., and de Simone, R. (2000). Esterel: a formal method applied to avionic software development. *Sci. Comput. Program.*, 36(1):5–25.

[Bezivin and Kurtev, 2005] Bezivin, J. and Kurtev, I. (2005). Model-based technology integration with the technical space concept. In *Metainformatics Symposium 2005*, Esbjerg, Denmark.

[Biegl, 1995] Biegl, C. (1995). Multigraph: an architecture for model-integrated computing. In *ICECCS '95: Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*, page 361, Washington, DC, USA. IEEE Computer Society.

[Biron et al., 2004] Biron, P. V., Permanente, K., and Malhotra, A. (2004). Xml schema part 2: Datatypes second edition. Technical report, W3C.

[Blanc et al., 2005] Blanc, X., Ramalho, F., and Robin, J. (2005). Metamodel reuse with mof. In *MoDELS*, pages 661–675.

[Boronat and Meseguer, 2008] Boronat, A. and Meseguer, J. (2008). An algebraic semantics for mof. In *FASE*, pages 377–391.

[Braun and Marschall, 2003] Braun, P. and Marschall, F. (2003). Botl - the bidirectional object oriented transformation language. Technical Report TUM-I0307, TUM.

[Bray et al., 2006] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2006). Extensible markup language (xml) 1.1 (second edition). Technical report, W3C.

[Breu et al., 1997] Breu, R., Hinkel, U., Hofmann, C., Klein, C., Paech, B., Rumpe, B., and Thurner, V. (1997). Towards a formalization of the unified modeling language. In *ECOOP*, pages 344–366.

[Broy, 2006] Broy, M. (2006). Challenges in automotive software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA. ACM.

[Broy et al., 2008] Broy, M., Feilkas, M., Grünbauer, J., Gruler, A., Harhurin, A., Hartmann, J., Penzenstadler, B., Schätz, B., and Wild, D. (2008). Umfassendes architekturmodell für das engineering eingebetteter software-intensiver systeme. Technical Report TUM-I0816, Technische Universität München.

[Broy et al., 2010] Broy, M., Feilkas, M., Herrmannsdörfer, M., Merenda, S., and Ratiu, D. (2010). Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE, Special Issue on Aerospace and Automotive Software*, 98(4):526–545.

[Broy and Stølen, 2001] Broy, M. and Stølen, K. (2001). *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Brucker and Wolff, 2002] Brucker, A. D. and Wolff, B. (2002). A proposal for a formal ocl semantics in isabelle/hol. In *TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, pages 99–114, London, UK. Springer-Verlag.

[Budinsky et al., 2003] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. J. (2003). *Eclipse Modeling Framework*. Number ISBN 978-0131425422 in Eclipse. Prentice Hall International. EMF.

[Bézivin, 2005] Bézivin, J. (2005). On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188.

[Bézivin et al., 2005] Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., and Kurtev, I. (2005). Bridging the generic modeling environment (gme) and the eclipse modeling framework (emf). In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05*, San Diego, California, USA.

[Bézivin and Lemesle, 1999] Bézivin, J. and Lemesle, R. (1999). Reflective modeling schemes. In *OOPSLA Workshop on Reflection and Software Engineering (OORaSE'99)*.

[Cabibbo and Carosi, 2005] Cabibbo, L. and Carosi, A. (2005). Managing inheritance hierarchies in object/relational mapping tools. In *CAiSE*, pages 135–150. O/R mapping.

[Cabibbo and Porcelli, 2003] Cabibbo, L. and Porcelli, R. (2003). M2orm2: A model for the transparent management of relationally persistent objects. In *DBPL*, pages 166–178. O/R mapping.

[Caspi et al., 1987] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. (1987). Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 178–188, New York, NY, USA. ACM.

[Cattell et al., 1999] Cattell, R., Barry, D., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F. (1999). *The Object Data Standard: ODMG 3.0*. Number ISBN: 1-55860-647-4. Morgan Kaufmann Publishers.

[Clark et al., 2002] Clark, T., Evans, A., and Kent, S. (2002). A metamodel for package extension with renaming. In *UML 2002 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 305–320. Springer Berlin / Heidelberg.

[Cleenewerck and Kurtev, 2007] Cleenewerck, T. and Kurtev, I. (2007). Separation of concerns in translational semantics for dsls in model engineering. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 985–992, New York, NY, USA. ACM Press.

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

[Cokus and Pericas-Geertsen, 2005a] Cokus, M. and Pericas-Geertsen, S. (2005a). Xml binary characterization properties. Technical report, W3C.

[Cokus and Pericas-Geertsen, 2005b] Cokus, M. and Pericas-Geertsen, S. (2005b). Xml binary characterization use cases. Technical report, W3C.

[Dar and Agrawal, 1993] Dar, S. and Agrawal, R. (1993). Extending sql with generalized transitive closure. *IEEE Trans. on Knowl. and Data Eng.*, 5(5):799–812.

[Davis, 2003] Davis, J. (2003). Gme: the generic modeling environment. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented program-*

ming, systems, languages, and applications, pages 82–83, New York, NY, USA. ACM Press.

[Drey et al., 2008] Drey, Z., Faucher, C., Fleurey, F., Mahé, V., and Vojtisek, D. (2008). *Kermeta language - Reference manual*. IRISA Triskell Project.

[Ebert and Franzke, 1994] Ebert, J. and Franzke, A. (1994). A declarative approach to graph based modeling. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 38–50, London, UK. Springer-Verlag.

[Eclipse, 2010a] Eclipse (2010a). Website of eclipse GMF. http://www.eclipse.org/modeling/gmf/.

[Eclipse, 2010b] Eclipse (2010b). Website of the eclipse development platform. http://www.eclipse.org.

[Edwards et al., 2004] Edwards, J., Jackson, D., and Torlak, E. (2004). A type system for object models. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 189–199, New York, NY, USA. ACM Press.

[Efftinge and Völter, 2006] Efftinge, S. and Völter, M. (2006). oaw xtext - a framework for textual dsls. In *Worksjop Modeling Symposium of Eclipse Summit Conference*.

[Emerson and Sztipanovits, 2006] Emerson, M. and Sztipanovits, J. (2006). Techniques for metamodel composition. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06) at OOPSLA'06*.

[Esterel, 2010] Esterel (2010). Esterel technologies webpage. http://www.esterel-technologies.com.

[Estublier et al., 2005] Estublier, J., Vega, G., and Ionita, A. D. (2005). Composing domain-specific languages for wide-scope software engineering applications. In *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 69–83. Springer Berlin / Heidelberg.

[Evermann and Wand, 2005] Evermann, J. and Wand, Y. (2005). Toward formalizing domain modeling semantics in language syntax. *IEEE Trans. Softw. Eng.*, 31(1):21–37.

[Falkowski, 2005] Falkowski, K. (2005). Modelltransformationsansätze im kontext modellgetriebener softwareentwicklung. Master's thesis, Universität Koblenz-Landau.

[Fallside and Walmsley, 2004] Fallside, D. C. and Walmsley, P. (2004). Xml schema part 0: Primer second edition. Technical report, W3C.

[Farail et al., 2006] Farail, P., Gaufillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Crégut, X., and Pantel, M. (2006). The TOPCASED project: a Toolkit in Open source for Critical Aeronautic SystEms Design. In *Embedded Real Time Software (ERTS)*.

[Fleurey et al., 2007] Fleurey, F., Breton, E., Baudry, B., Nicolas, A., and Jézéquel, J.-M. (2007). Model-driven engineering for software migration in a large industrial context. In *MoDELS*, pages 482–497.

[France et al., 1997] France, R. B., Bruel, J.-M., Larrondo-Petrie, M. M., Grant, E. S., and Saksena, M. (1997). Towards a rigorous object-oriented analysis and design method. In *ICFEM*, pages 7–16.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Number ISBN: 0-201-63361-2. Addison Wesley.

[Gargantini et al., 2006] Gargantini, A., Riccobene, E., and Scandurra, P. (2006). Deriving a textual notation from a metamodel: an experience on bridging modelware and grammarware. In *3M4MDA 2006 - European Workshop on Milestones, Models and Mappings for Model-Driven Architecture - European Conference on Model Driven Architecture*.

[Geer, 2005] Geer, D. (2005). Will binary xml speed network traffic? *Computer*, 38(4):16–18.

[Geisler et al., 1998] Geisler, R., Klar, M., and Pons, C. (1998). Dimensions and dichotomy in metamodeling. In *Proceedings of the Third BCS-FACS Northern Formal Methods Workshop*. Springer Verlag.

[Gitzel et al., 2004] Gitzel, R., Ott, I., and Schader, M. (2004). Ontological metamodel extension for generative architectures (omega). Technical report, University of Mannheim.

[Goldman and Lenkov, 2005] Goldman, O. and Lenkov, D. (2005). Xml binary characterization. Technical report, W3C.

[Gottlob et al., 1990] Gottlob, G., Kappel, G., and Schrefl, M. (1990). Semantics of object-oriented data models - the evolving algebra approach. In *East/West Database Workshop*, pages 144–160.

[Greenfield and Short, 2004] Greenfield, J. and Short, K. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing, Inc.

[Grimm, 2005] Grimm, K. (2005). Software-technologie im automobil. In Liggesmeyer, P. and Rombach, D., editors, *Software-Engineering eingebetteter Systeme: Grundlagen – Methodik – Anwendungen*, chapter 16, pages 407–430. Spektrum, Heidelberg.

[Grumbach and Milo, 1995] Grumbach, S. and Milo, T. (1995). An algebra for pomsets. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 191–207, London, UK. Springer-Verlag.

[Gruschko et al., 2007] Gruschko, B., Kolovos, D., and Paige, R. (2007). Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution*.

[Guerra et al., 2005] Guerra, E., Diaz, P., and de Lara, J. (2005). A formal approach to the generation of visual language environments supporting multiple views. *VLHCC*, pages 284–286.

[Günzler, 2005] Günzler, A. (2005). *Integrationskonzepte in der modellbasierten Produktentwicklung*. PhD thesis, Technische Universität München.

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320.

[Harel and Rumpe, 2004] Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"?. *IEEE Computer*, 37(10):64–72.

[Hayes and McBride, 2004] Hayes, P. and McBride, B. (2004). Rdf semantics. Technical report, W3C. Ontologies.

[Hedin, 2000] Hedin, G. (2000). Reference attributed grammars. *Informatica (Slovenia)*, 24(3).

[Henderson-Sellers and Barbier, 1999] Henderson-Sellers, B. and Barbier, F. (1999). Black and white diamonds. In *UML*, pages 550–565.

[Hepp et al., 2006] Hepp, M., Bachlechner, D., and Siorpaes, K. (2006). Ontowiki: community-driven ontology engineering and ontology usage based on wikis. In *WikiSym '06: Proceedings of the 2006 international symposium on Wikis*, pages 143–144, New York, NY, USA. ACM Press.

[Herrmannsdoerfer et al., 2008] Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2008). Automatability of coupled evolution of metamodels and models in practice. In *MoDELS 2008, Model Driven Engineering Languages and Systems, 11th International Conference*.

[Hibernate, 2010] Hibernate (2010). Website of the hibernate project. http://www.hibernate.org.

[Holt et al., 2002] Holt, R., Schürr, A., Sim, S. E., and Winter, A. (2002). Graph exchange language.

[Houssais, 2002] Houssais, B. (2002). The synchronous programming language signal, a tutorial. Technical report, IRISA.

[Hull, 1986] Hull, R. (1986). A survey of theoretical research on typed complex database objects. In *XP7.52 Workshop on Database Theory*.

[Hölzl, 2010] Hölzl, F. (2010). Website of autofocus iii - an engineering tool for embedded systems. http://af3.in.tum.de.

[IBM, 2004] IBM (2004). Emfatic.

[IBM, 2010] IBM (2010). Ibm rational doors. http://www.ibm.com/software/awdtools/doors/.

[IntentionalSoftware, 2010] IntentionalSoftware (2010). Website of intentional software. http://www.intentsoft.com.

[ISO, 1996] ISO (1996). Iso 14977: Extended ebnf.

[ISO, 2008] ISO (2008). Iso 9075: The standard query language.

[Jackson, 2006] Jackson, D. (2006). Alloy analyzer.

[Jaeschke and Schek, 1982] Jaeschke, G. and Schek, H.-J. (1982). Remarks on the algebra of non first normal form relations. In *PODS*, pages 124–138.

[Jouault et al., 2006a] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006a). Atl: a qvt-like transformation language. In *OOPSLA Companion*, pages 719–720.

[Jouault and Bézivin, 2006] Jouault, F. and Bézivin, J. (2006). Km3: a dsl for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, pages 171–185, Bologna, Italy.

[Jouault et al., 2006b] Jouault, F., Bézivin, J., and Kurtev, I. (2006b). Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA. ACM Press.

[Karsai et al., 2005] Karsai, G., Lang, A., and Neema, S. (2005). Design patterns for open tool integration. *Software and System Modeling*, 4(2):157–170.

[Karsai et al., 2004] Karsai, G., Maroti, M., Ledeczi, A., Gray, J., and Sztipanovits, J. (2004). Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling*, 12:263–278.

[Knuth, 1968] Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145.

[Knuth, 1971] Knuth, D. E. (1971). Correction: Semantics of context-free languages. *Mathematical Systems Theory*, 5(1):95–96.

[Knuth, 1990] Knuth, D. E. (1990). The genesis of attribute grammars. In *WAGA: Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12, New York, NY, USA. Springer-Verlag New York, Inc.

[Kugele et al., 2007] Kugele, S., Tautschnig, M., Bauer, A., Schallhart, C., Merenda, S., Haberl, W., Kühnel, C., Müller, F., Wang, Z., Wild, D., Rittmann, S., and Wechs, M. (2007). COLA – the component language. Technical Report TUM-I0714, Technischen Universität München.

[Kurtev et al., 2002] Kurtev, I., Bézivin, J., and Aksit, M. (2002). Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine.

[Kurtev et al., 2006] Kurtev, I., Bézivin, J., Jouault, F., and Valduriez, P. (2006). Model-based dsl frameworks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 602–616, New York, NY, USA. ACM Press.

[Kyas et al., 2005] Kyas, M., Fecher, H., de Boer, F. S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., and Kugler, H. (2005). Formalizing uml models and ocl constraints in pvs. *Electronic Notes in Theoretical Computer Science*, 115:39 – 47. Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004).

[Königs and Schürr, 2006] Königs, A. and Schürr, A. (2006). Mdi: A rule-based multi-document and tool integration approach. *Software and System Modeling*, 5(4):349–368.

[Lämmel, 2004] Lämmel, R. (2004). Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*.

[Ledeczi et al., 2001] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17.

[Margaria, 2005] Margaria, T. (2005). Web services-based tool-integration in the ETI platform. *Software and System Modeling*, 4(2):141–156.

[Markovi and Baar, 2006] Markovi, S. and Baar, T. (2006). An OCL semantics specified with QVT. In *Model Driven Engineering Languages and Systems*, volume 4199/2006 of *Lecture Notes in Computer Science*, pages 661–675. Springer Berlin / Heidelberg.

[Mathworks, 2010] Mathworks (2010). Mathworks webpage. http://www.mathworks.com.

[Merenda, 2005] Merenda, S. M. (2005). Sdf - structured data format: Binärcodierte repräsentation objektorientiert strukturierter daten. Master's thesis, Technische Universität München.

[Metamodels.org, 2010] Metamodels.org (2010). Website of metamodels.org - a platform for collaborative development of metamodel and datamodels. http://www.metamodels.org.

[Microsystems, 2002] Microsystems, S. (2002). Java metadata interface(jmi) specification. JSR 040 Java Community Process.

[Microsystems, 2010] Microsystems, S. (2010). The java website. http://java.sun.com.

[Minas, 2010] Minas, M. (2010). Website of DiaGen. http://www.unibw.de/inf2/DiaGen.

[ModelCVS, 2010] ModelCVS (2010). ModelCVS webpage. http://www.modelcvs.org/.

[Muller et al., 2006] Muller, P.-A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard4, S., and Jézéquel, J.-M. (2006). Model-driven analysis and synthesis of concrete syntax. In *MoDELS*, pages 98–110.

[Muller and Hassenforder, 2005] Muller, P.-A. and Hassenforder, M. (2005). Hutn as a bridge between modelware and grammarware. In *WISME Workshop, MODELS / UML'2005*, Montego Bay, Jamaica.

[Nardi and Brachman, 2003] Nardi, D. and Brachman, R. J. (2003). An introduction to description logics. pages 1–40.

[Nardi et al., 2003] Nardi, D., Brachman, R. J., Baader, F., and Nutt, W. (2003). *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA.

[Oliveira et al., 2005] Oliveira, H., Murta, L., and Werner, C. (2005). Odyssey-VCS: a flexible version control system for UML model elements. In *SCM '05: Proceedings of the 12th International Workshop on Software Configuration Management*, pages 1–16, New York, NY, USA. ACM.

[OMG, 2004] OMG (2004). Human-usable textual notation (hutn) specification. OMG Document formal/04-08-01.pdf.

[OMG, 2005a] OMG (2005a). Mof qvt final adopted specification. OMG Document ptc/05-11-01.

[OMG, 2005b] OMG (2005b). Xml metadata interchange specification. OMG Document formal/05-05-06.

[OMG, 2006a] OMG (2006a). Meta object facility (mof) core specification 2.0. OMG Document formal/06-01-01.

[OMG, 2006b] OMG (2006b). Object constraint language 2.0 specification. OMG Document formal/06-05-01.

[OMG, 2007] OMG (2007). Unified modeling language: Superstructure. OMG Document formal/07-02-05.pdf.

[OOMEGA, 2010] OOMEGA (2010). Website of oomega - a framework for model-based software engineering. http://www.oomega.net.

[Patel-Schneider et al., 2004] Patel-Schneider, P. F., Hayes, P., and Horrocks, I. (2004). Owl web ontology language semantics and abstract syntax. Technical report, W3C.

[Pavlidis, 1972] Pavlidis, T. (1972). Linear and context-free graph grammars. *J. ACM*, 19(1):11–22.

[Poernomo, 2006] Poernomo, I. (2006). A type theoretic framework for formal metamodelling. In *Architecting Systems with Trustworthy Components*, volume 3938/2006 of *Lecture Notes in Computer Science*, pages 262–298. Springer Berlin / Heidelberg.

[Pratt, 1985] Pratt, V. R. (1985). The pomset model of parallel processes: Unifying the temporal and the spatial. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 180–196, London, UK. Springer-Verlag.

[Rahm and Bernstein, 2006] Rahm, E. and Bernstein, P. A. (2006). An online bibliography on schema evolution. *SIGMOD Record*, 35(4):30–31.

[Reichmann et al., 2004] Reichmann, C., Kühl, M., Graf, P., and Müller-Glaser, K. D. (2004). Generalstore - a case-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In *ECBS '04: Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, page 225, Washington, DC, USA. IEEE Computer Society.

[Rosenberg and Salomaa, 1997] Rosenberg, G. and Salomaa, A. (1997). *Handbook of Formal Languages Vol. 1 - Word Language Grammar*. Number ISBN: 3-540-60420-0. Springer Verlag.

[Rozenberg, 1997] Rozenberg, G., editor (1997). *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

[Saeki and Kaiya, 2006] Saeki, M. and Kaiya, H. (2006). On relationships among models, meta models and ontologies. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06) at OOPSLA'06*.

[Scade, 2010] Scade (2010). Website of SCADE. http://www.esterel-technologies.com/products/scade-suite/.

[Simonyi, 1995] Simonyi, C. (1995). The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research.

[Sprinkle and Karsai, 2004] Sprinkle, J. and Karsai, G. (2004). A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307.

[Sutherland, 1966] Sutherland, W. R. (1966). *The on-line graphical specification of computer procedures*. PhD thesis, Massachusetts Institute of Technology (MIT).

[Thompson et al., 2004] Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2004). Xml schema part 1: Structures second edition. Technical report, W3C.

[Tigris.org, 2010] Tigris.org (2010). Subversion. http://subversion.tigris.org/.

[Tolvanen, 2004] Tolvanen, J.-P. (2004). MetaEdit+: domain-specific modeling for full code generation demonstrated [GPCE]. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 39–40, New York, NY, USA. ACM.

[Unicode, 2010] Unicode (2010). Website of the unicode consortium. http://www.unicode.org.

[Vanderbilt, 2005] Vanderbilt (2005). *A Generic Modeling Environment: GME 5 User's Manual (Version 5.0)*. Vanderbilt University.

[Vanderbilt, 2010] Vanderbilt (2010). Website of the graph rewrite and transformation (great) tool suite. http://escher.isis.vanderbilt.edu/tools/get_tool?GReAT.

[Vector, 2010] Vector (2010). Vector eASEE webpage. http://www.vector.com/vi_easee_en,,223.html.

[Viehstaedt and Minas, 1995] Viehstaedt, G. and Minas, M. (1995). Diagen: A generator for diagram editors based on a hypergraph model. In *NGITS*.

[Wachsmuth, 2007] Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *ECOOP'07: Proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer Berlin / Heidelberg.

[Wang et al., 1997] Wang, D. C., Appel, A. W., Korn, J. L., and Serra, C. S. (1997). The zephyr abstract syntax description language. In *DSL*, pages 213–228.

[WikiOnt, 2010] WikiOnt (2010). Website of the wikiont project. http://sourceforge.net/projects/wikiont.

[Williams and Haggar, 2005] Williams, S. D. and Haggar, P. (2005). Xml binary characterization measurement methodologies. Technical report, W3C.

[Winter et al., 2002] Winter, A., Kullbach, B., and Riediger, V. (2002). An overview of the gxl graph exchange language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK. Springer-Verlag.

Appendix A

Meta-Metamodel – The Metamodel of M2L

This appendix shows the complete metamodel including abstract and textual concrete syntax. Since the language *M2L* is bootstrapped, the following listing is a textual instance of itself. For a detailed description of each and every concept please refer to Chapter 9, *The overall specification of M2L*, p. 171.

Listing A.1: Complete Metamodel of M2L

```
meta-metamodel "Metamodelling Language M2L" (M2L) {
  abstract syntax {
    metapackage ORG {
      metapackage Metamodels {
        metapackage BasicConcepts {
          anyconcept [Any] ::>
            lkey(Poset) : C:Identifier,
            ikey(Poset) : C:Identifier,
            ckey(List) := ≽^⇆P:composite.1 P:lkey,
            &composite(Set),
            &template[0..1],
            &includedContext(Poset)
            where |1 P:lkey| ≤ 1 ∧ |1 P:ikey| ≤ 1
              ∧ ⇆(* ⇃* P:composite) ≠ ⇆P:template ⇒ |P:lkey| ≥ 1
              ∧ poset? (σroot?.P:lkey)
              ∧ poset? (^(P:composite ↓ σ|P:lkey| = 0).P:lkey)
              ∧ |P:template| = 0 ⇒ |P:ikey| = 0
              ∧ |P:template| = 1 ⇒ P:lkey = P:ikey
              ∧ |⇆P:composite| ≤ 1
              ∧ ↻ ∉ P:composite.^P:composite
              ∧ ↻ ∉ P:template.^P:template
              ∧ (|P:template| = 1 ⇒ (
                P:template.P:composite = P:composite.P:template
                ∧ ∀C:c:(↻ ∈ c ⇔ P:template ∈ c)
                ∧ ∀P:p\{lkey, ckey, ikey}:
                  (P:template.p = p.1 (((↻ ↓ (⇆p.
                  ^((P:composite ⊎ ⇆P:composite) ↓ σ|P:template|)
```

```
28                    )).P:template) ⊕ ↻))
29              )) ;
30
31          enum Boolean = { True, False } ;
32          Natural!! ::= pred[0..1] : C:Natural ;
33          Interval!! ::=
34            lower[1..1] : C:Natural,
35            upper[0..1] : C:Natural
36            where |P:upper| = 1 ⇒
37              (number)P:lower ≤ (number)P:upper ;
38          Character!! ::= unicode[1..1] : C:Natural
39            where (number)unicode ≤ 1114111 ;
40          String!! ::> character[0..*](List) : C:Character ;
41          Identifier!! refines String ::> character[1..*] ;
42
43          [Named] ::>
44            (PK)name[1..1] : C:Identifier,
45            (K)alternativeName[0..*](Set) : C:Identifier ;
46          [UniquelyNamed] refines Named ::>
47            alternativeName[0..0] ;
48          Folder refines FolderEntry ::>
49            subfolder[0..*](Toset) : C:Folder,
50            &entry[0..*](Toset) : C:FolderEntry ;
51          [FolderEntry] refines Named ;
52        }
53
54        metapackage M2L {
55          MetamodelFolder refines Folder ::>
56            name := {{ Identifier("Active Metamodels") }},
57            &entry : C:Metamodel,
58            &activeMetamodel := μ^P:subfolder.P:entry.^P:basedOn
59            where |C:MetamodelFolder| = 1
60              ∧ ^P:subfolder.P:entry ∈ C:Metamodel
61              ∧ C:Metametamodel ∈ P:activeMetamodel ;
62          Metamodel refines FolderEntry ::>
63            &basedOn[0..*](Set) : C:Metamodel ⇵* ^⇆P:basedOn,
64            abstractSyntax[1..1] : C:AbstractSyntax,
65            concreteSyntax[0..*] : C:ConcreteSyntax,
66            &exportedMetapackage :=
67              P:abstractSyntax.P:exportedMetapackage,
68            &visibleMetapackage :=
69              (↻ ⊕ P:basedOn).P:exportedMetapackage ;
70          Metametamodel refines Metamodel ::>
71            name := {{ Identifier("Metamodeling Language M2L"),
72              Identifier(M2L) }},
73            &basedOn[0..0],
74            where |C:Metametamodel| = 1 ;
75
76          metapackage AbstractSyntax {
77            AbstractSyntax! ::>
78              metapackage[0..*](Set) : C:Metapackage,
79              &exportedMetapackage :=
```

```
80                P:metapackage.P:exportedMetapackage,
81              &visibleMetapackage :=
82                ⇆P:composite.P:visibleMetapackage ;
83            Metapackage! refines Named ::>
84              subpackage[0..*](Set) : C:Metapackage,
85              conceptDef[0..*](Set) : C:ConceptDef,
86              &exportedMetapackage :=
87                ↻ ⊕ (P:subpackage.P:exportedMetapackage),
88              &visibleMetapackage :=
89                ⇆P:composite.P:visibleMetapackage ;
90
91            Concept!! ::=
92              qualifiedName[2..*](List) : C:Identifier ;
93            Property!! refines Identifier ::= ;
94
95            ConceptDef! refines Named ::>
96              concept[1..1] : C:Concept,
97              &refines[0..*](Set) :
98                (P:visibleMetapackage.P:conceptDef) ↓* ^⇆P:refines,
99              isAbstract[1..1] : C:Boolean,
100             isComplete[1..1] : C:Boolean,
101             conceptType[1..1] : C:ConceptType,
102             propertyDef[0..*](Set) ↔ conceptDef : C:PropertyDef,
103             additionalConstraint[0..1] : C:Predicate,
104             &visibleMetapackage :=
105               ⇆P:composite.P:visibleMetapackage,
106             &includedContext := P:visibleMetapackage
107             where P:concept.P:qualifiedName = (≥^⇆P:composite
108               ↓ (C:Metapackage ⊎ C:ConceptDef)).P:name ;
109
110           AnyConceptDef! refines ConceptDef ::>
111             concept :=
112               {{ Concept(ORG.Metamodels.BasicConcepts.Any) }},
113             &refines[0..0],
114             isAbstract : C:Boolean.True,
115             isComplete : C:Boolean.False,
116             conceptType : C:ConceptType.Strong
117             where |⇆P:refines| = 0 ;
118           EnumerationConceptDef! refines ConceptDef ::>
119             enumElement[2..*] : C:EnumElementConceptDef,
120             &refines[0..0],
121             isAbstract : C:Boolean.True,
122             isComplete : C:Boolean.True,
123             conceptType : C:ConceptType.Attribute,
124             propertyDef[0..0],
125             additionalConstraint[0..0] ;
126           EnumElementConceptDef! refines ConceptDef ::>
127             &refines := ⇆P:enumElement,
128             isAbstract : C:Boolean.False,
129             isComplete : C:Boolean.True,
130             conceptType : C:ConceptType.Attribute,
131             propertyDef[0..0],
```

```
132                 additionalConstraint[0..0]
133                 where ⇆P:composite ∈ C:EnumerationConceptDef ;
134               ExternalConceptDef! refines ConceptDef ::>
135                 &refines[0..0],
136                 isAbstract : C:Boolean.False,
137                 isComplete : C:Boolean.False,
138                 conceptType : C:ConceptType.Strong,
139                 propertyDef[0..0],
140                 additionalConstraint[0..0] ;
141
142               PropertyDef! ::>
143                 &conceptDef[1..1] ↔ propertyDef : C:ConceptDef,
144                 assumption[0..1] : C:Predicate,
145                 property[1..1] : C:Property,
146                 opposite[0..*](Set) : C:Property,
147                 keyType[0..1] : C:KeyType,
148                 linkType[1..1] : C:LinkType,
149                 multiplicity[0..1] : C:Interval,
150                 pomsetRestriction[0..1] : C:PomsetType,
151                 domain[0..1] : C:Edge,
152                 inferredValue[0..1] : C:Edge,
153                 &includedContext := P:conceptDef.P:includedContext ;
154
155               enum ConceptType = { Strong, Weak, Attribute } ;
156               enum KeyType = {
157                 PrimaryLocalkey, AlternativeLocalkey
158               } ;
159               enum LinkType = {
160                 Reference, Composition, Instantiation
161               } ;
162               enum PomsetType = {
163                 Set, Bag, List, Toset, Poset, Pomset
164               } ;
165             }
166
167             metapackage ConcreteSyntax {
168               SyntaxIdentifier!! refines Identifier ;
169               [ConcreteSyntax!] refines Named ::=
170                 name : C:SyntaxIdentifier,
171                 alternativeName : C:SyntaxIdentifier,
172                 isDefault : C:Boolean,
173                 syntaxPackage[0..*](Set) : C:ConcreteSyntaxPackage ;
174               [ConcreteSyntaxPackage!] ::=
175                 &metapackage[1..1] : P:includedContext.
176                   (P:metapackage ⊎ P:subpackage),
177                 subpackage[0..*](Set) : C:ConcreteSyntaxPackage,
178                 concreteSyntaxDef[0..*](Set) : C:ConcreteSyntaxDef,
179                 &includedContext := ϵ(
180                     (⇆(P:concreteSyntax.P:syntaxPackage).
181                     P:abstractSyntax)
182                   ⊎
183                     (⇆P:subpackage.P:metapackage)
```

```
184                 ),
185               lkey := P:metapackage.P:lkey ;
186             [ConcreteSyntaxDef!] ::>
187               &conceptDef[1..1] : P:includedContext.P:conceptDef,
188               &includedContext :=
189                 ⇆P:concreteSyntaxDef.P:metapackage,
190               lkey := P:conceptDef.P:lkey ;
191
192             metapackage Textual {
193               TextualSyntax! refines ConcreteSyntax ::=
194                 syntaxPackage : C:TextualSyntaxPackage ;
195               TextualSyntaxPackage!
196                   refines ConcreteSyntaxPackage ::=
197                 subpackage : C:TextualSyntaxPackage,
198                 concreteSyntaxDef : C:TextualSyntaxDef ;
199               TextualSyntaxDef! refines ConcreteSyntaxDef ::>
200                 mainSyntaxTemplate[1..1] : C:SyntaxTemplate ;
201
202               SyntaxTemplate!! ::=
203                 templateElement[0..*](List) : C:TemplateElement ;
204               [TemplateElement!!] ;
205               [Terminal!!] refines TemplateElement ;
206               ProperTerminal!! refines Terminal ::=
207                 symbols[1..1] : C:String
208                 where |P:symbols.P:character| > 0 ;
209               WhitespaceTerminal!! refines Terminal ::=
210                 whitespace[1..1] : C:Whitespace ;
211               enum Whitespace = { Space, Newline } ;
212               NonTerminal!! refines TemplateElement ::=
213                 edge[1..1] : C:Edge,
214                 linkType[1..1] : C:LinkType,
215                 differingSyntax[0..1] : C:SyntaxIdentifier,
216                 starting[0..1] : C:SyntaxTemplate,
217                 prefix[0..1] : C:SyntaxTemplate,
218                 infix[0..1] : C:SyntaxTemplate,
219                 suffix[0..1] : C:SyntaxTemplate,
220                 ending[0..1] : C:SyntaxTemplate
221                 where P:linkType ∈ C:LinkType.Reference ⇒
222                   |P:differingSyntax| = 0 ;
223               Option!! refines TemplateElement ::=
224                 predicate[1..1] : C:Predicate,
225                 thenCase[1..1] : C:SyntaxTemplate,
226                 elseCase[0..1] : C:SyntaxTemplate ;
227               Switch!! refines TemplateElement ::=
228                 alternative[2..*] : C:SyntaxTemplate ;
229               IncludeSyntaxDef!! refines TemplateElement ::=
230                 conceptDef[0..1] : C:ConceptDef,
231                 differingSyntax[0..1] : C:SyntaxIdentifier
232                 where |P:conceptDef| + |P:differingSyntax| ≥ 1 ;
233             }
234           }
235         }
```

```
236
237         metapackage EdgeAlgebra {
238           metapackage EdgeExpressions {
239             [Edge!!] ;
240
241             [ConstantEdge!!] refines Edge ;
242             EdgeValue!! refines ConstantEdge ::=
243               &value[0..*] ;
244             ConceptEdging!! refines ConstantEdge ::=
245               concept[1..1] : C:Concept ;
246             TypeEdging!! refines ConstantEdge ::=
247               concept[1..1] : C:Concept ;
248             PropertyEdging!! refines ConstantEdge ::=
249               property[1..1] : C:Property ;
250             Self!! refines ConstantEdge ::= ;
251             Equality!! refines ConstantEdge ::= ;
252             Successor!! refines ConstantEdge ::= ;
253             BoundedEdgeVariable!! refines ConstantEdge ::=
254               identifier[1..1] : C:Identifier ;
255
256             [EdgeOperator!!] refines Edge ::>
257               edgeOperand[1..*](List) : C:Edge ;
258             SubPomset!! refines EdgeOperator ::=
259               edgeOperand[1..1],
260               minDepth[1..1] : C:Numerical,
261               maxDepth[0..1] : C:Numerical ;
262
263             [SingleEdgeOperator!!] refines EdgeOperator ::=
264               edgeOperand[1..1] ;
265             First!! refines SingleEdgeOperator ;
266             Closure!! refines SingleEdgeOperator ;
267             EdgeInverse!! refines SingleEdgeOperator ;
268             OrderInverse!! refines SingleEdgeOperator ;
269             OrderDestroy!! refines SingleEdgeOperator ;
270             DuplicateDestroy!! refines SingleEdgeOperator ;
271
272             [MultiEdgeOperator!!] refines EdgeOperator ::=
273               edgeOperand[2..*] ;
274             Navigation!! refines MultiEdgeOperator ;
275             AdditiveUnion!! refines MultiEdgeOperator ;
276             Concatenation!! refines MultiEdgeOperator ;
277             Projection!! refines MultiEdgeOperator ;
278             Difference!! refines MultiEdgeOperator ;
279             Union!! refines MultiEdgeOperator ;
280             Intersection!! refines MultiEdgeOperator ;
281
282             [PredicateToEdge!!] refines Edge ::>
283               predicateOperand[1..*](List) : C:Predicate ;
284
285             [SinglePredicateToEdge!!] refines PredicateToEdge ::=
286               predicateOperand[1..1] ;
287             CastPredicateToEdge!! refines SinglePredicateToEdge ;
```

```
288             PredicateSelection!! refines SinglePredicateToEdge ::=
289               boundedVariable[0..1] : C:Identifier ;
290
291             [NumericalToEdge!!] refines Edge ::>
292               numericalOperand[1..*](List) : C:Numerical ;
293
294             [SingleNumericalToEdge!!] refines NumericalToEdge ::=
295               numericalOperand[1..1] ;
296             CastNumericalToEdge!! refines SingleNumericalToEdge ;
297             NumericalSelection!! refines SingleNumericalToEdge ::=
298               boundedVariable[0..1] : C:Identifier ;
299           }
300
301           metapackage PredicateExpressions {
302             [Predicate!!] ;
303
304             [ConstantPredicate!!] refines Predicate ;
305             PredicateValue!! refines ConstantPredicate ::=
306               value[1..1] : C:Boolean ;
307             IsRoot!! refines ConstantPredicate ::= ;
308
309             [PredicateOperator!!] refines Predicate ::>
310               predicateOperand[1..*](List) : C:Predicate ;
311
312             [QuantifiedEdging!!] refines PredicateOperator ::>
313               boundedVariable[1..1] : C:Identifier,
314               predicateOperand[1..1] ;
315             QuantifiedConceptEdging!! refines QuantifiedEdging ::=
316               excludedConcept[0..*](Set) : C:Concept ;
317             QuantifiedTypeEdging!! refines QuantifiedEdging ::=
318               excludedConcept[0..*](Set) : C:Concept ;
319             QuantifiedPropertyEdging!! refines QuantifiedEdging ::=
320               excludedProperty[0..*](Set) : C:Property ;
321
322             [SinglePredicateOperator!!]
323               refines PredicateOperator ::=
324               predicateOperand[1..1] ;
325             Not!! refines SinglePredicateOperator ;
326
327             [MultiPredicateOperator!!]
328               refines PredicateOperator ::=
329               predicateOperand[2..*] ;
330             And!! refines MultiPredicateOperator ;
331             Or!! refines MultiPredicateOperator ;
332             Xor!! refines MultiPredicateOperator ;
333             Iff!! refines MultiPredicateOperator ;
334             Implies!! refines MultiPredicateOperator ;
335
336             [EdgeToPredicate!!] refines Predicate ::>
337               edgeOperand[1..*](List) : C:Edge ;
338
339             [SingleEdgeToPredicate!!] refines EdgeToPredicate ::=
```

```
340             edgeOperand[1..1] ;
341           CastEdgeToPredicate!! refines SingleEdgeToPredicate ;
342           IsEmpty!! refines SingleEdgeToPredicate ;
343           IsSingleton!! refines SingleEdgeToPredicate ;
344           IsSet!! refines SingleEdgeToPredicate ;
345           IsBag!! refines SingleEdgeToPredicate ;
346           IsList!! refines SingleEdgeToPredicate ;
347           IsToset!! refines SingleEdgeToPredicate ;
348           IsPoset!! refines SingleEdgeToPredicate ;
349
350           [MultiEdgeToPredicate!!] refines EdgeToPredicate ::=
351             edgeOperand[2..*] ;
352           EdgeEqual!! refines MultiEdgeToPredicate ;
353           EdgeNotEqual!! refines MultiEdgeToPredicate ;
354           ConsistsOf!! refines MultiEdgeToPredicate ;
355           NotConsistsOf!! refines MultiEdgeToPredicate ;
356           Subset!! refines MultiEdgeToPredicate ;
357           NotSubset!! refines MultiEdgeToPredicate ;
358           SubsetOrEqual!! refines MultiEdgeToPredicate ;
359           NotSubsetOrEqual!! refines MultiEdgeToPredicate ;
360
361           [NumericalToPredicate!!] refines Predicate ::>
362             numericalOperand[1..*](List) : C:Numerical ;
363
364           [MultiNumericalToPredicate!!]
365             refines NumericalToPredicate ::=
366             numericalOperand[2..*] ;
367           NumericalEqual!! refines MultiNumericalToPredicate ;
368           NumericalNotEqual!! refines MultiNumericalToPredicate ;
369           LessOrEqual!! refines MultiNumericalToPredicate ;
370           LessThan!! refines MultiNumericalToPredicate ;
371           GreaterOrEqual!! refines MultiNumericalToPredicate ;
372           GreaterThan!! refines MultiNumericalToPredicate ;
373         }
374
375         metapackage NumericalExpressions {
376           [Numerical!!] ;
377
378           [ConstantNumerical!!] refines Numerical ;
379           NumericalValue!! refines ConstantNumerical ::=
380             value[1..1] : C:Natural ;
381
382           [NumericalOperator!!] refines Numerical ::>
383             numericalOperand[1..*](List) : C:Numerical ;
384
385           [MultiNumericalOperator!!]
386             refines NumericalOperator ::=
387             numericalOperand[2..*] ;
388           Addition!! refines MultiNumericalOperator ;
389           Subtraction!! refines MultiNumericalOperator ;
390           Multiplication!! refines MultiNumericalOperator ;
391           IntegerDivision!! refines MultiNumericalOperator ;
```

```
392             Modulo!! refines MultiNumericalOperator ;
393             Minimum!! refines MultiNumericalOperator ;
394             Maximum!! refines MultiNumericalOperator ;
395
396             [EdgeToNumerical!!] refines Numerical ::=
397               edgeOperand[1..*](List) : Edge ;
398
399             [SingleEdgeToNumerical!!] refines EdgeToNumerical ::=
400               edgeOperand[1..1] ;
401             CastEdgeToNumerical!! refines SingleEdgeToNumerical ;
402             Cardinality!! refines SingleEdgeToNumerical ;
403             Depth!! refines SingleEdgeToNumerical ;
404           }
405         }
406       }
407     }
408   }
409
410   textual default concrete syntax "M2L/Text" ("Textual_M2L") {
411     syntaxpackage ORG {
412       syntaxpackage Metamodels {
413         syntaxpackage BasicConcepts {
414           Boolean.True: ("⊤" OR "true") ;
415           Boolean.False: ("⊥" OR "false") ;
416           Interval: (P:lower) ".."
417             (|P:upper| = 0 ? "*" : (P:upper)) ;
418
419           Named: (P:name)
420             ("(" || (P:alternativeName) / "," _ || ")") ;
421           Folder: "folder" _ <Named> _ "{"
422             (nl | P:subfolder) (nl | &P:entry) nl "}" ;
423         }
424
425         syntaxpackage M2L {
426           MetamodelFolder: "metamodel" _ <Folder> ;
427           Metamodel: "metamodel" _ <Named>
428             (nl "based" _ "on" _ || &P:basedOn / "," _) "{"
429             (nl | P:abstractSyntax) (nl | P:concreteSyntax)
430             nl "}" ;
431           Metametamodel: "meta-" <Metamodel> ;
432
433           syntaxpackage AbstractSyntax {
434             AbstractSyntax: "abstract" _ "syntax" _ "{"
435               (nl | P:metapackage) nl "}" ;
436             Metapackage: "metapackage" _ <Named> _ "{"
437               (nl | P:conceptDef | _ ";") (nl | P:subpackage)
438               nl "}" ;
439
440             Concept: (P:qualifiedName / ".") ;
441             ConceptDef: ((bool)(P:isAbstract)
442                 ? "[" <Named> (P:conceptType) "]"
443                 : <Named> (P:conceptType))
```

```
444                 (_ "refines" _ || &P:super / "," _) _
445                 ((bool)P:isComplete ? "::=")
446                 ((¬(bool)P:isComplete ? "::>")
447                   || nl | P:propertyDef / ",")
448                 (nl "where" _ | P:additionalConstraint) ;
449
450               AnyConceptDef: "anyconcept" _ <ConceptDef> ;
451               EnumerationConceptDef: "enum" _ <Named> (_ "=" _ "{" _
452                 || P:enumElement / "," _ || _ "}") ;
453               EnumElementConceptDef: <Named> ;
454               ExternalConceptDef: "external" _ <Named> ;
455
456               PropertyDef: ("?" _ | P:assumption | _ "?" _)
457                 (P:keyType) (P:linkType) (P:property)
458                 ("[" || P:multiplicity / "," || "]")
459                 ("(" || P:pomsetRestriction / "," || ")")
460                 (_ ("↔" OR "<->") _ || P:opposite / ",")
461                 (_ ":" _ || P:domain)
462                 (_ ":=" _ || P:inferredValue) ;
463
464               ConceptType.Strong: ;
465               ConceptType.Weak: "!" ;
466               ConceptType.Attribute: "!!" ;
467
468               KeyType.PrimaryLocalkey: "(PK)" ;
469               KeyType.AlternativeLocalkey: "(K)" ;
470
471               LinkType.Reference: "&" ;
472               LinkType.Composition: ;
473               LinkType.Instantiation: "%" ;
474             }
475
476             syntaxpackage ConcreteSyntax {
477               ConcreteSyntax: ((bool)P:default ? "default" _)
478                 "concrete" "syntax" _ <Named> _ "{"
479                 (nl | P:syntaxPackage)
480                 nl "}" ;
481               ConcreteSyntaxPackage: "syntaxpackage" _
482                 (&P:metapackage) _ "{"
483                 (nl | P:concreteSyntaxDef)
484                 (nl | P:subpackage)
485                 nl "}" ;
486               ConcreteSyntaxDef: (&P:conceptDef) ":" _ ;
487
488               syntaxpackage Textual {
489                 TextualSyntax: "textual" _ <ConcreteSyntax> ;
490                 TextualSyntaxDef: <ConcreteSyntaxDef>
491                   (P:mainSyntaxTemplate) _ ";" ;
492
493                 SyntaxTemplate: (P:templateElement / _) ;
494                 ProperTerminal: (P:symbols[QUOTED]) ;
495                 WhitespaceTerminal: (P:whitespace) ;
```

```
496             Whitespace.Space: "_" ;
497             Whitespace.Newline: "nl" ;
498             NonTerminal: "("
499               (P:starting | _ "||" _) (P:prefix | _ "|" _)
500               (P:linkType) (P:edge)
501               ("[" | P:differingSyntax | "]")
502               (_ "/" _ | P:infix)
503               (_ "|" _ | P:suffix) (_ "||" _ | P:ending)
504              ")" ;
505             Option: "(" (P:predicate) _ "?" _ (P:thenCase)
506               (_ ":" _ | P:elseCase) ")" ;
507             Switch: "(" (P:alternative / _ "OR" _) ")" ;
508             IncludeSyntaxDef: "<" (&P:conceptDef)
509               ("[" | P:differingSyntax | "]") ">" ;
510         }
511       }
512     }
513
514     syntaxpackage EdgeAlgebra {
515       syntaxpackage EdgeExpressions {
516         EdgeValue: "{{" _
517           ((P:value["TEXTUAL_UNIQUE"] / "," _)
518            OR (&P:value / "," _)) _ "}}" ;
519         ConceptEdging: ("C:" | P:concept) ;
520         TypeEdging: ("T:" | P:concept) ;
521         PropertyEdging: ("P:" | P:property) ;
522         Self: ("↻" OR "self") ;
523         Equality: ("↪" OR "equality") ;
524         Successor: "*" ;
525         BoundedEdgeVariable: (P:identifier) ;
526
527         SubPomset: (
528             "[" (P:minDepth) ("," | P:maxDepth) "]"
529             (P:edgeOperand[Bracketed]) ;
530           OR
531             "subpomset" "(" (P:edgeOperand)
532             ("," _ | P:minDepth) ("," _ | P:maxDepth) ")"
533           ) ;
534
535         First: ("1" OR "first") _ (P:edgeOperand[Bracketed]) ;
536         Closure: ("^" OR "closure" _)
537           (P:edgeOperand[Bracketed]) ;
538         EdgeInverse: ("⇆" OR "edgeInv" _)
539           (P:edgeOperand[Bracketed]) ;
540         OrderInverse: ("≽" OR "orderInv" _)
541           (P:edgeOperand[Bracketed]) ;
542         OrderDestroy: ("μ" OR "orderDest" _)
543           (P:edgeOperand[Bracketed]) ;
544         DuplicateDestroy: ("ϵ" OR "dupDest" _)
545           (P:edgeOperand[Bracketed]) ;
546
547         Navigation: (P:edgeOperand[Bracketed] / ".") ;
```

```
548             AdditiveUnion: (P:edgeOperand[Bracketed]
549               / _ ("⊎" OR "addUnion") _) ;
550             Concatenation: (P:edgeOperand[Bracketed]
551               / _ ("⊕" OR "concat") _) ;
552             Projection: (P:edgeOperand[Bracketed]
553               / _ ("↓" OR "projectOn") _ ) ;
554             Difference: (P:edgeOperand[Bracketed] / _ "\\" _ ) ;
555             Union: (P:edgeOperand[Bracketed]
556               / _ ("∪" OR "union") _) ;
557             Intersection: (P:edgeOperand[Bracketed]
558               / _ ("∩" OR "intersect") _) ;
559
560             CastPredicateToEdge:
561               "(edge)" (P:predicateOperand[Bracketed]) ;
562             PredicateSelection: ("σ" OR "select")
563               (_ | P:boundedVariable | ":")
564               (_ | P:predicateOperand[Bracketed]) ;
565
566             CastNumericalToEdge:
567               "(edge)" (P:numericalOperand[Bracketed]) ;
568             NumericalSelection: ("σ" OR "select" _)
569               (P:boundedVariable | ":")
570               (P:numericalOperand[Bracketed]) ;
571           }
572
573           syntaxpackage PredicateExpressions {
574             PredicateValue: (P:value) ;
575             Root: "root?" ;
576
577             QuantifiedConceptEdging: ("∀C:" OR "forallC:")
578               (P:boundedVariable)
579               ("\\" "{" || &P:excludedConcept / "," _ || "}")
580               ":" _ (P:predicateOperand[Bracketed]) ;
581             QuantifiedTypeEdging: ("∀T:" OR "forallT:")
582               (P:boundedVariable)
583               ("\\" "{" || &P:excludedConcept / "," _ || "}")
584               ":" _ (P:predicateOperand[Bracketed]) ;
585             QuantifiedPropertyEdging: ("∀P:" OR "forallP:")
586               (P:boundedVariable)
587               ("\\" "{" || &P:excludedProperty / "," _ || "}")
588               ":" _ (P:predicateOperand[Bracketed]) ;
589
590             Not: ("¬" OR "!") (P:predicateOperand[Bracketed]) ;
591
592             And: (P:predicateOperand[Bracketed]
593               / _ ("∧" OR "&") _) ;
594             Or:  (P:predicateOperand[Bracketed]
595               / _ ("∨" OR "v") _) ;
596             Xor: (P:predicateOperand[Bracketed]
597               / _ ("⊕" OR "xor") _) ;
598             Iff: (P:predicateOperand[Bracketed]
599               / _ ("⇔" OR "<=>") _) ;
```

```
600             Implies: (P:predicateOperand[Bracketed]
601               / _ ("⇒" OR "=>") _) ;
602
603             CastEdgeToPredicate:
604               "(bool)" (P:edgeOperand[Bracketed]) ;
605             IsEmpty: "empty?" _ (P:edgeOperand[Bracketed]) ;
606             IsSingleton:
607               "singleton?" _ (P:edgeOperand[Bracketed]) ;
608             IsSet: "set?" _ (P:edgeOperand[Bracketed]) ;
609             IsBag: "bag?" _ (P:edgeOperand[Bracketed]) ;
610             IsList: "list?" _ (P:edgeOperand[Bracketed]) ;
611             IsToset: "toset?" _ (P:edgeOperand[Bracketed]) ;
612             IsPoset: "poset?" _ (P:edgeOperand[Bracketed]) ;
613
614             EdgeEqual: (P:edgeOperand[Bracketed]
615               / _ "=" _) ;
616             EdgeNotEqual: (P:edgeOperand[Bracketed]
617               / _ ("≠" OR "!=") _) ;
618             ConsistsOf: (P:edgeOperand[Bracketed]
619               / _ ("∈" OR "consistsOf") _) ;
620             NotConsistsOf: (P:edgeOperand[Bracketed]
621               / _ ("∉" OR "!consistsOf") _) ;
622             Subset: (P:edgeOperand[Bracketed]
623               / _ ("⊂" OR "subset") _) ;
624             NotSubset: (P:edgeOperand[Bracketed]
625               / _ ("⊄" OR "!subset") _) ;
626             SubsetOrEqual: (P:edgeOperand[Bracketed]
627               / _ ("⊆" OR "subsetEq") _) ;
628             NotSubsetOrEqual: (P:edgeOperand[Bracketed]
629               / _ ("⊈" OR "!subsetEq") _) ;
630
631             NumericalEqual: (P:numericalOperand[Bracketed]
632               / _ "=" _) ;
633             NumericalNotEqual: (P:numericalOperand[Bracketed]
634               / _ ("≠" OR "!=") _) ;
635             LessOrEqual: (P:numericalOperand[Bracketed]
636               / _ ("≤" OR "<=") _) ;
637             LessThan: (P:numericalOperand[Bracketed]
638               / _ "<" _) ;
639             GreaterOrEqual: (P:numericalOperand[Bracketed]
640               / _ ("≥" OR ">=") _) ;
641             GreaterThan: (P:numericalOperand[Bracketed]
642               / _ ">" _) ;
643           }
644
645           syntaxpackage NumericalExpressions {
646             NumericalValue: (P:value) ;
647
648             Addition: (P:numericalOperand[Bracketed]
649               / _ "+" _) ;
650             Subtraction: (P:numericalOperand[Bracketed]
651               / _ "-" _) ;
```

```
652             Multiplication: (P:numericalOperand[Bracketed]
653               / _ ("·" OR "*") _) ;
654             IntegerDivison: (P:numericalOperand[Bracketed]
655               / _ ("÷" OR "div") _) ;
656             Modulo: (P:numericalOperand[Bracketed]
657               / _ ("%" OR "mod") _) ;
658             Minimum: "min" "(" (P:numericalOperand / "," _) ")" ;
659             Maximum: "max" "(" (P:numericalOperand / "," _) ")" ;
660
661             CastEdgeToNumerical:
662               "(number)" (P:edgeOperand[Bracketed]) ;
663             Cardinality: ("size" _ (P:edgeOperand[Bracketed])
664               OR "|" (P:edgeOperand) "|") ;
665             Depth: ("depth" _ (P:edgeOperand[Bracketed]) ;
666               OR "||" (P:edgeOperand) "||") ;
667           }
668         }
669       }
670     }
671   }
672
673   textual concrete syntax Bracketed {
674     syntaxpackage ORG {
675       syntaxpackage Metamodels {
676         syntaxpackage EdgeAlgebra {
677           syntaxpackage EdgeExpressions {
678             Edge: (<["M2L/Text"]> OR "(" <["M2L/Text"]> ")") ;
679
680             Navigation:      "(" <["M2L/Text"]> ")" ;
681             AdditiveUnion:   "(" <["M2L/Text"]> ")" ;
682             Concatenation:   "(" <["M2L/Text"]> ")" ;
683             Projection:      "(" <["M2L/Text"]> ")" ;
684             Difference:      "(" <["M2L/Text"]> ")" ;
685             Union:           "(" <["M2L/Text"]> ")" ;
686             Intersection:    "(" <["M2L/Text"]> ")" ;
687           }
688
689           syntaxpackage PredicateExpressions {
690             Predicate: (<["M2L/Text"]> OR "(" <["M2L/Text"]> ")") ;
691
692             And:      "(" <["M2L/Text"]> ")" ;
693             Or:       "(" <["M2L/Text"]> ")" ;
694             Xor:      "(" <["M2L/Text"]> ")" ;
695             Iff:      "(" <["M2L/Text"]> ")" ;
696             Implies: "(" <["M2L/Text"]> ")" ;
697           }
698
699           syntaxpackage NumericalExpressions {
700             Numerical: (<["M2L/Text"]> OR "(" <["M2L/Text"]> ")") ;
701
702             Addition:        "(" <["M2L/Text"]> ")" ;
703             Subtraction:     "(" <["M2L/Text"]> ")" ;
```

```
704                 Multiplication: "(" <["M2L/Text"]> ")" ;
705                 IntegerDivison: "(" <["M2L/Text"]> ")" ;
706                 Modulo:         "(" <["M2L/Text"]> ")" ;
707             }
708         }
709     }
710   }
711 }
712
```

Appendix B

Metamodel and exemplary models for the Running Example

This appendix shows the complete metamodel as well as the exemplary models for the overall running example. The running example illustrates a simple language for modelling data flow algorithms. The metamodel includes both abstract and textual concrete syntax. In particular two textual concrete syntaxes are defined: The *Structural* one concentrates on the fact that a data flow diagram consists of (sub-)components which are connected by channels. The *Functional* concentrates on the fact that a data flow diagram can be seen as a mathematical function (may be with additional variables).

For a deeper introduction to the running example please refer to Chapter 3, *Running example: Modelling dataflow algorithms*, p. 45.

Contents

B.1. Metamodel of the Running Example

Listing B.1: Metamodel of the Running Example

```
1 metamodel DataflowNetworks
2   based on "Metamodelling Language M2L" {
3   abstract syntax {
4     metapackage ORG {
5       metapackage Metamodels {
6         metapackage Demos {
7           metapackage DataflowNetworks {
8             [DefContainer] refines FolderEntry ::>
```

```
9                &includedLib[0..*](Set) ↔ includedBy : C:Library,
10               componentDef[0..*](Set) : C:Component,
11               &includedContext := ⥁ ⊎ P:includedLib.P:export
12                 ⊎ ⥋P:composite.P:includedContext ;
13
14             Library refines DefContainer ::>
15               &includedBy[0..*](Set)
16                 ↔ includedLib : C:DefContainer,
17               sublibrary[0..*](Set) : C:Library,
18               &export := ⥁ ⊎ P:sublibrary.P:export ;
19
20             [Component!] refines DefContainer ::>
21               signature[1..1] : C:Signature,
22               isPre := (edge)(P:ckey = {{ Identifier(basic),
23                 Identifier(timing), Identifier(pre) }}) ;
24               isAdd := (edge)(P:ckey = {{ Identifier(basic),
25                 Identifier(arithmetrics), Identifier(add) }}) ;
26               isMult := (edge)(P:ckey = {{ Identifier(basic),
27                 Identifier(arithmetrics), Identifier(mult) }}) ;
28
29             Signature! ::>
30               inPort[0..*](List) : C:Port,
31               outPort[0..*](List) : C:Port ;
32
33             Port! refines Named ::>
34               isInPort := (edge)¬empty? ⥋P:inPort,
35               isOutPort := (edge)¬empty? ⥋P:outPort ;
36
37             Block! refines Component ::> ;
38
39             Network! refines Component ::>
40               %subcomponent[0..*](Set) :
41                 P:includedContext.P:componentDef,
42               channel[0..*](Set) : C:Channel ;
43
44             Channel! refines Named ::>
45               &fromPort[1..1] : ⥋P:channel.
46                 (P:signature.P:inPort
47                   ⊎ P:subcomponent.P:signature.P:outPort),
48               &toPort[1..*](Set) : ⥋P:channel.
49                 (P:signature.P:outPort
50                   ⊎ P:subcomponent.P:signature.P:inPort) ;
51               &borderPort := 1 (
52                     (P:fromPort ∩ σ(bool)P:isInPort)
53                   ⊕
54                     (P:toPort ∩ σ(bool)P:isOutPort)
55                 );
56               replaceByBorderPort := (edge)(
57                     |P:fromPort ∩ σ(bool)P:isInPort| = 1
58                   ∨
59                     |P:toPort ∩ σ(bool)P:isOutPort| = 1
60                 ),
```

```
61                isInputForPre := (edge)({{ Boolean(true) }}
62                  ⊂ (P:toPort.⇆P:composite.⇆P:composite.P:isPre)) ;
63            }
64          }
65        }
66      }
67    }
68    textual default concrete syntax Structural {
69      syntaxpackage ORG {
70        syntaxpackage Metamodels {
71          syntaxpackage Demos {
72            syntaxpackage DataflowNetworks {
73              DefContainer: <[Prefix]> _ <Named> _ <[PreBody]>
74                _ "{" <[Body_Structural]> "}" ;
75              Signature: "(" (P:inPort / "," _) _ "->" _
76                (P:outPort / "," _) ")" ;
77              Port: (P:name) ":" "Any" ;
78              Channel: "channel" _ <Named> ":" _
79                (&P:fromPort / "," _) _ "=>" _
80                (&P:toPort / "," _) ";" ;
81            }
82          }
83        }
84      }
85    }
86    textual concrete syntax Functional {
87      syntaxpackage ORG {
88        syntaxpackage Metamodels {
89          syntaxpackage Demos {
90            syntaxpackage DataflowNetworks {
91              DefContainer: <[Prefix]> _ <Named> _ <[PreBody]>
92                _ "{" <[Body_Functional]> "}" ;
93            }
94          }
95        }
96      }
97    }
98    textual concrete syntax Prefix {
99      syntaxpackage ORG {
100       syntaxpackage Metamodels {
101         syntaxpackage Demos {
102           syntaxpackage DataflowNetworks {
103             DefContainer: "container" ;
104             Library: "library" ;
105             Component: "component" ;
106             Block: "block" ;
107             Network: "network" ;
108           }
109         }
110       }
111     }
112   }
```

```
113  textual concrete syntax PreBody {
114    syntaxpackage ORG {
115      syntaxpackage Metamodels {
116        syntaxpackage Demos {
117          syntaxpackage DataflowNetworks {
118            DefContainer: "" ;
119            Component: (_ | P:signature) ;
120          }
121        }
122      }
123    }
124  }
125  textual concrete syntax Body_Structural {
126    syntaxpackage ORG {
127      syntaxpackage Metamodels {
128        syntaxpackage Demos {
129          syntaxpackage DataflowNetworks {
130            DefContainer:
131              (nl "include" _ | &P:includedLib | ";" || nl)
132              (nl nl | P:componentDef[Structural] || nl) ;
133            Library: <DefContainer>
134              (nl nl "sub" | P:sublibrary[Structural] || nl) ;
135            Network: <Component>
136              (nl "subcomponent" _ | %P:subcomponent | ";" || nl)
137              (nl | P:channel[Structural] || nl) ;
138          }
139        }
140      }
141    }
142  }
143  textual concrete syntax Body_Functional {
144    syntaxpackage ORG {
145      syntaxpackage Metamodels {
146        syntaxpackage Demos {
147          syntaxpackage DataflowNetworks {
148            DefContainer:
149              (nl "include" _ | &P:includedLib | ";" || nl)
150              (nl nl | P:componentDef[Functional] || nl) ;
151            Network: <Component> (nl | (
152                (P:subcomponent
153                  ∩ σ¬|P:signature.P:outPort| = 1)
154              ∪
155                (P:subcomponent
156                  ∩ (P:signature.P:outPort.⇆P:toPort
157                   .P:fromPort.⇆P:composite.⇆P:composite))
158              ∪
159                ((P:signature.μP:outPort)
160                  ∩ (P:signature.P:inPort.⇆P:fromPort.P:toPort))
161              ∪
162                (P:signature.μP:outPort
163                  ∩ σ|(⇆P:toPort.P:toPort) ∩ σ(bool)P:isOutPort|
164                  ≥ 2)
```

```
165               )[Functional_Assignment] || nl) ;
166             Library : <DefContainer>
167               (nl nl "sub" | P:sublibrary[Functional] || nl) ;
168           }
169         }
170       }
171     }
172   }
173   textual concrete syntax Functional_Assignment {
174     syntaxpackage ORG {
175       syntaxpackage Metamodels {
176         syntaxpackage Demos {
177           syntaxpackage DataflowNetworks {
178             Port: (&↻) _ ":=" _
179               (⇆P:toPort[Functional_ChannelOrBorderPortName])
180               _ ";" ;
181             Component: (|P:signature.P:outPort| <= 1
182                 ? ((P:signature.P:outPort.⇆P:fromPort)
183                   [Functional_ChannelOrBorderPortName] || _ ":=" _)
184                 : ("(" || (P:signature.P:outPort.⇆P:fromPort)
185                   [Functional_ChannelOrBorderPortName] / "," _
186                   || ")" _ ":=" _)
187               ) <[Functional_Argument]> _ ";" ;
188           }
189         }
190       }
191     }
192   }
193   textual concrete syntax Functional_Argument {
194     syntaxpackage ORG {
195       syntaxpackage Metamodels {
196         syntaxpackage Demos {
197           syntaxpackage DataflowNetworks {
198             Channel : ((
199                 (bool)P:isInputForPre
200               ∨ (
201                   ¬|P:fromPort.⇆P:composite.P:outPort| = 1
202                 ∧
203                   (bool)(P:fromPort.P:isOutPort)
204                 )
205               )
206                 ? <[Functional_ChannelOrBorderPortName]>
207                 : ((bool)(P:fromPort.P:isInPort)
208                     ? (&P:fromPort)
209                     : (P:fromPort.⇆P:composite.⇆P:composite)
210                   )
211               ) ;
212             Component : (&P:template) "("
213               ((P:signature.P:inPort.⇆P:toPort)
214               [Functional_Argument] / "," _) ")" ;
215           }
216         }
```

```
      }
    }
  }
  textual concrete syntax Functional_ChannelOrBorderPortName {
    syntaxpackage ORG {
      syntaxpackage Metamodels {
        syntaxpackage Demos {
          syntaxpackage DataflowNetworks {
            Channel : ((bool)(P:replaceByBorderPort)
                ? (&P:borderPort)
                : (&↻)
              ) ;
          }
        }
      }
    }
  }
}
```

B.2. Exemplary Model: Basic Library

B.2.1. Functional/Structural Syntax

Listing B.2: Exemplary Model: Basic Library (functional/structural syntax)

```
library basic {
  sublibrary arithmetrics {
    block add (x:Any, y:Any -> result:Any) {}
    block mult (x:Any, y:Any -> result:Any) {}
  }
  sublibrary timing {
    block dT ( -> out:Any) {}
    block pre (in:Any -> out:Any) {}
  }
}
```

B.2.2. Canonical Syntax

Listing B.3: Exemplary Model: Basic Library (canonical syntax)

```
Library basic {
  includedBy: integrator;
  sublibrary: Library arithmetrics {
    componentDef: Block add {
      signature: Signature {
        inPort: Port x {};
        inPort: Port y {};
        outPort: Port result {};
      };
```

```
    };
    componentDef: Block mult {
      signature: Signature {
        inPort: Port x {};
        inPort: Port y {};
        outPort: Port result {};
      };
    };
  };
  sublibrary: Library timing {
    includedBy: "Textual Syntax Demonstration"."pre loop";
    componentDef: Block dT {
      signature: Signature {
        outPort: Port out {};
      };
    };
    componentDef: Block pre {
      signature: Signature {
        inPort: Port in {};
        outPort: Port out {};
      };
    };
  };
}
```

B.3. Exemplary Model: Integrator Network

B.3.1. Functional Syntax

Listing B.4: Exemplary Model: Integrator Network (functional syntax)

```
network integrator (x:Any -> y:Any) {
  include basic;

  y := add(mult(x, dT()), pre(y)) ;
}
```

B.3.2. Structural Syntax

Listing B.5: Exemplary Model: Integrator Network (structural syntax)

```
network integrator (x:Any -> y:Any) {
  include basic;

  subcomponent dT[dT0];
  subcomponent pre[pre0];
  subcomponent mult[mult0];
  subcomponent add[add0];

```

```
  channel c1: x => mult0.x;
  channel c2: dT0.out => mult0.y;
  channel c3: mult0.result => add0.x;
  channel c4: pre0.out => add0.y;
  channel c5: add0.result => pre0.in, y;
}
```

B.3.3. Canonical Syntax

Listing B.6: Exemplary Model: Integrator Network (canonical syntax)

```
Network integrator {
  includedLib: basic;
  signature: Signature {
    inPort: Port x {};
    outPort: Port y {};
  };
  subcomponent: Block add0 {
    ikey: add0;
    template: add;
    signature: Signature {
      template: add.signature;
      inPort: Port x {
        template: add.x;
      };
      inPort: Port y {
        template: add.y;
      };
      outPort: Port result {
        template: add.result;
      };
    };
  };
  subcomponent: Block dT0 {
    ikey: dT0;
    template: dT;
    signature: Signature {
      template: dT.signature;
      outPort: Port out {
        template: dT.out;
      };
    };
  };
  subcomponent: Block mult0 {
    ikey: mult0;
    template: mult;
    signature: Signature {
      template: mult.signature;
      inPort: Port x {
        template: mult.x;
      };
```

```
41        inPort: Port y {
42          template: mult.y;
43        };
44        outPort: Port result {
45          template: mult.result;
46        };
47      };
48    };
49    subcomponent: Block pre0 {
50      ikey: pre0;
51      template: pre;
52      signature: Signature {
53        template: pre.signature;
54        inPort: Port in {
55          template: pre.in;
56        };
57        outPort: Port out {
58          template: pre.out;
59        };
60      };
61    };
62    channel: Channel c1 {
63      fromPort: x;
64      toPort: mult0.x;
65    };
66    channel: Channel c2 {
67      fromPort: dT0.out;
68      toPort: mult0.y;
69    };
70    channel: Channel c3 {
71      fromPort: mult0.result;
72      toPort: add0.x;
73    };
74    channel: Channel c4 {
75      fromPort: pre0.out;
76      toPort: add0.y;
77    };
78    channel: Channel c5 {
79      fromPort: add0.result;
80      toPort: pre0.in;
81      toPort: y;
82    };
83  }
```

B.4. Exemplary Model: Demonstration Vehicle

B.4.1. Functional Syntax

Listing B.7: Exemplary Model: Demonstration Vehicle (functional syntax)

```
network "Demonstration Vehicle" ( -> ) {

  network "Adaptive Cruise Control" (mode:Any, "desired speed":Any,
    "current speed":Any, distance:Any -> status:Any,
    "target speed":Any) {
    ...
  }
  block Display (status:Any -> ) {}
  block Engine ("target speed":Any -> ) {}
  block "Radar Sensor" ( -> distance:Any) {}
  block "Rotation Sensor" ( -> "current speed":Any) {}
  block "User Interface" ( -> mode:Any, "desired speed":Any) {}

  (mode, "desired speed") := "User Interface"() ;
  (status, "target speed") := "Adaptive Cruise Control"(
    mode, "desired speed", "Rotation Sensor"(), "Radar Sensor"()) ;
  Display(status) ;
  Engine("target speed") ;
}
```

B.4.2. Structural Syntax

Listing B.8: Exemplary Model: Demonstration Vehicle (structural syntax)

```
network "Demonstration Vehicle" ( -> ) {

  network "Adaptive Cruise Control" (mode:Any, "desired speed":Any,
    "current speed":Any, distance:Any -> status:Any,
    "target speed":Any) {
    ...
  }
  block Display (status:Any -> ) {}
  block Engine ("target speed":Any -> ) {}
  block "Radar Sensor" ( -> distance:Any) {}
  block "Rotation Sensor" ( -> "current speed":Any) {}
  block "User Interface" ( -> mode:Any, "desired speed":Any) {}

  subcomponent "Adaptive Cruise Control"
    ["Adaptive Cruise Control0"];
  subcomponent Display[Display0];
  subcomponent Engine[Engine0];
  subcomponent "Radar Sensor"["Radar Sensor0"];
  subcomponent "Rotation Sensor"["Rotation Sensor0"];
  subcomponent "User Interface"["User Interface0"];

  channel "current speed": "Rotation Sensor0"."current speed"
    => "Adaptive Cruise Control0"."current speed";
  channel "desired speed": "User Interface0"."desired speed"
    => "Adaptive Cruise Control0"."desired speed";
  channel distance: "Radar Sensor0".distance
    => "Adaptive Cruise Control0".distance;
```

```
  channel mode: "User Interface0".mode
    => "Adaptive Cruise Control0".mode;
  channel status: "Adaptive Cruise Control0".status
    => Display0.status;
  channel "target speed": "Adaptive Cruise Control0"."target speed"
    => Engine0."target speed";
}
```

B.4.3. Canonical Syntax

Listing B.9: Exemplary Model: Demonstration Vehicle (canonical syntax)

```
Network "Demonstration Vehicle" {
  componentDef: Network "Adaptive Cruise Control" {
    signature: Signature {
      inPort: Port mode {};
      inPort: Port "desired speed" {};
      inPort: Port "current speed" {};
      inPort: Port distance {};
      outPort: Port status {};
      outPort: Port "target speed" {};
    };
  };
  componentDef: Block Display {
    signature: Signature {
      inPort: Port status {};
    };
  };
  componentDef: Block Engine {
    signature: Signature {
      inPort: Port "target speed" {};
    };
  };
  componentDef: Block "Radar Sensor" {
    signature: Signature {
      outPort: Port distance {};
    };
  };
  componentDef: Block "Rotation Sensor" {
    signature: Signature {
      outPort: Port "current speed" {};
    };
  };
  componentDef: Block "User Interface" {
    signature: Signature {
      outPort: Port mode {};
      outPort: Port "desired speed" {};
    };
  };
  signature: Signature {};
  subcomponent: Network "Adaptive Cruise Control0" {
```

```
40      ikey: "Adaptive Cruise Control0";
41      template: "Adaptive Cruise Control";
42      signature: Signature {
43        template: "Adaptive Cruise Control".signature;
44        inPort: Port mode {
45          template: "Adaptive Cruise Control".mode;
46        };
47        inPort: Port "desired speed" {
48          template: "Adaptive Cruise Control"."desired speed";
49        };
50        inPort: Port "current speed" {
51          template: "Adaptive Cruise Control"."current speed";
52        };
53        inPort: Port distance {
54          template: "Adaptive Cruise Control".distance;
55        };
56        outPort: Port status {
57          template: "Adaptive Cruise Control".status;
58        };
59        outPort: Port "target speed" {
60          template: "Adaptive Cruise Control"."target speed";
61        };
62      };
63    };
64    subcomponent: Block Display0 {
65      ikey: Display0;
66      template: Display;
67      signature: Signature {
68        template: Display.signature;
69        inPort: Port status {
70          template: Display.status;
71        };
72      };
73    };
74    subcomponent: Block Engine0 {
75      ikey: Engine0;
76      template: Engine;
77      signature: Signature {
78        template: Engine.signature;
79        inPort: Port "target speed" {
80          template: Engine."target speed";
81        };
82      };
83    };
84    subcomponent: Block "Radar Sensor0" {
85      ikey: "Radar Sensor0";
86      template: "Radar Sensor";
87      signature: Signature {
88        template: "Radar Sensor".signature;
89        outPort: Port distance {
90          template: "Radar Sensor".distance;
91        };
```

```
92        };
93      };
94      subcomponent: Block "Rotation Sensor0" {
95        ikey: "Rotation Sensor0";
96        template: "Rotation Sensor";
97        signature: Signature {
98          template: "Rotation Sensor".signature;
99          outPort: Port "current speed" {
100           template: "Rotation Sensor"."current speed";
101         };
102       };
103     };
104     subcomponent: Block "User Interface0" {
105       ikey: "User Interface0";
106       template: "User Interface";
107       signature: Signature {
108         template: "User Interface".signature;
109         outPort: Port mode {
110           template: "User Interface".mode;
111         };
112         outPort: Port "desired speed" {
113           template: "User Interface"."desired speed";
114         };
115       };
116     };
117     channel: Channel "current speed" {
118       fromPort: "Rotation Sensor0"."current speed";
119       toPort: "Adaptive Cruise Control0"."current speed";
120     };
121     channel: Channel "desired speed" {
122       fromPort: "User Interface0"."desired speed";
123       toPort: "Adaptive Cruise Control0"."desired speed";
124     };
125     channel: Channel distance {
126       fromPort: "Radar Sensor0".distance;
127       toPort: "Adaptive Cruise Control0".distance;
128     };
129     channel: Channel mode {
130       fromPort: "User Interface0".mode;
131       toPort: "Adaptive Cruise Control0".mode;
132     };
133     channel: Channel status {
134       fromPort: "Adaptive Cruise Control0".status;
135       toPort: Display0.status;
136     };
137     channel: Channel "target speed" {
138       fromPort: "Adaptive Cruise Control0"."target speed";
139       toPort: Engine0."target speed";
140     };
141   }
```

B.5. Exemplary Model: Textual Syntax Demonstration

B.5.1. Functional Syntax

Listing B.10: Exemplary Model: Textual Syntax Demonstration (functional syntax)

```
library "Textual Syntax Demonstration" {
  network "alternative out" (in1:Any -> out1:Any, out2:Any) {
    block A (in1:Any -> out1:Any) {}

    out2 := A(in1) ;
    out1 := in1 ;
  }

  network duplicator (in1:Any -> out1:Any, out2:Any) {
    out1 := in1 ;
    out2 := in1 ;
  }

  network identity (in1:Any -> out1:Any) {
    out1 := in1 ;
  }

  network mixed ( -> out1:Any, out2:Any) {
    block A ( -> out1:Any, out2:Any) {}
    block B (in1:Any -> out1:Any) {}

    (out1, ch2) := A() ;
    out2 := B(ch2) ;
  }

  network "pre loop" ( -> out1:Any) {
    include basic.timing;

    out1 := pre(out1) ;
  }

  network sink (in1:Any -> ) {
    block A (in1:Any -> ) {}

    A(in1) ;
  }

  network source1 ( -> out1:Any, out2:Any) {
    block A ( -> out1:Any) {}

    ch1 := A() ;
    out1 := ch1 ;
    out2 := ch1 ;
  }

  network source2 ( -> out1:Any, out2:Any) {
```

```
47      block A ( -> out1:Any, out2:Any) {}
48
49      (out1, out2) := A() ;
50    }
51  }
```

B.5.2. Structural Syntax

Listing B.11: Exemplary Model: Textual Syntax Demonstration (structural syntax)

```
1  library "Textual Syntax Demonstration" {
2
3    network "alternative out" (in1:Any -> out1:Any, out2:Any) {
4      block A (in1:Any -> out1:Any) {}
5      subcomponent A[A0];
6
7      channel ch1: in1 => A0.in1, out1;
8      channel ch2: A0.out1 => out2;
9    }
10
11   network duplicator (in1:Any -> out1:Any, out2:Any) {
12     channel ch1: in1 => out1, out2;
13   }
14
15   network identity (in1:Any -> out1:Any) {
16     channel ch1: in1 => out1;
17   }
18
19   network mixed ( -> out1:Any, out2:Any) {
20     block A ( -> out1:Any, out2:Any) {}
21     block B (in1:Any -> out1:Any) {}
22     subcomponent A[A0];
23     subcomponent B[B0];
24
25     channel ch1: A0.out1 => out1;
26     channel ch2: A0.out2 => B0.in1;
27     channel ch3: B0.out1 => out2;
28   }
29
30   network "pre loop" ( -> out1:Any) {
31     include basic.timing;
32     subcomponent pre[pre0];
33
34     channel ch1: pre0.out => pre0.in, out1;
35   }
36
37   network sink (in1:Any -> ) {
38     block A (in1:Any -> ) {}
39     subcomponent A[A0];
40
41     channel ch1: in1 => A0.in1;
```

```
42    }
43
44    network source1 ( -> out1:Any, out2:Any) {
45      block A ( -> out1:Any) {}
46      subcomponent A[A0];
47
48      channel ch1: A0.out1 => out1, out2;
49    }
50
51    network source2 ( -> out1:Any, out2:Any) {
52      block A ( -> out1:Any, out2:Any) {}
53      subcomponent A[A0];
54
55      channel ch1: A0.out1 => out1;
56      channel ch2: A0.out2 => out2;
57    }
58  }
```

B.5.3. Canonical Syntax

Listing B.12: Exemplary Model: Textual Syntax Demonstration (canonical syntax)

```
1  Library "Textual Syntax Demonstration" {
2    componentDef: Network "alternative out" {
3      componentDef: Block A {
4        signature: Signature {
5          inPort: Port in1 {};
6          outPort: Port out1 {};
7        };
8      };
9      signature: Signature {
10       inPort: Port in1 {};
11       outPort: Port out1 {};
12       outPort: Port out2 {};
13     };
14     subcomponent: Block A0 {
15       ikey: A0;
16       template: A;
17       signature: Signature {
18         template: A.signature;
19         inPort: Port in1 {
20           template: A.in1;
21         };
22         outPort: Port out1 {
23           template: A.out1;
24         };
25       };
26     };
27     channel: Channel ch1 {
28       fromPort: in1;
29       toPort: A0.in1;
```

```
30        toPort: out1;
31      };
32      channel: Channel ch2 {
33        fromPort: A0.out1;
34        toPort: out2;
35      };
36    };
37    componentDef: Network duplicator {
38      signature: Signature {
39        inPort: Port in1 {};
40        outPort: Port out1 {};
41        outPort: Port out2 {};
42      };
43      channel: Channel ch1 {
44        fromPort: in1;
45        toPort: out1;
46        toPort: out2;
47      };
48    };
49    componentDef: Network identity {
50      signature: Signature {
51        inPort: Port in1 {};
52        outPort: Port out1 {};
53      };
54      channel: Channel ch1 {
55        fromPort: in1;
56        toPort: out1;
57      };
58    };
59    componentDef: Network mixed {
60      componentDef: Block A {
61        signature: Signature {
62          outPort: Port out1 {};
63          outPort: Port out2 {};
64        };
65      };
66      componentDef: Block B {
67        signature: Signature {
68          inPort: Port in1 {};
69          outPort: Port out1 {};
70        };
71      };
72      signature: Signature {
73        outPort: Port out1 {};
74        outPort: Port out2 {};
75      };
76      subcomponent: Block A0 {
77        ikey: A0;
78        template: A;
79        signature: Signature {
80          template: A.signature;
81          outPort: Port out1 {
```

```
82            template: A.out1;
83          };
84          outPort: Port out2 {
85            template: A.out2;
86          };
87        };
88      };
89      subcomponent: Block B0 {
90        ikey: B0;
91        template: B;
92        signature: Signature {
93          template: B.signature;
94          inPort: Port in1 {
95            template: B.in1;
96          };
97          outPort: Port out1 {
98            template: B.out1;
99          };
100       };
101     };
102     channel: Channel ch1 {
103       fromPort: A0.out1;
104       toPort: out1;
105     };
106     channel: Channel ch2 {
107       fromPort: A0.out2;
108       toPort: B0.in1;
109     };
110     channel: Channel ch3 {
111       fromPort: B0.out1;
112       toPort: out2;
113     };
114   };
115   componentDef: Network "pre loop" {
116     includedLib: basic.timing;
117     signature: Signature {
118       outPort: Port out1 {};
119     };
120     subcomponent: Block pre0 {
121       ikey: pre0;
122       template: pre;
123       signature: Signature {
124         template: pre.signature;
125         inPort: Port in {
126           template: basic.timing.pre.in;
127         };
128         outPort: Port out {
129           template: basic.timing.pre.out;
130         };
131       };
132     };
133     channel: Channel ch1 {
```

```
134       fromPort: pre0.out;
135       toPort: pre0.in;
136       toPort: out1;
137     };
138   };
139   componentDef: Network sink {
140     componentDef: Block A {
141       signature: Signature {
142         inPort: Port in1 {};
143       };
144     };
145     signature: Signature {
146       inPort: Port in1 {};
147     };
148     subcomponent: Block A0 {
149       ikey: A0;
150       template: A;
151       signature: Signature {
152         template: A.signature;
153         inPort: Port in1 {
154           template: A.in1;
155         };
156       };
157     };
158     channel: Channel ch1 {
159       fromPort: in1;
160       toPort: A0.in1;
161     };
162   };
163   componentDef: Network source1 {
164     componentDef: Block A {
165       signature: Signature {
166         outPort: Port out1 {};
167       };
168     };
169     signature: Signature {
170       outPort: Port out1 {};
171       outPort: Port out2 {};
172     };
173     subcomponent: Block A0 {
174       ikey: A0;
175       template: A;
176       signature: Signature {
177         template: A.signature;
178         outPort: Port out1 {
179           template: A.out1;
180         };
181       };
182     };
183     channel: Channel ch1 {
184       fromPort: A0.out1;
185       toPort: out1;
```

```
186         toPort: out2;
187       };
188     };
189     componentDef: Network source2 {
190       componentDef: Block A {
191         signature: Signature {
192           outPort: Port out1 {};
193           outPort: Port out2 {};
194         };
195       };
196       signature: Signature {
197         outPort: Port out1 {};
198         outPort: Port out2 {};
199       };
200       subcomponent: Block A0 {
201         ikey: A0;
202         template: A;
203         signature: Signature {
204           template: A.signature;
205           outPort: Port out1 {
206             template: A.out1;
207           };
208           outPort: Port out2 {
209             template: A.out2;
210           };
211         };
212       };
213       channel: Channel ch1 {
214         fromPort: A0.out1;
215         toPort: out1;
216       };
217       channel: Channel ch2 {
218         fromPort: A0.out2;
219         toPort: out2;
220       };
221     };
222 }
```