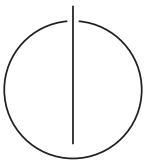
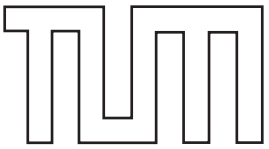


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable D1.2.B-1

Specification Techniques

Author: Alarico Campetelli
María Victoria Cengarle
Irina Gaponova
Alexander Harhurin
Daniel Ratiu
Judith Thyssen
Version: 0.9
Date: April 27, 2010
Status: Draft

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

About the Document

Model-based development assumes the pervasive use of models along all software development phases. Models are usually built using modelling tools. Behind each tool there is a modelling language that comprises one or more specification techniques. A specification technique represents the essential ideas (archetypal view) behind a modelling (sub-)language. For example, the well-known Rational Rhapsody tool is based on the Unified Modeling Language (UML, see [OMG09]) which contains a specification techniques for each one of its sub-languages (for instance, the UML sub-language of State Machines is based on the Statecharts specification technique), and Statecharts occur in different modelling languages and tools in slightly different forms. Thus, concrete modelling languages instantiate one or more specification techniques.

Currently, there are quite a number of specification techniques available that have different focuses and fit better for certain modelling purposes and/or process phases. Examples range from techniques that aim to facilitate the communication among engineers (e. g., Use Cases) up to techniques semantically rich in order to allow generation of code or formal verification (e. g., Statecharts). Furthermore, different specification techniques might address different aspects of the system such as its behaviour, structure, or interaction with the environment. Distinguishing among the strong points of each specification technique is difficult due to the big variety of their dialects and various maturity degrees of tool support.

In general, for a given task, an engineer can choose between many specification techniques. The choice of a technique influences the measure in which the models built using that technique can be refined later in the development process, which of their properties can be analysed, or how they can be integrated with other models using the same or another technique. Conversely, the use of an inadequate specification technique can hinder the evolution of the model in later process phases or limit the spectrum of analyses that can be performed. That is, the choice of a specification technique enables what can be subsequently done with the models.

In this document we present a set of criteria for classifying well-known and widespread specification techniques. While in practice the choice of a specification technique is normally driven by the availability of tools in an organisation, we aim at building a catalogue that can be used off-the-shelf by engineers in order to make informed decisions about the adequacy of a certain specification technique for different modelling tasks. For each specification technique we also provide information about its mostly used modelling dialects.

Contents

1 Introduction

Model-based development advocates the pervasive use of models throughout the entire software development process. Early defined models capture requirements on the system, and are subsequently transformed and enriched until an implementation is completed. Models are therefore at different levels of detail, of formalisation, and are used for different purposes by different stakeholders in different process phases.

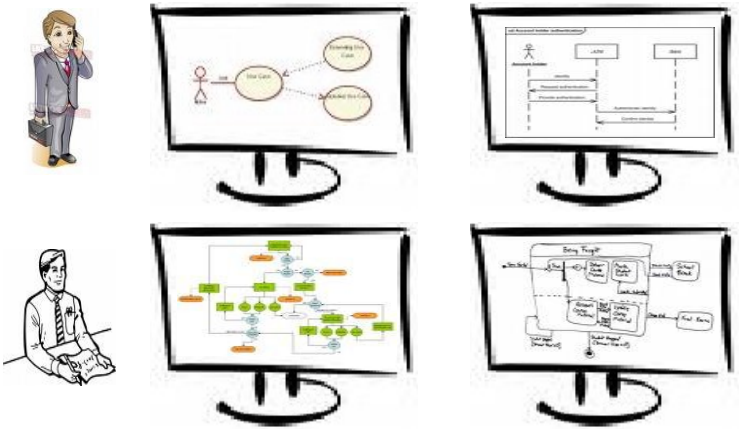


Figure 1: Motivation overview

Tools, modelling languages, and specification techniques. Typically, engineers from various domains (e. g., from automotive or avionics) use a handful of tools (e. g., Simulink, Rhapsody) to model their intents (cf. Figure 1). Each tool puts a modelling language at engineers disposal as a means for describing their systems. By just using a tool, it is often not clear that (a certain part of) its modelling language is similar to (another part) of the modelling language of a further tool. Moreover, different modelling languages might be presented in a similar manner in different tools. Unsurprisingly, this situation leads to confusions.

Figure 2 illustrates the relation between tools, modelling languages, and specification techniques. Behind each tool there is a modelling language that the stakeholders use to build the models. Usually, modelling languages contain several logically distinct sub-languages. For example, the Unified Modeling Language (UML [OMG09]) is a collection of basic languages like State Machine Diagrams, Component Diagrams, etc. Each (basic) modelling language is based on a specification technique that is a set of archetypal modelling constructs. The relation between a tool and a specification technique is many-to-many: a tool can use a set of specification techniques, and a specification technique can be the basis of the modelling language used by different tools. The specification techniques can in turn be classified into specification paradigms: a paradigm comprises the commonalities of different specification techniques.

Note: There are some specification techniques for which there is only one concrete modelling language. For instance, and to the best of our knowledge, LSCs are only implemented in the Play-Engine; see [HM03]. Another example are the Use Cases from UML, for which there is

only one modelling language, namely UML, implemented in different tools like, e. g., Argo-UML and Rhapsody.

In practice, a specification technique appears in different forms or dialects in different languages. However, many dialects of modelling languages can be understood and classified in terms of the specification techniques that they are based on. Therefore, instead of studying the multitude of modelling languages and dialects, we focus in this document on a smaller set of basic specification techniques.

Example: 'Automata' or 'State Transition Systems' represent a specification paradigm. This paradigm comprises many specification techniques like I/O Automata, Timed Automata, Hybrid Automata, Probabilistic Automata, Statecharts, and so on (for details see Section 3.3). These specification techniques, possibly in different dialects, are present in different modelling languages (e. g., UML contains a dialect of statecharts, the language behind the tool Statemate contains a slightly different dialect of statecharts). Modelling languages are in turn parts of different tools (e. g., Telelogic Rhapsody and ArgoUML are UML-based tools; Statemate, Matlab Simulink or Esterel Studio are tools that implement their own dialect of statecharts).

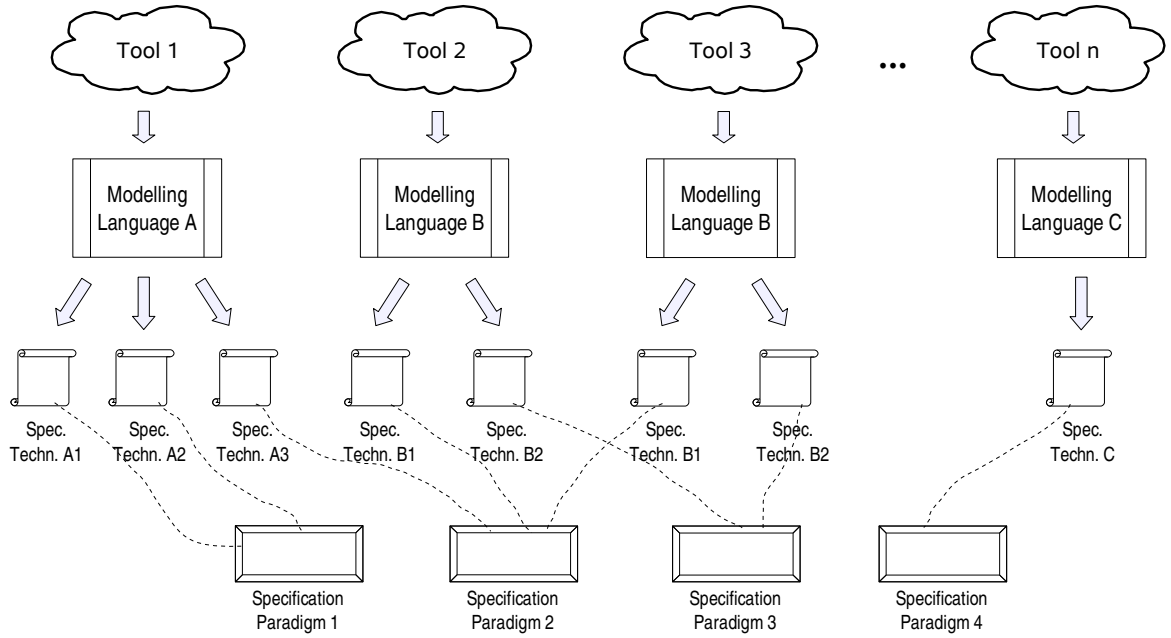


Figure 2: Tools, modelling languages, and specification techniques

Syntax, semantics, and pragmatics. As already mentioned, there are many dialects of a specification technique. Almost each tool provides its specific dialect, and to end users different dialects might appear to be highly different. However, behind these dialects there are specification concepts that occur recurrently in modelling languages, and that constitute the specification techniques.

On the one hand, specification techniques appear in various forms and dialects as provided by specialised tools – i. e., different tools put at disposal different notations (concrete syntaxes) for

modelling. On the other hand, very similar notations are used to describe different aspects of a system. While some differences concern only the presentation (syntax) and have no impact on the meaning (semantics), some similarities likewise concern the syntax but are accompanied by divergences in the semantics. For the sake of comprehensibility, these situations need be made explicit.

The differences between several instances of a specification technique occur along following dimensions:

- *concrete syntax* means the exact notation used to represent models. Different modelling tools support different notations and styles – e. g., some tools favour a graphical syntax while others textual or tabular ones.
- *abstract syntax* describes the modelling language, namely the set of modelling constructs and the basic ways to combine them. It describes the modelling concepts directly supported by the language.
- *syntactic sugar:* some languages provide different syntactic ways to describe the same concept and aim at clarity, easiness, or just alternatives that some users might prefer. These ways can be removed from the overall syntax without any loss of expressivity.
- *semantics* refers to the meaning of the model written in a language. The differences in semantics between two languages become dangerous when not accompanied by syntactic dissimilarity. In these cases, the models look alike but have different meanings. This, in turn, endangers transformation and integration of models containing those similarly looking constructs – the users of the tools for instance might (misleadingly) decide that two models are compatible when, in fact, they are not. Semantics can be given either informally in natural language (imprecise) or formally as mathematical formula (rigorous and precise).

Specification techniques. In practice, the choice of a specification technique is normally driven by the availability of tools in an organisation and not necessarily by a founded and informed decision. Engineers think in terms of the concrete dialect of a specification technique that is available in the tools they have at hand. In the present document, however, we focus on specification techniques and not on modelling languages. Thereby, we abstract in our presentation from syntactic issues (i. e., how do the models exactly occur in tools [concrete syntax, syntactic sugar], or which constructs exactly does a modelling language provide [abstract syntax]).

There is a wide spectrum for the use of specification techniques. Different specification techniques are used for different purposes at different stages of the development process. The techniques used in very early phases (e. g., use cases) must be understood by usually strongly different stakeholders and thereby focus on enabling inter-human communication. Other specification techniques are targeted at a first formalisation of the system, i. e., they support early formalisation, and need be at least partly verifiable and verified, e. g., by checking constraints fulfillment. Finally, some specification techniques are used for generating code and should have a formally defined, i. e., mathematically precise, semantics.

In this document, we aim at presenting a set of criteria for classifying well-known and widespread specification techniques. Our goal is to build up a catalogue that can be used by engineers in

order to make informed decisions about the adequacy of a certain specification technique for different modelling tasks.

Note: We are aware that, due to the heterogeneity of the specification techniques, not all of these can be easily compared pairwise.

Outline. The rest of this document is structured as follows: in Section 2 we present a set of criteria used for the classification of different specification techniques; in Section 3 we present our classification of the techniques according to these criteria; in Section 4 we present a summary and our conclusions.

2 Criteria for the Classification of Specification Techniques

Each of the following sub-sections presents a criterion relevant for a classification of specification techniques. In Section 2.1 we present the views of a system covered by specifications, in Section 2.2 we focus on criteria necessary to distinguish between different specification techniques that address the behaviour; in Section 2.3 we present the software development phase typical for different specification techniques, in Section 2.4 we present the abstraction layers where a technique is mostly commonly found, in Section 2.5 we discuss the compositionality principle, in Section 2.6 we discuss tool support, in Section 2.7 we discuss typical analysis techniques, and, finally, in Section 2.8 we present typical notations used for different specification techniques.

2.1 Covered System Views

In the software development praxis, in order to reduce its complexity, the system is modelled by an abstraction. In order to make the development even easier, several conjoint modelling abstractions, also called views, can be used. They facilitate the modelling since a model capturing every single system feature is unhandy and error prone. The views structure the description of a complex system. Each view focuses on one specific aspect. According to [Bro95], views concern with following system aspects:

- data (also called information model)
- structure (also called organisational or architectural model)
- behaviour (interface model, behaviour history model)
- process (dynamic view)
- state (state transition model).

In the literature, also other sets of conjoint modelling abstractions can be found. For instance [HLN+90] proposes the following set: *(1)* structural view, *(2)* behavioural view, and *(3)* functional view. We believe that the functional view of this classification supplements the above one:

- functional view

Typically, each specification technique addresses primarily one view over the system. The view addressed by a specification technique is many times only implicit and hidden behind the concepts that the specification technique offers. Using a specification technique to describe a system view which it does not address or for which it is not intended, leads to encodings and ambiguities.

Data view. The data view captures structures for representing the data of the systems, as for instance sorts (often also called types or modes) and their range. The typical specification technique for this view is the E/R-diagram. The data view usually follows the object-oriented approach and can be used to organise data relevant both for the static and the dynamic aspects of a system.

Structural view. The structural view shows the architecture, i. e., describes the static structure of the system. This view consists of nodes representing components, and arcs between nodes representing communication lines (channels). Thus, the structure is shown as a network of communicating components. During the development process, there may be several structural views, for instance indicating the logical structure and the physical structure of a system; see [Hil99]. Structural views can be represented by, e. g., component diagrams.

Behaviour view. The behaviour view describes the behaviour of the system by showing the collaboration or interaction between system parts (or sub-systems) as well as the relation between input and output interfaces of the system. The behaviour view specifies changes in the system state by describing conditions and events that fire a change; it addresses concerns like causality, concurrency and synchronisation. Typical specification techniques used for the description of a behaviour view are data flow diagrams.

Process view. The process view describes the run time decomposition of a system into processes with emphasis on concurrency and synchronisation aspects. A process is a sequence of consecutive actions that have some causal relationship. There are many variation of mathematical models for processes, one of the typical specification technique being Petri Nets.

State view. The state view specifies state transition model, i. e., the white-box behaviour of a sub-system which is presented as a black box in the behaviour view. Typical modelling techniques for this purpose are state machines.

Functional view. The functional view shows the functional decomposition of a system. That is, it presents a hierarchy of activities together with the details of the data items and control signals that flow between them. It specifies what activities can take place (without saying when the activities will be activated, whether or not they terminate on their own, and whether they can be carried out in parallel) and what data can flow (without saying whether and when it will).

Note: We distinguish between static and dynamic aspects of a system. The static aspects are captured by the data and structure views, all the other system views from above being concerned with the dynamic aspects. We can sometimes specify both of these aspects (dynamic and static), using the same specification technique. In the following sub-section we focus on the dynamic aspects of a system since, on the one hand, dynamics is more complex than statics, and on the other, the expressiveness of specification techniques differ significantly regarding dynamics.

2.2 Underlying Model of Computation

Specification techniques that address the dynamic aspects of systems are highly complex. In this section, we focus on different characteristics that differentiate them from each other. A particular subset of such characteristics refers to 'what behaviour can be described', i. e., to a relation between a specification technique and an underlying 'Model of Computation'. Thereby, specification techniques and 'Models of Computation' are orthogonal concepts, representing abstractions on *how* to describe and *what* to describe, respectively.

Figure 3: Instantaneous feedback

2.2.1 Definitions

In the literature on model-based development, different authors speak of 'specification technique', 'modelling language', 'model of computation', 'model of execution', 'modelling theory', etc., and unfortunately often mean different concepts. Unsurprisingly, this leads to confusion and misunderstanding. In Section 1, we presented the differences between 'specification techniques' and 'modelling languages'. Below we focus on Models of Computation.

Models of Computation (MoCs) depict different behavioural aspects of systems and answer the question *'what is modelled?'*. They provide a framework within which a system can be modelled. Our understanding of MoCs is similar to that in [HLL+03, ZLL07].

Customarily, a model is an abstraction of a target system, which describes the significant features of a system and abstracts away from irrelevant ones. The modelling is successful if the resulting model is useful in later phases of the development; for instance, if the model can be used for analysis purposes (like, e. g., static analysis, simulation, generation of testing cases) or for automatic code generation. The key issue for such a success is the balance between describing a target system as accurate as possible and abstracting away from irrelevant details – i. e., finding the right level of abstraction. This level of abstraction, which can be defined formally and has a precise semantics, is a Model of Computation. A system is then modelled by means of a MoC. MoCs can be allocated to appropriate system views (see Section 2.1) dealing with system behaviour.

MoCs usually define a formal semantics and can be perfectly suited for design and analysis, but they not necessarily possess a clear execution semantics. For example the finite state machine (FSM) in Figure 3 depicting instantaneous feedback, if implemented in hardware and on the synchrony assumption, does not necessarily show a deterministic execution behaviour. Different ways to attain a deterministic execution are discussed in [BCE+03]; among others, microsteps as in, e. g., VHDL [VHD09], unique fixpoint as in, e. g., Esterel [Est05], constraints as in, e. g., Signal [AR79], and prohibition of zero-delay loops as in, e. g., Lustre [CPHP87]. Thus, a **model of execution** is an auxiliary semantics, which enriches a given one by an execution strategy, that is used to execute a concrete model on a concrete hardware.

As described above, MoCs depict different behavioural aspects of software systems. In order to classify and compare different MoCs, we itemise the aspects they can address. We call them **aspects of MoCs** and describe them in Section 2.2.2. A fine differentiation of MoCs can be done by means of the these aspects.

2.2.2 Aspects of Models of Computation

We introduce the aspects of MoCs, also called behavioural aspects. They address features of systems to be modelled and allow to capture the similarities and differences between MoCs. The aspects together with their respective variants are listed below:

Time. Time is used for single events to establish the precedence relation and define the distance between them, or for pairs of sequenced (back-to-back) events and to compare the intervals between them.

- *Continuous.* The system is based on continuous time values.
  - *Event discrete.* Data are discrete, i. e., in form of discrete events.
  - *Data continuous.* Data form a continuous domain.
- *Discrete.* The system is based on discrete time values.
  - *Partial order.* The precedence relationship is only partially defined for events due to a discretisation error (we speak here about sequential systems). In such case, two events which are not simultaneously in reality can receive the same timestamp.
  - *Complete order.* The precedence relationship is defined for all events (we speak here of sequential systems). Time intervals are small enough to allow only one event occurring at a time.[1]
- *No time.* Only precedence relationship is defined but no information about the distance between two events is provided.

Clock. Clock is a device[2] to measure the time.

- *Local/multiple.* Only local clocks are available. In a distributed system, this means the presence of multiple clocks with or without synchronisation. If synchronisation takes place, the synchronisation error should be taken into consideration.[3]
  - *Partial order.* The precedence relationship is only partially defined for events.
  - *Total order.* The relationship of event precedence is totally defined.
- *Global/single.* A global clock is available. In a non-distributed system, global and local clock coincide.

Concurrency. Concurrency means that more than one process is executed at a time.

- *No.* Sequential computation.
- *Yes.* Concurrent computation.
  - *Synchronisation.* Synchronisation of entities running concurrently might be necessary.
  - *Communication.* Communication between entities running concurrently might be necessary.

[1] In a sequential system, we do not have the notion of simultaneity. Even if the events in the real word are simultaneous, the system can not capture simultaneity because the observer is sequential. Sparse time can be used to handle simultaneity in a concurrent framework, which means that evens occurring simultaneously have the same time stamp.

[2] The clock can be a physical device or a software facility.

[3] All known clock synchronisation algorithms have a limited accuracy. For instance, NTP has an accuracy from 10 millisecond in a global area network to 200 microseconds in a local area network, whereas the IEEE 1588 standard is able to synchronise clocks up to nano-seconds accuracy.

1. *Explicit/implicit.* Explicit communication via message passing, implicit communication via shared memory.
2. *Synchronous/asynchronous.* Synchronous communication only takes place if both, sender and receiver, are ready for data exchange. In case of asynchronous communication, the sender never blocks, i. e., it does not wait for the receiver to be ready for data exchange.[4]
3. *Deterministic/non-deterministic.* The communication is deterministic if all data sent arrive at the receiver undamaged and in the order they were dispatched. On the contrary, the communication is non-deterministic if one or both of those conditions is not true. In case of non-deterministic communication, there are the concepts of latency, packet loss, falsified data:
   - *Latency* is the delay between sending and arrival of data.
   - *Packet loss* specifies how many data is lost.
   - *Falsified data* means damage of data during delivery, e. g., due to bit inversion.
4. *Dynamic/static.* A way to transmit a datum from one entity to another is dynamic (differ for each datum) or static.

Behaviour. Behaviour is a manner in which a software system behaves, i. e., how and when it acts or reacts in response to external or internal stimuli.

- *Time-triggered (periodic).* Computation of a task takes places during a predefined time slot. Scheduling between all tasks is needed in advance.
- *Event/demand-triggered (aperiodic).* Computation takes place after arriving of a sporadic (input) event.

Flow. Flow defines how the inputs/instructions of a system are interpreted in order to compute outputs.

- *Control flow.* Control flow describes the reaction to input events according to internal states, algorithms — reactive domain.
- *Data flow.* Data flow describes data processing, stream processing — transformative domain.

Determinism. Determinism concerns the relation between the data that is input to and the data that is output by a system.

- *Deterministic.* A system is deterministic if it always generates the same outputs after receiving the same inputs.
- *Non-deterministic.* A system is deterministic if it may generate different outputs after receiving the same inputs.

Data. Data describes which kind of data a system can deal with.

- *Continuous.* Continuous data can be assigned an infinite number of values between whole numbers.
- *Discrete.* Discrete data means that the data set is enumerable.
- *Finite.* Finite data sets are discrete and contain a finite set of values, only.

[4] In the theoretical definition, the length of a sender's buffer/message queue is considered to be unlimited. In the practical implementation, however, the sender is limited by the length of a buffer/message queue.

Note: Many of the aspects presented above are dependent on each other. For example, if a MoC addresses concurrency, then it also must address communication and synchronisation. On the contrary, if a MoC does not address concurrency but only sequential computation, then the concepts of communication and synchronisation are superfluous and can be safely ignored.

Discussion. The above aspects have different degrees of 'expressiveness.' For example, some of them can describe timing behaviour, some others cannot. Depending on the kind(s) of systems to be described, the chosen MoC needs appropriate expressive power to model them. If, for instance, we model a non-timed critical system (e. g., a data-mining application), it is important to express in which order the actions take place (e. g., exploration/data preparation, model building and validation, application of the model to new data), but it may be unnecessary to express how long each or all of those actions take. If by contrast a real-time system is to be modelled (e. g., a braking system for a car), then time is of utmost importance, as timing constraints are a part of the system functionality.

Intuitively, less expressive aspects of MoCs are included in more expressive ones. For example, it is possible (a) to go from a timed model to an untimed model just by omitting time, or (b) to express explicit synchronous communication channels by means of two explicit asynchronous communication channels[5], or (c) to express implicit communication with explicit asynchronous communication with queue length 1. We moreover assume that some values of aspects of MoCs are not disjunctive but rather inclusive. This means, simpler constructs can be expressed using more complex ones.

2.2.3 Specification Techniques and Aspects of MoCs

There is a many-to-many relation between specification techniques and the behavioural aspects listed in 2.2.2. Each of these aspects, on the one hand, can be described using multiple specification techniques. A single specification technique, on the other, can describe different aspects of MoCs.

The relation between the specification techniques and the aspects of MoC is presented in Tables 2 and 3. Each line of Table 3 contains a specification technique and its columns represent tuples containing the values of different aspects of MoC. If multiple tuples in the table are equal, then the corresponding specification techniques have the same expressiveness according to the considered behaviour aspects. Since we do not claim completeness of the list of aspects of MoC, by simply having the same characteristics we cannot assert that these specification techniques be completely equivalent. However, when two specification techniques have the same characteristics, this is a good hint that the specification techniques are compatible and that they can be translated into each other without loss of information. The complete relations between the specification techniques and the aspects of MoCs is developed in Section 3.

[5] In this case, the synchronous communication is emulated through, e. g., a software loop where the sender is waiting for the response.

2.3 Typical Software Development Process Stage

Some specification techniques can be used in all stages of software development process while others are typically used only in some stages. Below we shortly present software development stages where specification techniques are typically used.

A software development process is a structure imposed on the development of a software product. Traditionally, the software development process is constituted by the following phases: requirements engineering, design, implementation, verification and maintenance. Different development process models (like, e. g., waterfall, RUP, V-model) define the order of these development activities. In the following, we describe the single development activities and give hints about the characteristics that the models used in each activity should possess.

Requirements engineering. In this phase, the requirements of the proposed system are collected by analysing the needs of the users. Requirements engineers are concerned with establishing what the system has to perform and do not determine either how this task is to be achieved or how the system will be designed or built. The resulting requirements specifications typically describe the system's functional and non-functional requirements as expected by the user. The requirements engineering phase consists of requirements elicitation and requirements analysis, and results in a consistent specification.

Typical stakeholders involved in the requirements engineering include the business domain experts and up to people with a more software technical background. In model based development, models are used from the requirements engineering phase to enable the unambiguous communication between different stakeholders, verification of consistency and completeness of requirements, or the validation of requirements. For example, specifications that can be simulated are used in order to enable end-users to an early requirements validation.

System and software design. In this phase, system engineers analyse and understand the proposed system by studying the requirements specification in order to derive possible implementations out of the high-level requirements. These activities result in a high-level or logical architecture, which comprises modules, externally visible properties of each module, and relationships between them. High-level architectures are concerned with making sure that the system will meet the user requirements, and are the starting point for the first division of labor between different teams.

Models used in this phase need to be easy to refine in subsequent phases. They need to be appropriate for describing the architecture in the large and detailed enough to specify the interfaces between different sub-systems.

Detailed design. In this phase, the high-level architecture is translated to the low-level design or the concrete operational guide following the architecture. The system engineers are concerned with developing specific designs by taking into account design patterns, best practices, and physical topologies. The engineers decide which functionalities are to be performed by software and which ones by hardware.

This phase results in a low-level design model, which is broken up into small modules, each of them so explained that the programmer can start coding directly. This model contains a detailed functional logic of each module.

Implementation. Here, software engineers actually program the code that runs on the target platform.

In model-based development, the code is generated from higher-level models produced in the detailed design phase. The models at this level should be thorough enough to describe the system in the highest detail; in the best case, they are executable.

Validation and Verification. Validation and verification are steps done to ensure that the developed system meets the user vision (*validation*) as well as the technical requirements (*verification*) that guided its design and development.

Specification models developed in the early phases of requirements engineering serve now as specifications for validation (e. g., running the user acceptance tests) and verification (e. g., running a model checker).

To sum up, different specification techniques produce different kinds of models, some being more adequate in early phases, others being adequate in the detailed design. By using an inadequate specification technique, we can loose information or hinder the subsequent refinement of models.

2.4 Supported Abstraction Layer

In order to keep the complexity under control, we need to use models at different levels of abstraction. High-level models specified in early phases should be incrementally enriched with information in the following steps. Different specification techniques can be used to define models at different levels of abstraction. Thereby, we aim to classify the specification techniques according to the abstraction layer that they typically support.

In [RST09] we described a framework that comprises the landscape of models that are built during the system development as depicted in Figure 4. This framework aims at supporting a systematic development process of embedded software systems based on the seamlessly integrated use of different models at different stages in the development process. It comprises two different dimensions: one given by the *level of granularity* at which the system is regarded, and one given by different *software development views* on the system:

- *Levels of granularity.* By regarding a system at different levels of granularity, we structurally decompose the system into sub-systems, and sub-systems into sub-sub-systems until we reach basic blocks that can be considered atomic. Decomposing the whole system into smaller and less complex parts, helps to reduce complexity following the "divide and conquer" principle.
- *Software development views.* A system can be regarded from different perspectives, each perspective representing different kinds of information about the system. We call each of these perspectives a "software development view". Our framework contains the following development views: the functional view, the logical (structural) view, and the technical view. The views aim to reduce the complexity of the software development by supporting a stepwise refinement of information from usage functionality to the realisation on hardware.

For the classification of the specification techniques, we focus on the different development views:

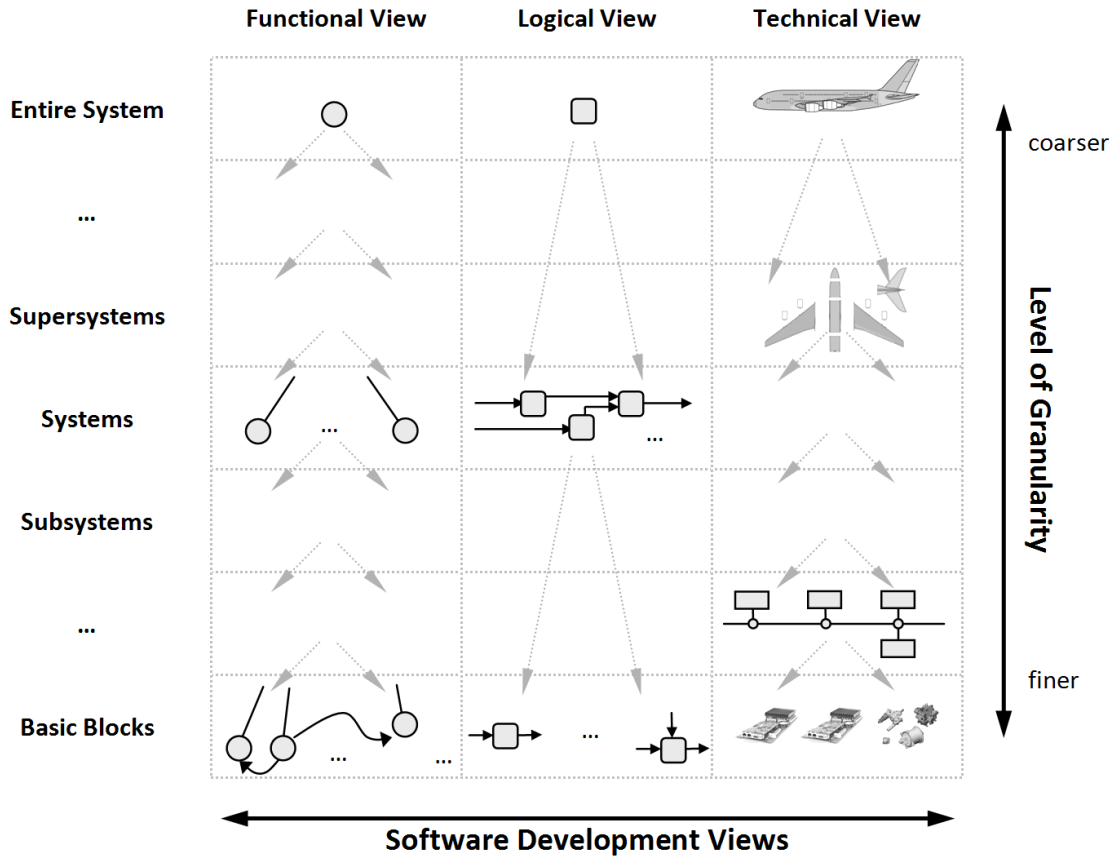


Figure 4: Two-dimensional abstraction: levels of granularity and software development view

Functional view. The functional view focalises on a specification of the functional requirements of the system to be. It structures the system's functionality according to the behaviour required by the system's users. By using formally founded models, this layer provides the basis to detect undesired interactions between functions at an early stage of the development. Due to the high level of abstraction, this layer is a step toward closing the gap to the informal requirements. Thus, it provides the starting point of a formally founded model-based development process.

The models of the functional view are used to enable the unambiguous communication between different stakeholders, verification of consistency and completeness of functional requirements, or the validation of them. Therefore, we need here specification techniques that focus on the interaction between the system and its environment rather than the internal structure of the system. Desirably, these techniques are executable thus enable end users to an early validation of the requirements.

Logical view. The logical view addresses the logical component architecture. Here, the functional hierarchy is decomposed into a network of interacting components that realise the observable behaviour described at the functional layer. Due to this layer's independence from an implementation, the complexity of the model is reduced and a high potential for reuse is created.

Models of the logical view need to be appropriate for describing the architecture in the large and detailed enough to specify the interfaces between different sub-systems. A model of the logical view should structure the system functionalities in a way that facilitates their changing, reuse, deployment, and, in general, maintenance. Therefore, we need here specification techniques that focus on the internal structure of the system. These techniques should be able to decompose the system into more manageable sub-systems handled by different teams, and to capture major interfaces between sub-systems.

Technical view. The technical view describes the hardware platform in terms of electronic control units (ECUs) that are connected to busses. A deployment mapping is specified that connects the logical components defined in the logical layer onto these ECUs. Thus, a physically distributed realisation of the system is defined and the complete middleware which implements the logical communication can be generated. Additionally, the interaction between the logical system and the physical environment via sensors and actuators is modelled.

The models of this view should be executable on the target platform and should be detailed enough and describe the system in the highest detail. Thus, the appropriate specification techniques should be able to address such issues like bus protocols, resource management, etc.

Note: The views presented in this paragraph should not be mixed with those described in 2.1. The views here all aim to provide a systematic way to structure the software modelling process, but are orthogonal to the views on a software system.

2.5 Compositionality

Modular development is a software design technique that dictates to compose software from separate parts, called modules. Modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components. Modules can be independently created and then used in different systems to drive multiple functionalities.

For the successful use of a specification technique in the modular development, this technique has to be compositional. A technique is compositional if it can determine the meaning of a complex expression by the meanings of its constituent expressions and the rules used to combine them.

2.6 Tool Support

Every tool uses a modelling language that, as explained above, is based on a set of specification techniques. Therefore, tool support does not exist for a specification technique alone but for a modelling language. In practice, the existent tools represent the main means to propagate modelling and specification techniques in industry. For example, UML and its dialects are so widespread since there exist many tools that support to a certain degree working with UML models. We distinguish among the following categories of functionalities offered by a tool to its users:

Basic functionality. The basic functionality includes the possibilities to edit, save, and manage versions of models. Without the basic functionality, no other complex functionality can be supported.

Analysis functionality. Analysis functionality represents the possibility to perform different analyses on models such as defining and verifying different consistency conditions, simulation of the model, verification of different characteristics. The type of analysis that can be performed is strongly dependent on the specification techniques supported by the tool (see Section 2.7).

Synthesis functionality. Synthesis functionality represents the possibility to transform a model, or generate the code. This functionality is very important to enssure the integration of a tool in the process: it allows the results of the tool to be further used by other tools. Again, depending on the specification techniques supported, the tool can offer synthesis to different kinds of models (e. g., tools that support only class diagrams cannot be used for the synthesis of behaviour).

2.7 Analysis Techniques

The verification and validation of software have a central role in the field of safety critical embedded systems. For example, the damages due to errors (almost) always cost money and even human lives. Depending on the criticality of the system part that is under development, we need different kinds of maturity for analysis; for example, the avionics certification standard DO-178B [RTC82] defines five levels of safety that different system parts must comply. Besides validation that the system implements the desired functionality, it must be verified that the safety conditions, real-time constraints, etc., are satisfied.

The verification of large systems is notoriously difficult or often even impossible due to the combinatorial explosion of the number of cases to be considered and to the size of the search space in case of exhaustive approaches. In model-based development, the use of adequate modelling techniques promises to make the verification more approachable/feasible.

In this document, by analysis techniques we denote the whole range of approaches that deal with the verification and validation of systems. Analysis techniques check that a systems does indeed meet its specification. Below we summarise the most well-known analysis techniques used for verification and validation of systems: inspections, reviews and walkthroughs, system testing, system simulation, formal verification, and runtime verification.

2.7.1 Inspections, Reviews, and Walkthroughs

Reviews. Software reviews [CLB03] are the most general analysis techniques: they can be used on all development artifacts (from pure text to richly defined models) as well as in all development phases (from requirements analysis up to implementation). They are fundamentally manual methods, that are however supported by adequate tools and can be backed up by methodologies. In order to make these analyses in a disciplined manner, and to ensure a high confidence in their coverage, usually they are done on the basis of checklists with different system characteristics of interest.

Being highly manual, these analyses are expensive and slow and therefore mostly used to check different properties of a system that cannot be checked in an automatic manner. This happens in the case of weakly defined models or in early development phases. These analyses are primary means to validation.

There are some variants of the review methods, namely walkthroughs (more informal) and inspections (more formal), which are described next.

Inspections. The inspection as described in the IEEE standard [IEE97] is basically the Fagan Inspection (see [Fag02]), a five-step, formalised process. Code inspections are a highly efficient test method which cannot be substituted by any other test methods. They are time consuming but find an important number of errors if done properly. However, their success depends on the methods and checks applied and on the diligence of the inspectors.

Walkthroughs. Walkthroughs differ significantly from inspections. A walkthrough is less formal, has fewer steps, and does not use a checklist to guide or a written report to document the team's work. A walkthrough can have a twofold purpose. First of all, it can be performed to evaluate a software product in order to:

- find anomalies,
- improve the software product,
- consider alternative implementations, and/or
- evaluate the conformance to standards and specifications.

Walktrough is a method which should be used throughout the design phase of a software product to collect ideas and inputs from other team members, what leads to an overall improvement of the product. The second objective of a walkthrough is to exchange techniques and perform training of the participants. It is a method to raise all team members to the same level of knowledge regarding programming styles and details of the product.

2.7.2 Testing and Simulation

Testing. Testing is a basic technique used to check the correct implementation of a system with respect to a property. Typically, the property is derived from the formal specification (Figure 5). A test case is a basic unit of testing and represents the checks of a particular set of inputs and outputs of a system. Testing is a method to identify errors, but it cannot guarantee the complete correctness because usually the set of test cases is not exhaustive. Ideally, tests of a system have a high coverage and consider all the relevant situations that can occur.

A testing procedure is called *black-box* if the test cases are defined without any information about the internal structure of the system. *White-box* testing, on the contrary, uses the knowledge about the system implementation in order to generate test cases. The most important metric for a white-box testing procedure is the code coverage. It is applied to the source code specification and describes the degree to which the source code of a program has been tested. In order to specify this degree, there are a number of coverage criteria [Cor96], e. g., function, branch, or path coverage.

In the most basic form, tests are manual: an engineer defines a certain set of input data and checks whether the system generates the expected output if stimulated with the defined inputs. Many times, however, tests are done in an automatic manner: based on

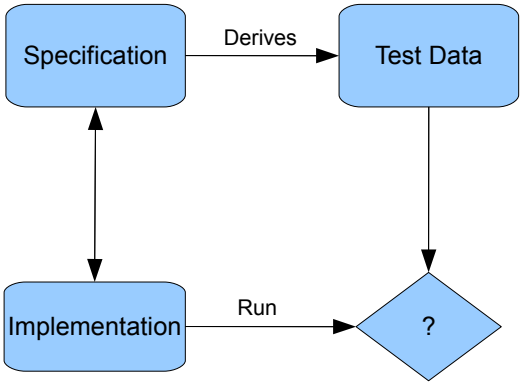


Figure 5: Testing framework

a specification, a set of test cases is generated that is to be executed on the system. For example, MSC-based specifications are used as inputs for automatic test case generation.

Simulation. Simulation is typically an execution of the design of the system-to-be, for observing interactively the behaviour of the system and checking its correctness. The verification by simulation is typically made by first defining input sequences, which can but need not be random, suitable to test certain functionality, and afterwards simulating the system with those input sequences; the produced output sequences are finally checked to determine if the system behaves as expected. The verification is not exhaustive but limited to the cases scheduled by the designers; nevertheless, simulation is still a good and fast way to verify early designs. In the framework of specification techniques, it is convenient to have a graphical visualisation of the simulation and that the animation highlights directly over the specification, for aiding the discovery of eventual errors.

2.7.3 Formal Verification

Formal verification [CW96] means formally proving the system's correctness with respect to a certain formal specification or property, using formal methods. We subdivide the formal verification techniques in static analysis on the one hand, i. e., techniques that elaborate information on the behaviour of a program without executing it, such as abstract static analyses, model checking and bounded model checking; and theorem proving on the other hand, meaning the process of finding a proof of a property where the system and its desired properties are expressed as formulas in some mathematical logic. We describe in more detail the two major formal verification techniques:

Model checking. Given a model of a system, model checking tests automatically whether this model meets a given specification. Typically, the system behaviour is abstractly represented by a graph, where the nodes represent the system's states and the arcs represent possible transitions between the states.

The properties to be checked are usually expressed in a temporal logic. Temporal logic is a class of modal logic and a powerful means to express several properties about systems. It

extends propositional logic by the addition of time operators, in the sense that formulas can evaluate to different truth values over time. There are different types of temporal logic notations that correspond to different views of time: branching vs. linear, discrete vs. continuous, past vs. future, real time. Examples of temporal logic formal languages are linear temporal logic (LTL), computational tree logic (CTL) and timed CTL. Examples of properties that can be expressed and consequently verified by model checkers are:

- Reachability: some particular situation can be reached
- Safety: something never occurs
- Liveness: something will ultimately occur
- Fairness: something shall (or shall not) occur infinitely often
- Deadlock-freeness: the system can always evolve to a successor state

While a violation of a safety property can be detected by a finite sequence of executions steps, a violation of a liveness property may be detected only by an infinite execution of the system. Consequently, liveness properties are harder to be verified by model checkers.

Model checking is completely automatic and relatively fast, sometimes producing an answer in a matter of minutes. Model checking can be used to check partial specifications, and can so provide useful information about a system's correctness even if the system has not been completely specified. An important feature of model checking is that it produces counterexamples, which usually represent subtle errors in design, and thus can be used to aid in debugging. The main disadvantage of model checking is the state explosion problem: even small system may result in a huge state space. This problem has been addressed in many ways, by optimisations to save time and more importantly space. Thank to that, model checking has become powerful enough to be widely used in industry in the verification of newly developed designs.

Theorem Proving. Theorem proving [CW96] is the process of showing that the model satisfies the specification by means of formal proof, i. e., a sequence of statements each of which is an instance of an axiom of the system or follows from previously proved statements by means of inference rules. The basic idea is that both the model and the specification are represented within a suitable logic. This logic is given by a formal system, which defines a set of logical (i. e., universally valid) axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the logical axioms and the proper axioms of the system.

Theorem provers need not exhaustively visit the program's state space to verify properties. Consequently, a theorem prover approach can reason about infinite state spaces and state spaces involving complex data types and recursion. Theorem provers support fully automated analysis in restricted cases only, so usually the method is semi-automatic.

Theorem provers are increasingly used today in the mechanical verification of safety critical properties of hardware and software designs, but the generated proofs can be large and difficult to understand. Therefore, a great deal of user expertise and effort is required. This requirement presents perhaps the greatest barrier to the widespread adoption and usage of theorem provers.

2.7.4 Runtime verification

Runtime verification [CM04] is a verification technique that combines formal verification and program execution. It is the process of detecting faults in a system under scrutiny by passively observing its input/output behaviour during its normal operation. The observed behaviour of the target system, e. g., in terms of log traces, can be monitored and verified dynamically to satisfy given requirements.

Runtime verification is performed while the real system is running. Thus, it increases the confidence on whether the implementation conforms to its specification. Furthermore, it allows a running system to reflect on its own behaviour in order to detect its own deviation from the prespecified behaviour.

Continuous monitoring of the runtime behaviour of a system can improve our confidence in the system by ensuring that the current execution is consistent with its requirements at runtime. In the literature, at least the following four reasons are mentioned in order to argue for runtime verification:

- If you check the model of your system you cannot be confident on the implementation since correctness of the model does not imply correctness of the implementation.
- Some information is available only at runtime or is convenient to be checked at runtime.
- Behaviour may depend heavily on the environment of the target system; then it is not possible to obtain the information necessary to test the system.
- In the case of systems where security is important or in the case of critical systems, it is useful also to monitor behaviour or properties that have been statically proved or tested.

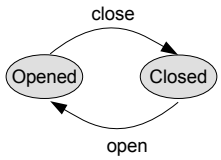
2.8 Typical Notations

The term notation is a synonym for concrete syntax and is the most visible part of a modelling language to its end-users. In order to realise a model, engineers need to use a concrete notation. Typically, there are textual notations, diagrammatic notations, or tables. To concrete syntax belongs everything related to the layout, colors used, fonts, etc. – i. e., everything that a user sees when describing a model.

It is important to note that the choice of a particular notation belongs to the pragmatics of a language, and is independent from the language definition per se as well as from the language's capabilities. The same specification concept can be represented graphically in different tools by using different shapes or colors. Or vice versa, different specification concepts can be represented through the same graphical notations in different tools.

Many specification techniques favour a standard concrete syntax, while others can be often seen in different notations. In Figure 6 we present different typical notations for describing state automata. We have a pure textual notation, two graphical notations, and a table. Each of these notations is adequate (from pragmatic reasons) for different purposes: for example, a graphical notation can be animated more easily, while the textual notation is more compact.

```
automaton {
  state Opened, Closed;
  transition open: Closed → Opened;
  transition close: Opened → Closed;
}
```



Src. / Dst.	Opened	Closed
Opened	--	close
Closed	open	--

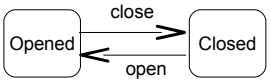


Figure 6: Examples of notations for describing automata

3 Classification of Specification Techniques

In this section we present a classification of several widespread specification techniques according to the framework introduced in Section 2.

A first step in preparation of this document is to decide which specification technique to consider for our classification. Following our experience, and the feedback provided by our industry partners from the tools questionary, we decided in the first stage to restrict ourselves to a handful of common specification techniques developed in the academia and/or used in practice.

Table 1 shows at a glance the specification techniques and how we classified them. In the leftmost column the specification paradigms are presented, and in the second column corresponding specification techniques. Each line in the table corresponds to a specification technique and each one of the further columns correspond to a criterion defined in Section 2.

In Tables 2 and 3 we present the specification techniques for modelling of behaviour in more detail, namely by presenting which behavioural aspects can be expressed with a certain specification technique. As mentioned in 2.2.3, if different lines in the table are equal, then the corresponding specification techniques are equivalent according to our criteria. This is a first hint of the equivalence between the expressive power of different specification techniques.

Outline of this section. Each of the following subsections address the specification techniques that belong to a certain specification paradigm. We have chosen the following specification paradigms: component-based development (Section 3.1), use cases (Section 3.2) automata (Section 3.3), control-flow-based specifications (Section 3.4), data-flow-based specifications (Section 3.5), function block specifications (Section 3.6), Petri nets (Section 3.7), sequence diagrams (Section 3.8), constraint-based specifications (Section 3.9), algebraic specifications (Section 3.10), and process-algebra-based specifications (Section 3.11).

Each of these sections has the same structure: we present an overview over the specification paradigm and give a mathematical model where there is one that is established in the community. In the following we present the set of important specification techniques for the paradigm, and continue with the classification of the specification techniques according to the criteria presented in Section 2.

Specification Paradigm	Specification Technique	System Views[6]	Development Phase[7]	Abstraction Layer [8]	Tool Support	Analysis Techniques [9]	Typical Notation
Component-Based Design	Component Diagram	S	SSD, DD	L	UML	testing (Rational Rhapsody)	graphical
Use-Case-Based Design	Use Case Diagram	F	RE	F	UML	testing (Rational Rhapsody)	graphical, textual
Automata	UML State Machine	B (P)	RE, SSD, DD, I	L (F, T)	UML	testing, simulation (WinA&D, ASCET, Rational Rhapsody)	graphical, tabular
Control Flow Specification	UML Activity Diagram	P (B)	RE, SSD (DD, I, VV)	F	UML	testing, simulation, model checking (Rational Rhapsody, Toolkit for Conceptual Modelling)	graphical
Data Flow Specification	Data Flow Diagram	B	RE, SSD (DD)	F, L	Visio, Simulink, SCADE	testing and verification (Simulink, SCADE, WinA&D, Rational Rhapsody)	graphical
Function Block Specification	Function Block Diagram	F	RE, SSD, DD	F, L	SysML Siemens Simatic	model checking, testing, simulation, formal verification (PRISE, ASCET, SCADE, Simulink)	graphical
Petri Nets	Simple Petri Nets	B (P)	DD	L	CPN Tools	simulation, model checking (CPN Tools, PRES+)	graphical
Sequence Diagrams	UML Sequence Diagram	B	RE, SSD	F, L	UML	testing, simulation, model checking (Rational Rhapsody, TURTLE Tool)	graphical
Constraint-based Specification	OCL	all	DD, I, VV	all	Dresden OCL	testing	logic formula
Algebraic Specification	HOL-CASL	D	DD, VV	F	Isabelle	static analysis, theorem proving (Heterogeneous Tool Set)	textual
Process Algebra	CCS	B, P	SSD, VV	F, L			textual

Table 1: Classification of Specification Techniques

[6]D = data, S = structure, B = behaviour, P = process, F = functional
[7]RE = requirement engineering, SSD = system and software design, DD = detailed design, I = implementation, VV = validation and verification
[8]F = functional, L = logical, T = technical
[9]All specifications support inspections, reviews and walkthroughs

Specification Paradigm	Specification Technique	Aspects of MoC						
		Time	**Clock**	**Concurrency**	**Behaviour**	**Flow**	**Determinism**	**Data**
Automata	UML State Machines	discr	glob	data-synchr	event	contr	no	discr
	I/O Automata	discr	glob	data-synchr	time	contr	no	discr
	Timed Automata	cont	local	data-synchr	time & event	contr	no	discr
	Hybrid automata	cont	local	data-synchr	time & event	contr	no	cont
	AutoFocus automata	discr	glob	asynchr	time	contr	no	discr
Control Flow Specification	UML Activity Diagrams	no	no	asynchr	event	contr	no	discr
	CFGs	no	no	asynchr	event	contr	no	discr
Data Flow Diagrams	Matlab Simulink, Lustre	discr	global	synchr	event	data	no	cont
	Kahn process networks	discr	global	asynchr	event	data	det	cont
	Real-Time DFDs	ev discr	global	asynchr, explicit	event	data & contr	no	cont
Function Block Specification	(F)FBD	–	–	–	event	data	no	discr
	EFFB	–	–	–	event	data & contr	no	cont, discr
Petri Nets	Simple Petri Nets	discr	local, global	asynchr	event, time	data	no	cont, discr
Sequence Diagrams	MSC / UML Sequence Diagrams	ev discr	local (po)	expl, asynch, nondet	event	data	no	–
	LSC	ev discr	local (po)	expl, asynch, nondet	event	data	no	–

Table 2: Classification of Behavioural Specification Techniques on the basis of MoC (1/2)

Specification Paradigm	Specification Technique	Aspects of MoC						
		Time	**Clock**	**Concurrency**	**Behaviour**	**Flow**	**Determinism**	**Data**
Constraint-based Specification	OCL	no time	mult	seq	event	control	no	discr
	CLP	no	single	seq	event	control	no	discr
Algebraic Specification	HOL-CASL	no	no	no	no	not cov	not cov	depends
	CALS-LTL	prec	no	no	no	not cov	not cov	depends
Process Algebra	CCS	ev discr	local (po)	expl, asynch, det	event	data	no	discr
	CSP							
	ACP							

Table 3: Classification of Behavioural Specification Techniques on the basis of MoC (2/2)

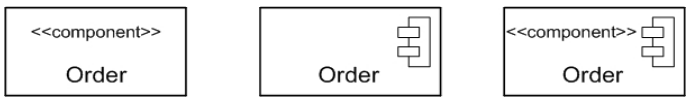


Figure 7: UML 2 components of the Component Diagram

3.1 Component-Based Design

3.1.1 General Description of the Paradigm

Component-based design focus on the construction of the system's architecture by decomposing a software system into its components. It is used from early phases of the software development process on leading from high-level architectures to more detailed ones.

The component-based design shows the software components (incorporating parts of system functionality), usually as black-boxes, and interfaces in-between them. Main purpose is to show the structural relationships between components of a system.

The main advantages of the component-based design are the following. First, it is helpful or even necessary to master the complexity of large software systems by constructing them in a structured and modular way. Secondly, the software components represent modular building blocks that can be easily reused in a new system and/or substituted by new components as long as the interface remains the same. Thirdly, component-based development is useful for distributed development since software components can be developed independently as long as their functionality and interfaces are specified properly.

3.1.2 Specification Techniques Comprised in the Paradigm

UML 2 component diagram [Bel04]. A UML 2 component diagram is as an architecture-level artifact used for systems and software design or/and for detailed design. Here, the components are autonomous, encapsulated units within a system or sub-system that provide one or more interfaces. A component is presented in a UML 2 diagram as a rectangle with the component's name and the component stereotype text and/or icon. The component stereotype's text is 'component' and the component stereotype icon is a rectangle with two smaller rectangles protruding on its left side. The three equivalent notations for a component are presented in Figure 7.

Additionally, the interfaces provided and/or required by a component can be specified. The interfaces provided represent the formal contract of services the component provides to its consumers/clients. There are two equivalent possibilities for this shown in Figure 8. A component notation can have additional compartments stacked below the name compartment with 'required interfaces' and/or 'provided interfaces' followed by the corresponding interfaces lists. Or a component can be presented as a simple rectangle with an provided interface(s) presented as a link with a complete circle at their end (sometimes called 'lollipop' symbol) and/or with an required interface(s) presented as a link with only a half circle at their end (sometimes called 'socket' symbol).

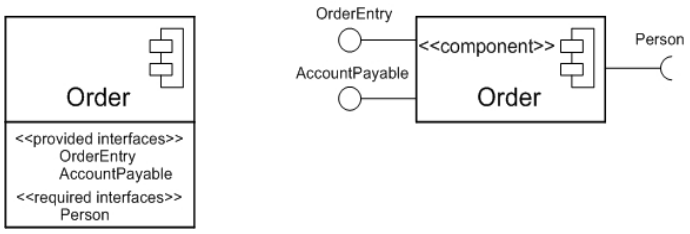


Figure 8: The UML 2 Component Diagram: different specifications of interfaces

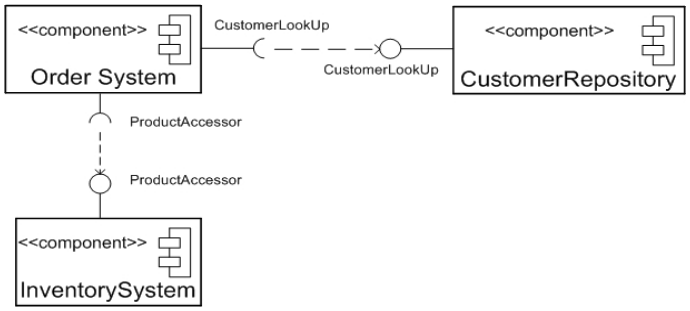


Figure 9: The UML 2 Component Diagram: Example

An example for an UML 2 component diagram is presented in Figure 9.

Rich Components [DVM+05]. The handling of non-functional properties is an important issue by component-based design. Such properties (e.g. time and QoS) have to be modelled, i. e., the component-based design has to be enriched accordingly. The Rich Component Model developed by the OFFIS team allows for specifying and verifying functional and non-functional requirements, their horizontal, vertical, and inter-viewpoint composition at different abstraction levels [DVM+05]. To capture all aspects of a rich component different specification techniques based on different paradigms are used, such as component diagrams, composite structure diagrams, sequence diagrams, state machines etc. Rich components extend classical component models (from UML 2.0) by

- extending component specifications to cover all viewpoints (functional and non-functional ones such as real-time, safety, resources, power, etc.),
- explicating the dependency between the promises of a component specification and assumptions on its environment;
- providing classifiers to such assumptions, relating the positioning in a layered design space (horizontal, up, down), as well as specifying confidence information of these assumptions.

Focus [Bro07]. The FOCUS approach decomposes a system into a network of software components, see example in Figure 10. The software components are black-box entities defined by their names and formally specified interfaces. The interfaces are connected through (typed)

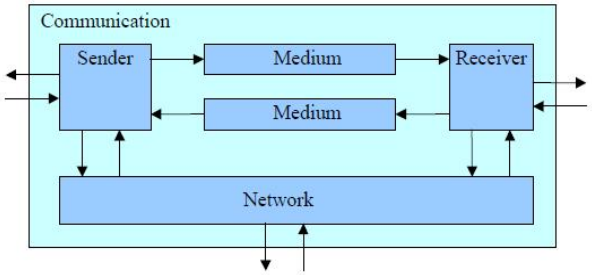


Figure 10: FOCUS: Network of software components

channels where message are interchanged. The intercommunication between the components is described formally by stream processing functions mapping input messages received on the input channels to output messages sent via the output channels. The formally specification of the syntactic interfaces and the interface behaviour allows to formally define composition operators and refinement.

3.1.3 Classification of the Specification Techniques

Covered system views. The specification techniques based on the 'Component-based Paradigm' paradigm cover the structure (also called organisational or architectural model) system view.

Underlying model of computation. The specification techniques described in this section do not cover behavioural aspects of a system. Hence, they cannot be classified according to underlying model of computation.

Addressed stage of the development process. These techniques are often used in the system and software design and/or in the detailed design phases of the development process.

Supported abstraction layers. The corresponding techniques support the logical view.

Compositionality. All component-based specifications support compositionality by nature requiring to split a system into components.

Available tool support. There is a rich tool support available for the specification techniques based on this paradigm. UML component diagram is supported by each UML editor.

Analysis techniques. Manual analysis techniques are used, i. e., inspections, reviews and walkthroughs. Static analysis in form of interface correctness checking can also be feasible.

Typical notations. Typical notation is graphical, but textual might also be possible.

3.2 Use-Case-Based Design

3.2.1 General Description of the Paradigm

Use cases, introduced by Jacobson in [Jac04], describe the system from the user's point of view. They describe "who" can do "what" with the system from a black-box point of view. A complete and unambiguous use case describes one aspect of usage of the system without presuming any specific design or implementation. Thus, the use case technique is employed to capture a system's behavioural requirements by detailing all the scenarios that users will be performing. The result of use case modelling should be that all required system functionality is described in the use cases.

More precisely, use cases describe the interaction between one or more actors and the system itself, represented as a sequence of simple steps. Actors are something or someone which exists outside the system under study, and that take part in a sequence of activities in a dialogue with the system to achieve some goal. Actors may be end-users, other systems, or hardware devices. Use cases typically avoid technical jargon, preferring instead the language of the end-user or domain expert. Use cases are often co-authored by systems analysts and end-users.

In use-case-based design [RS99], the use case model is at the conceptual center of the approach because it drives everything that follows, e. g., interaction/behaviour/state modelling as a refinement of the interaction/ the dynamic behaviour of the objects within the use case model, requirements tracing, deriving of test cases and so on.

3.2.2 Specification Techniques Comprised in the Paradigm

UML use case diagram [Bit02]. Use case diagrams are formally included in two modelling languages defined by the OMG. Both the UML [OMG09] and SysML [OMG08] standards define a graphical notation for modelling use cases with diagrams. These diagrams provide an graphical overview of the use cases for a given system and the relations between them.

In Figure 11 a sample use case diagram of an calculator is depicted. The basic elements of a use case diagram are as follows:

- *Use cases* represented by ovals, e. g., "Enter Number" or "Display Number".
- *Actors* represented by stick figures, e. g., "Engineer" or "Printer".
- *Associations between actors and use cases* indicated by solid lines. An association exists whenever an actor is involved with an interaction described by a use case.
- *System boundary boxes (optional)* to indicates the scope of your system. Anything within the box represents functionality that is in scope and anything outside the box is not.

Furthermore, different relationships between actors and between different use cases can be modelled in a use case diagram:

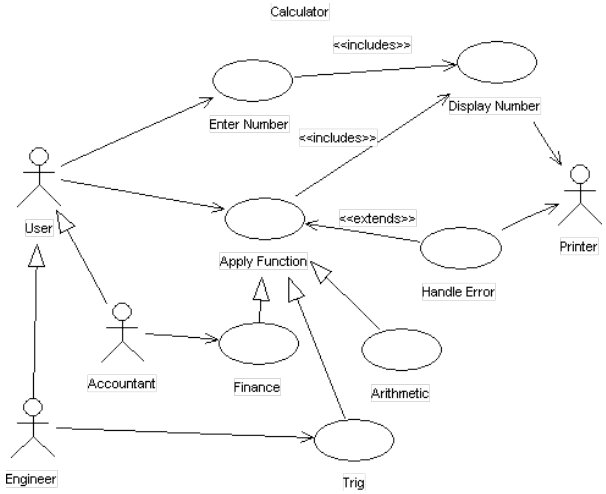


Figure 11: Sample Use Case Diagram [Pea09]

- *Actor generalisation* depicted by a solid line ending in a hollow triangle drawn from the specialised to the more general actor, e. g., "Engineer" and "Accountant" are both a (specialised) "User" of the calculator.
- *Include* is a directed relationship between two use cases, implying that the behaviour of the included use case is inserted into the behaviour of the including use case. This is useful for extracting truly common behaviours from multiple use cases into a single description. The notation is a dashed arrow from the including to the included use case, with the stereotype "include". In our example, the use case "Enter Number" includes the use case "Display Number".
- *Extend* is a directed relationship between two use cases that indicates that the behaviour of the extension use case may be inserted in the extended use case under some conditions. The notation is a dashed arrow from the extension to the extended use case, with the stereotype "extend". Notes or constraints may be associated with this relationship to illustrate the conditions under which this behaviour will be executed. In our example, the use case "Apply Function" might be extended by the use case "Handle Error".
- *Generalisation* is a relationship among use cases to extract common behaviours, constraints and assumptions to a general use case, describe them once, and deal with it in the same way, except for the details in the specialised cases. The notation is a solid line ending in a hollow triangle drawn from the specialised to the more general use case. In the example, "Arithmetic" is a specialisation of the use case "Apply Function".

Use case template. One complaint about the standards has been that they do not define a format for describing these use cases. Generally, both graphical notation and descriptions are important as they document the use case, showing the purpose for which an actor uses a system.

Thus, there is no standard template for documenting detailed use cases. A number of competing schemes exist, and individuals are encouraged to use templates that work for them or the

project they are on. There is, however, considerable agreement about the core sections, see for example [Coc00]:

- *Use case name.* as unique identifier for the use case. It should be written in verb-noun format, should describe an achievable goal, and should be sufficient for the end-user to understand what the use case is about.
- *Actors.* involved actors that either act on the system – a primary actors – or are acted on by the system – a secondary actors.
- *Preconditions.* conditions that must be true for the trigger (see below) to meaningfully cause the initiation of the use case.
- *Triggers.* events that causes the use case to be initiated.
- *Basic course of events:* typical course of events, also called basic flow, normal flow, or happy flow.
- *Alternative paths, extensions, exceptions.* secondary paths or alternative scenarios, which are variations on the main theme. Exceptions, or what happens when things go wrong at the system level, may also be described in a section of their own.
- *Postconditions.* The postconditions section describes what the change in state of the system will be after the use case completes.

3.2.3 Classification of the Specification Techniques

Covered system views. Use Cases are typically used to specify the functional view.

Underlying model of computation. Since use cases described the behavioural aspects of a system only informally by natural language, they cannot be classified according to the underlying model of computation.

Addressed stage of the development process. Use cases are typically used during the requirements engineering phase of the software development process. They aim at capturing and structuring the system's behavioural requirements from a black-box point of view.

Supported abstraction layers. Use cases are used in the functional view to elicitate, structure, and (textually) describe the system functionality from the user's point of view.

Available tool support. Modelling of use case diagrams is supported by every UML Case tool (e. g., Visio, ArgoUML, Enterprise Architect).

Compositionality. Since their lack a formal foundation, use cases do not offer a well-defined meaning of compositionality. However, use case diagrams are employed to provide a graphical overview of the different use cases of the system including relations between them.

Analysis techniques. Informal methods as inspections, reviews and walkthroughs can be applied to use case diagrams and use case descriptions for verifying and validating them. Formal analysis techniques and testing, simulation, and runtime verification can not be applied since use case specifications are are neither formally founded nor executable.

Typical notations. Use case diagrams are typically represented graphically (cf. Figure 11), the description of single use cases is normally given as structured text.

3.3 Automata

3.3.1 General Description of the Paradigm

A state machine or state automaton is a model of behaviour composed of a number of states, transitions between those states, and actions. The current state of a system is determined by the foregoing state and the received input of the system. A transition indicates a state change and is described by a condition that must be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment. Automata are usually used to formally define the operational semantics of a system.

As an example of the basic formalisation of automata, we describe in the following a Mealy machine. A Mealy machine is a finite state automaton that generates an output based on its current state and input. This means that the state diagram will include both an input and output signal for each transition edge (cf. Figure 12).

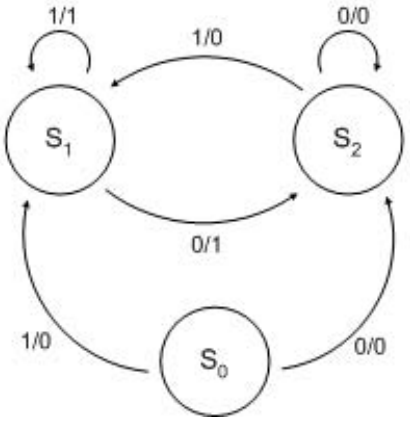


Figure 12: Example of Mealy Machine[Mea09]

Formally, a Mealy machine is a 6-tuple $(S, s_0, \Sigma, \Gamma, T, G)$, where

- S is a set of states;
- s_0 is a initial state with $s_0 \in S$;
- Σ is the input alphabet;
- Γ is the output alphabet;

- T is a transition function $T : S \times \Sigma \rightarrow S$ which maps a state and the input alphabet to the next state;
- G is an output function $G : S \times \Sigma \rightarrow \Gamma$ which maps each state and the input alphabet to the output alphabet.

3.3.2 Specification Techniques Comprised in the Paradigm

There are several kinds of state automata, e. g., Moore machines [Moo56], Mealy machines, Statecharts by Harel [Har87], I/O Automata by Lynch and Tuttle [LT89], Hybrid Automata by Henzinger [Hen96], Timed Automata by Alur [Alu99], Interface Automata by de Alfaro and Henzinger [dAH01], Timed I/O Automata by Kaynar et al. [KLSV03], Hybrid I/O Automata by Lynch et al. [LSV03], AutoFocus Automata by Schätz et al. [SPHP02], and many others.

3.3.3 Classification of the Specification Techniques

In the following automata are classified according to the criteria introduced in Section 2.

Covered system views. Automata are typically used to specify the behaviour and process views. They describe the dynamic aspects of the system.

Underlying model of computation. Below we detail the aspects of models of computation for automata (see also Table 2):

- *Time.* There are automata dialects supporting discrete (e. g., Statecharts or I/O Automata) as well as continuous time (e. g., Hybrid or Timed Automata).
- *Clock.* Local (e. g., Timed Automata) as well as global (e. g., I/O Automata) clocks are supported by automata.
- *Concurrency.* Automata can be executed simultaneously. There are synchronised (e. g., I/O Automata) and asynchronised (e. g., AutoFocus Automata) execution semantics of automata.
- *Behaviour.* Time-triggered (e. g., Timed Automata) as well as event-triggered (e. g., I/O Automata) behaviour is supported.
- *Flow.* Automata specify control flow.
- *Determinism.* Automata may be non-deterministic.
- *Data.* There are automata that can deal with continuous and infinite data (e. g., Hybrid Automata). However, most automata dialects are defined for discrete data.

Addressed stage of the development process. Automata are originally introduced to give meaning to computer programs in a mathematically rigorous way. Thus, they were initially used in the detailed design and implementation phases of the development process. However, over the years automata have been establishing in the system and software design and even in the requirements engineering phases, especially in the domain of reactive systems, where the relation between actions and reactions is in focus of consideration. This trend has been supported by the introduction of several modelling and analysis tools. For example, Hybrid Automata are used in the requirements engineering and system and software design phases to describe the whole system (i. e., software and physical components of the system) and its physical environment. In the detailed design and implementation phases discrete automata (e. g., I/O Automata) are used.

Supported abstraction layers. Discrete automata are basically used in the logical view, where the behaviour of components is described by automata. However, they also address the functional and technical views. Continuous automata (like Hybrid Automata) are often used in the functional view. There are approaches (i.e. [BH09]) which describe scenarios or functions by automata. Since all computer programs (written in a embedded-system typical language like C) are state machines, discrete automata can be used in the technical view, too.

Compositionality. All established automata-based approaches are compositional, i. e., the meaning of a combined automaton can be determined by the meanings of its constituent automata.

Available tool support. Modelling of statecharts is supported by every UML and further case tool (e. g., Visio, StarUML, Eclipse, AutoFocus, etc.). A lot of them support code generation. For analysis and simulation tools for automata see next paragraph and Appendix A.

Analysis techniques. Informal methods as inspections, reviews and walkthroughs can be applied to state machine specification for verification and validation. Analysis techniques as testing, simulation and runtime verification can be used to ensure that the state machine behaviour accords to the state diagram specification. Usually these techniques explore only a small number of cases in the state space. Formal verification techniques can guarantee a comprehensive check, but on the other hand they face the state explosion problem. In order to reduce this problem, especially in model checking, it is possible to translate a state machine into an abstract model that has the same properties as the original system. For instance an abstraction widely used in model checking is to encode the state machine model using a data structure called Reduced Ordered Binary Decision Diagrams (ROBDDs) [Bry86], that can be used to represent boolean functions and operations on such in an efficient manner.

In the appendix (A.6) is described a tool for runtime verification. The Assessment Studio module for ASCET provides model checking (A.4). Testing and dynamic simulation are supported by Rational Rhapsody and WinA&D (see Appendix A.2). We have also tool support for model checking which verifies specifications composed by a UML statechart diagram and other UML diagrams together: Papyrus UML, Hugo/RT, PROCO and Rhapsody in C++ (see Appendix A.4).

Typical notations. Automata are typically represented graphically (cf. Figure 12) or as tables (cf. Figure 6).

3.4 Control Flow Specification

3.4.1 General Description of the Paradigm

Control flow specifications describe the control flow of a business process, process or program by breaking a process down to a finite number of steps that get executed one at a time. The control flow governs how the next step to be executed is determined (step B after step A), but it does not say anything about what the inputs and outputs of the steps are, how the steps get performed internally, or why we might want to perform step A and then step B [Rid04].

In general, control flow diagrams show sequential steps, if-then-else conditions, repetition, and/or case conditions. Control flow specifications are usually graphically represented by control flow diagrams using suitably annotated geometrical figures to represent operations, data, or equipment, and arrows to indicate the sequential flow from one to another.

Control flow specifications can be modelled from the perspective of different user groups (such as managers, system analysts and clerks), leading to different classes of control flow diagrams, e. g.,

- document flowcharts, showing controls over a document-flow through a system,
- data flowcharts, showing controls over a data flows in a system,
- system flowcharts showing controls at a physical or resource level, or
- program flowcharts, showing the controls in a program within a system [Ste03].

Notice that every type of flowchart focuses on some kind of control, rather than on the particular flow itself.

3.4.2 Specification Techniques Comprised in the Paradigm

Flowcharts. A flowchart is a common type of diagram, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows.

A typical flowchart is depicted in Figure 13. Typical building blocks used in flowcharts are:

- *Start and end symbols* represented as circles, ovals or rounded rectangles, usually containing the word “Start” or “End”, or another phrase signaling the start or end of a process.
- *Arrows* showing the “flow of control”.
- *Processing steps* represented as rectangles.
- *Conditional or decision* represented as a diamond. Decision blocks typically contain a Yes/No question or True/False test. Decision blocks have (at least) two (labeled) arrows coming out of it, one corresponding to Yes or True, and one corresponding to No or False.

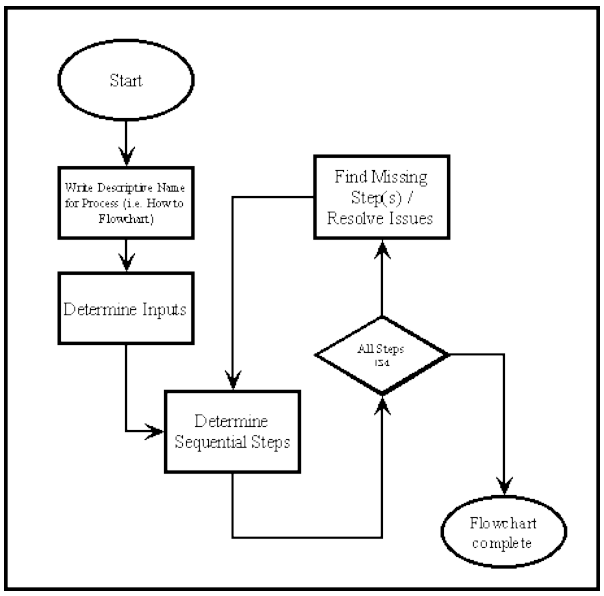


Figure 13: Sample Flowchart [Han09]

In some dialects, further symbols are available to denote e. g., input/output (represented as a parallelogram), manual operations (trapezoid with the longest parallel side at the top) or data files (cylinder).

Many diagram techniques exist that are similar to flowcharts and can be seen as a variation their of (e. g., UML activity diagrams).

UML Activity Diagrams. An UML activity diagram[OMG09], is a diagram to specify the dynamics of a system. In the UML, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system. UML 2 activity diagrams are typically used for business process modelling, for modelling the logic captured by a single use case or usage scenario, or for modelling the detailed logic of a business rule. It has constructs to express sequence, choice and parallelism. Its syntax is inspired by both Petri nets (see Section 3.7), statecharts (see Section 3.3) and flowcharts. In many ways UML activity diagrams are the object-oriented equivalent of flow charts and data flow diagrams (DFDs) (see Section 3.5)

An activity diagram consists of the following basic elements depicted in Figure 14 (cf. [Amb04]):

- *Initial activity.*
- *Final activity.*
- *Activity.*
- *Decisions.* Similar to flowcharts, a logic where a decision is to be made is depicted by a diamond, with the options written on either sides of the arrows emerging from the diamond.
- *Signal.* An activity can send or receives messages. These activities are called input and output signal, respectively.

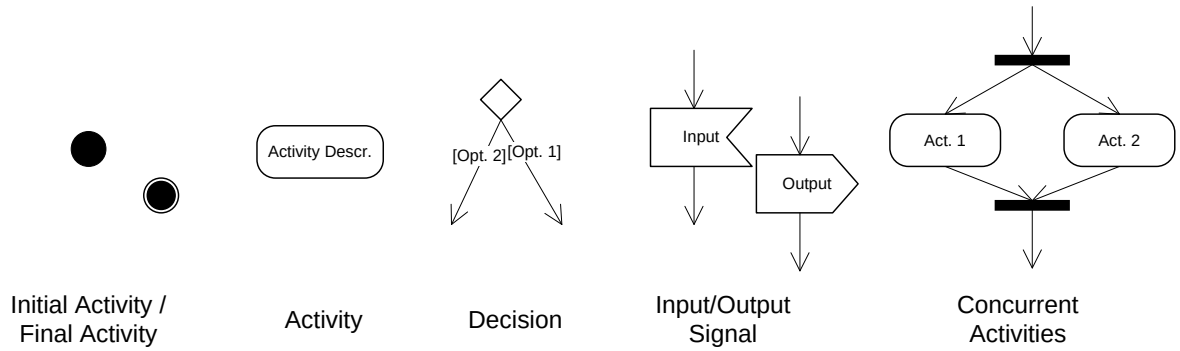


Figure 14: Basic Elements of an UML Activity Diagram

- *Concurrent activities.* are activities that occur simultaneously or in parallel. This is represented by a horizontal split (*Fork*) and the two concurrent activities next to each other, and the horizontal line again (*Join*) to show the end of the parallel activity.
- *Merge.* A diamond with several flows entering and one leaving. The implication is that one or more incoming flows must reach this point until processing continues, based on any guards on the outgoing flow.

There are further constructs to structure and hierarchically decompose activity diagrams:

- *Partition (Swimlanes).* Indicate who/what is performing the activities.
- *Sub-activity indicator.* A rake in the bottom corner of an activity indicates that the activity is described by a more finely detailed activity diagram.

UML activity diagrams are expressive enough to model all the constructs needed in workflow models. Unfortunately, the activity diagram semantics defined by the OMG is informal and ambiguous. However, there exist different approaches to assign a formal semantics to UML activity diagrams (e. g., in [EW01] based on statecharts or in [YzYzYf04] based on Petri nets).

Control flow graph. A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

In a control flow graph each node in the graph represents a basic block, i. e., a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

The CFG is essential to many compiler optimisations and static analysis tools, e. g., to analyse for reachability or infinite loops.

Further specification techniques. There exist further techniques for the control flow oriented specification of software systems, e. g., Nassi-Shneiderman [NS73] or Jackson diagrams [Jac75].

3.4.3 Classification of the Specification Techniques

In the following DFDs are classified according to the criteria introduced in Section 2.

Covered system views. Control flow specifications are primarily used to describe business processes, thus they address the process view. However, they can also be used to describe the behaviour of a system.

Underlying model of computation. Below we detail the aspects of models of computation for control flow specifications (see also Table 2):

- *Time.* Control flow specifications do not reason about time. Only a precedence relationship is defined between the but no information about the timing issues is modelled.
- *Clock.* There is no clock in control flow specification. Events are only ordered according to their precedence relation.
- *Concurrency.* In control flow specifications, e. g., in activity diagrams, concurrent activities can be explicitly modelled. For instance in activity diagrams, synchronisation between the concurrent activities is explicitly realised by the fork and join operator.
- *Behaviour.* Event-triggered behaviour is modelled.
- *Flow.* Obviously, control flow specifications specify control flow.
- *Determinism.* In general, control flow specifications might be non-deterministic.
- *Data.* The data aspect is not addressed by control flow specifications. However, in general, they are based on discrete data.

Addressed stage of the development process. Control flow specifications (esp. activity diagrams) address the early phases of the development process and are used for requirements modelling and in systems and software design. Control flow graphs support the design and analyses of algorithms and programs. Therefore, they are used in the detailed design, implementation as well as validation and verification phase.

Supported abstraction layers. Activity diagrams support the functional layer and help to identify the systems functionalities. Control flow specifications can be used in the specification of the behaviour in the logical layer, too.

Compositionality. Structuring mechanisms to hierarchically decompose a model are indispensable in order to deal with complex real-world systems. In general, control flow oriented specifcation like activity diagrams do not offer such hierarchy mechanisms. Thus, they must not be regarded to be compositional.

Available tool support. There is a broad range of tools available for modelling control floe specifications. Almost every UML-tool (e. g., Visio, Rational Rhapsody, Enterprise Architect, SmartDraw,...) supports activity diagrams.

Analysis techniques. Informal methods as inspections, reviews and walkthroughs can be applied to control flow specifications for verification and validation. Different static analysis tools perform various analyses on control flow graphs e. g., analyses for reachability or infinite loops. Testing, simulation, and model checking is supported too, e. g., by Rational Rhapsody, Toolkit for Conceptual Modelling.

Typical notations. Control flow specifications are graphically represented.

3.5 Data Flow Specification

3.5.1 General Description of the Paradigm

In the late 1970s data-flow diagrams (DFDs) were introduced and popularised for structured analysis and design [GS77, DeM79]. DFDs show the flow of data through an information system – from external entities into the system, how the data moved from one internal process to another, as well as its logical storage.

Note, that a DFD provides no information about the timing or ordering of processes, or about whether processes will operate in sequence or in parallel. It is therefore quite different from a flowchart, which shows the flow of control through an algorithm, allowing a reader to determine what operations will be performed, in what order, and under what circumstances, but not what kinds of data will be input to and output from the system, nor where the data will come from and go to, nor where the data will be stored (all of which are shown on a DFD).

3.5.2 Specification Techniques Comprised in the Paradigm

Basic Data Flow Diagrams. Basic data flow diagrams show

- the processes within the system,
- the data stores (files) supporting the system's operation,
- the information flows within the system,
- the system boundary, and
- interactions with external entities.

In Figure 15 a typical graphical notation of the basic elements of DFDs is depicted. DFDs

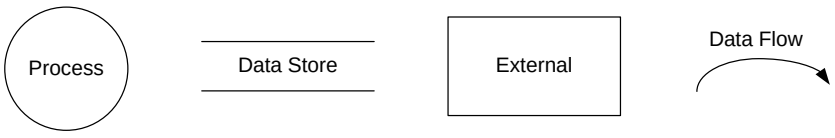


Figure 15: Data Flow Diagram – Notation

can be hierarchical organised in a top-level diagram, the *context diagram*, (Level 0) underlain by cascading lower level diagrams (Level 1, Level 2,...) that represent different parts of the system. The context diagram only contains one process node that generalises the function

of the entire system in relationship to external entities. The first level DFD shows the main processes within the system. Each of these processes can be broken into further processes.

Data Flow Diagrams for real-time systems. In [WM86, You89] extensions of classic DFDs for real-time systems are introduced. In addition to the previously introduced concepts, we need a way of modelling

- control flows (i. e., signals or interrupts),
- control processes (i. e., bubbles whose only job is to coordinate and synchronise the activities of other bubbles in the DFD), and
- buffers

These elements are shown graphically with dashed lines on the DFD, as illustrated in Figure 16.

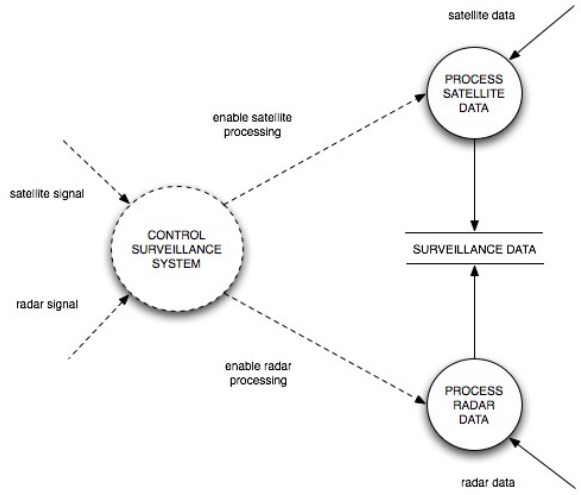


Figure 16: Example of a Real-Time DFD [You09]

3.5.3 Classification of the Specification Techniques

In the following DFDs are classified according to the criteria introduced in Section 2.

Covered system views. DFDs are used to model the behaviour of a system by showing the flow of data through the system.

Underlying model of computation. Below we detail the aspects of models of computation for DFDs (see also Table 2):

- *Time and Clock.* As mentioned before, Basic DFDs provide no information about the timing or ordering of processes, or about whether processes will operate in sequence or in parallel. Their extension for real-time systems, however, introduce control processes that explicitly coordinate other possibly (concurrent) processes. Their are different implementations of DFDs, like synchronous dataflow models as implemented in Matlab Simulink, Lustre/Esterel [HCRP91], or Kahn process networks [Kah74] with a precise semantics. In general, they assume discrete time. For instance, Simulink uses an idealised timing model for block execution and communication. Both happen infinitely fast at exact points in simulated time. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps [MM07].
- *Concurrency.* In DFDs communication between concurrent processes is explicitly modelled by message exchangw. Synchronous DFDs assume synchronous message passing, while in other dialects like Kahn process networks communication is realised by asynchronous message passing via infinitely large FIFOs.
- *Behaviour.* DFDs are event-triggered.
- *Flow.* DFDs specify the flow of data through the system. Extended DFD dialects (e. g., Real-Time DFDs), allow to model the control flow in addition to the pure data view.
- *Determinism.* Kahn process networks, for example, are deterministic. In general, however, control flow specifications might be non-deterministic.
- *Data.* Basic DFDs are defined for discrete data, but their exist extensions that can deal with continuous data.

Addressed stage of the development process. DFDs were originally introduced in the context of structured analysis and design method (SADM) [GS77, DeM79], a software engineering methodology for describing systems as a hierarchy of functions. They are one of the three essential perspectives of the SSADM.

Consequently, data flow diagrams are used in the requirements engineering and the design phases. With a data-flow diagram, users are able to visualise how the system will operate, what the system will accomplish, and how the system will be implemented.

Supported abstraction layers. Since SADM aims at describing systems as a hierarchy of functions, DFDs clearly address the functional view.

Compositionality. DFDs offer mechanisms to hierarchically describe a system by a top-level diagram underlain by cascading lower level diagrams that detail different parts of the system. Thus they can be considered to be compositional.

Available tool support. There exist many commercial tools for drawing basic DFDs (e. g., Visio). For the real-time extensions there might exist some research tools. However, real-time DFDs are not supported by the commercial tools as Vision. Simulink and SCADE offer toolchains for the modelling of data flow diagrams as well as simulation, formal verification, and code generation.

Analysis techniques. Inspections, reviews and walkthroughs can be applied for checking errors in the data flow diagrams. We can specify test cases for checking the conformance of the diagram to its specification. There are two tools which provide testing techniques: Rational Rhapsody and WinA&D (see Appendix A.2).

Typical notations. Typically, DFDs are represented graphically, as depicted in Figures 15 and 16.

3.6 Function Block Specification

3.6.1 General Description of the Paradigm

A function block diagram is a diagram of a system, in which the principal parts or functions are represented by blocks connected by lines, that show the relationships of the blocks. The blocks portray mathematical or logical operations that occur in time sequence. They do not represent the physical entities, such as processors or relays, that perform those operations.

3.6.2 Specification Techniques Comprised in the Paradigm

Function Block Diagram (FBD). A function block diagram (FBD) is a diagram, that describes a function between input variables and output variables as a set of interconnected blocks (see Figure 17).

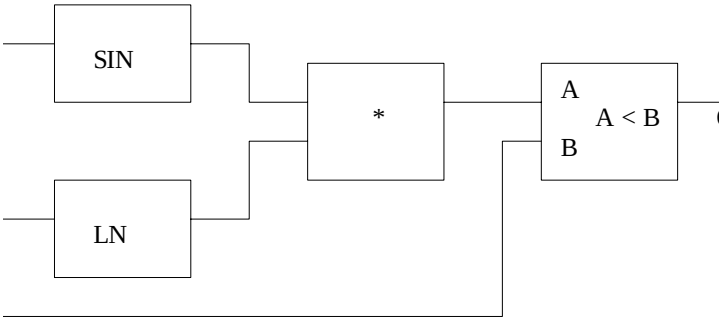


Figure 17: Sample Function Block Diagram [Jac05]

Directed lines are used to connect input variables to function inputs, function outputs to output variables, and function outputs to inputs of other functions. The connection is oriented, meaning that the line carries associated data from the left end to the right end. The left and right ends of the connection line must be of the same type. Multiple right connection, also called divergence can be used to broadcast information from its left end to each of its right ends.

Function block diagram is one of five languages for logic or control configuration supported by standard IEC 61131-3 for a control system such as a Programmable Logic Controller (PLC, Speicherprogrammierbare Steuerung SPS).

Function Flow Block Diagram (FFBD). (Synonyms: *Functional Flow Diagrams*, *functional block diagrams*, *functional flows*)

The FFBD notation was developed in the 1950s, and is widely used in classical systems engineering [NAS95, Lon95].

An FFBD shows the functions that a system is to perform and the order in which they are to be enabled (and performed). Each function (represented by a block) occurs following the preceding function. Some functions may be performed in parallel, or alternate paths may be taken. The duration of the function and the time between functions is not shown. The FFBD does not contain any information relating to the flow of data between functions, and therefore does not represent any data triggering of functions. The FFBD only presents the control sequencing for the functions.

An sample FFBD is given in Figure 18.

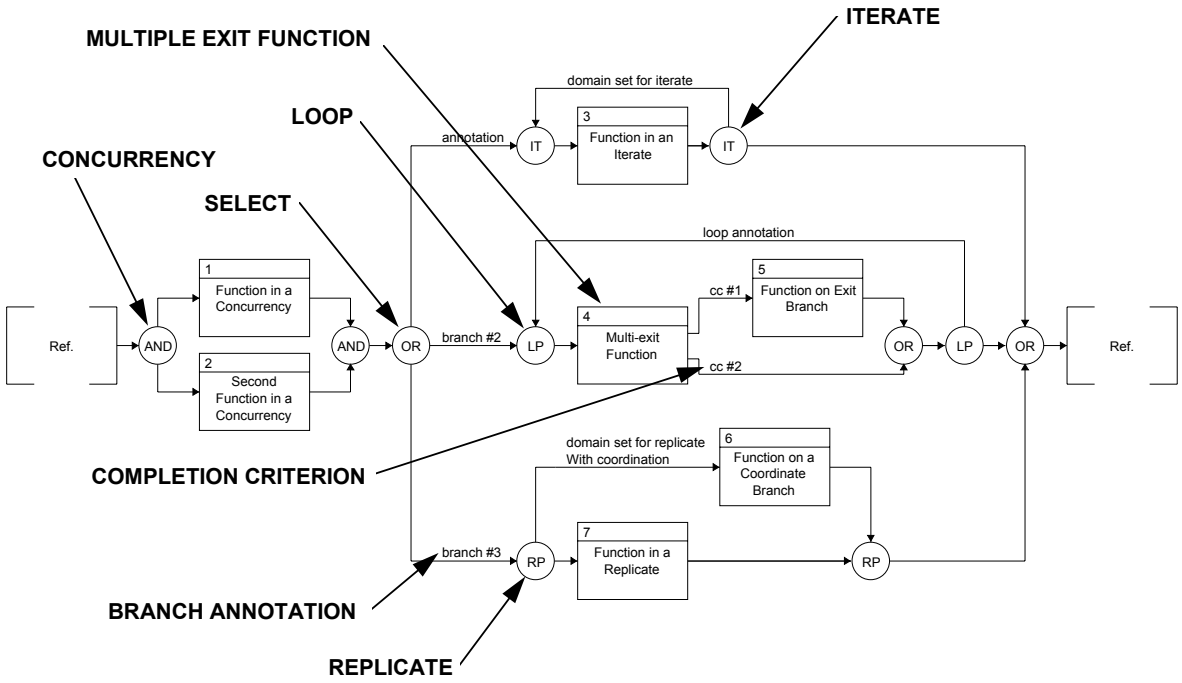


Figure 18: Sample FFBD [Lon95]

FFBDs can be developed in a series of levels - reflecting the functional decomposition of the system. Each block in a top level diagram can then be expanded to a series of functions in the next level diagram.

Enhanced Function Flow Block Diagram (EFFBD). The EFFBD combines the control dimension of the functional model in an FFBD format with data flow aspects to effectively capture data dependencies [Lon95].Thus, the Enhanced FFBD represents:

1. functions,

2. control flows, and
3. data flows.

The logic constructs allow you to indicate the control structure and sequencing relationships of all functions accomplished by the system being analysed and specified. When displaying the data flow as an overlay on the control flow, the EFFBD graphically distinguishes between triggering and non-triggering data inputs. Triggering data is required before a function can begin execution. Therefore, triggers are actually data items with control implications. In Figure 19, triggers are shown with green backgrounds and with the double-headed arrows. Non-triggering data inputs are shown with gray backgrounds and with single-headed arrows.

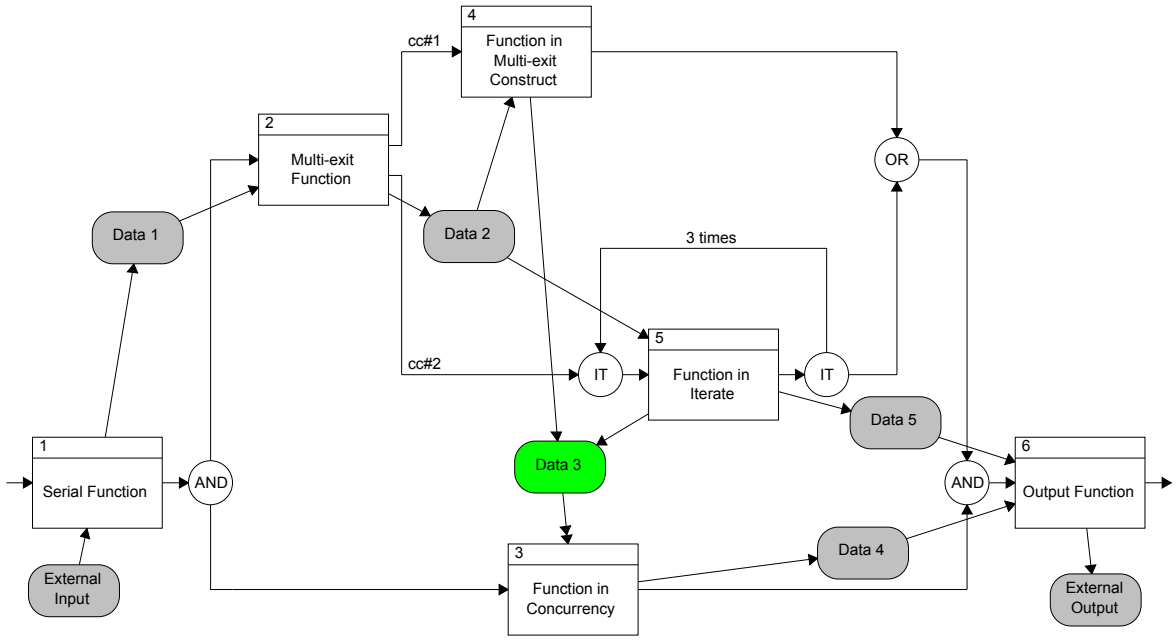


Figure 19: Sample EFFBD [Lon95]

The Enhanced FFBD specification of a system is complete enough that it is executable as a discrete event model, providing the capability of dynamic, as well as static, validation.

3.6.3 Classification of the Specification Techniques

In the following FBDs are classified according to the criteria introduced in Section 2.

Covered system views. Functional block diagrams are used to model the behaviour of a system by showing the system's functions and their connections.

Underlying model of computation. Below we detail the aspects of models of computation for FBDs (see also Table 2):

- *Time, Clock, and Concurrency.* In general, FBDs provide no information about the execution time of the function blocks.
- *Behaviour.* Function flow diagrams are event-triggered.
- *Flow.* FBDs and FFBDs specify data flow, EFFBDs additionally specify control flow.
- *Determinism.* In general, control flow specifications might be non-deterministic.
- *Data.* In general, function block specifications can be specified for discrete as well as continuous data.

Addressed stage of the development process. Block diagram are typically used for a higher level description aimed at understanding the overall concepts. To this end, they are used during the requirements engineering and early design phases. In the detailed design phase, FBDs and (E)FFBDs are used for a detailed specification of the system's behaviour.

Supported abstraction layers. High-level FBDs are primarily used on the functional layer, more detailed FBDs and EFFBDs are also used on the logical layer.

Compositionality. Analogously to control flow specifications (see cf. Section 3.4), FBDs do not offer a hierarchy mechanism and consequently are not compositional.

Available tool support. A multitude of tools support FBDs (e. g., Visio). Besides, there exists a Siemens proprietary tool chain including editors, analysis functionality and code generation for PLCs.

Analysis techniques. Inspections, reviews and walkthroughs can be applied to perform informal verification to function block diagrams. It is possible to apply formal verification translating the function block in an input format for model checkers, as well as testing and simulation. ASCET tool supports testing and simulation; SCADE has a Simulator module for executing simulations and Simulink generates test cases for their function block specifications (see Appendix A.2). SCADE and Simulink support also formal verification by the Prover Plug-In (see Appendix A.3). It is also reported two model checking tools for function block diagram: Prise and the Assessment Studio module for ASCET (see Appendix A.4).

Typical Notation Block Diagrams are typically represented graphically. A typical example for a function block diagram is depicted in Figure 17.

3.7 Petri Nets

3.7.1 General Description of the Paradigm

Petri Nets is a formal and graphical language for modelling systems with concurrency and resource sharing. Petri Nets have been under development since the beginning of the 60s, where Carl Adam Petri defined the language. It was the first time a general theory for discrete parallel systems was formulated. The language is a generalisation of automata theory such that the concept of concurrently occurring events can be expressed.

A Petri Net consists of two types of nodes: places (pictured as circles) and transitions (usually pictured as bars). Arcs can exist only between places and transitions. In each place can exist several tokens (illustrated through dots). With respect to a transition, a place is called an "input place" if it is before the transition, or vice versa an "output place". A transition is ready to fire (i. e., activated) iff there is at least one token at each one of its input places.

In Figure 20 we illustrate an example of a Petri Net that models two processes: a sender and a receiver. The sender and receiver synchronise themselves by having the same transition that will be enabled only when both input places have tokens.

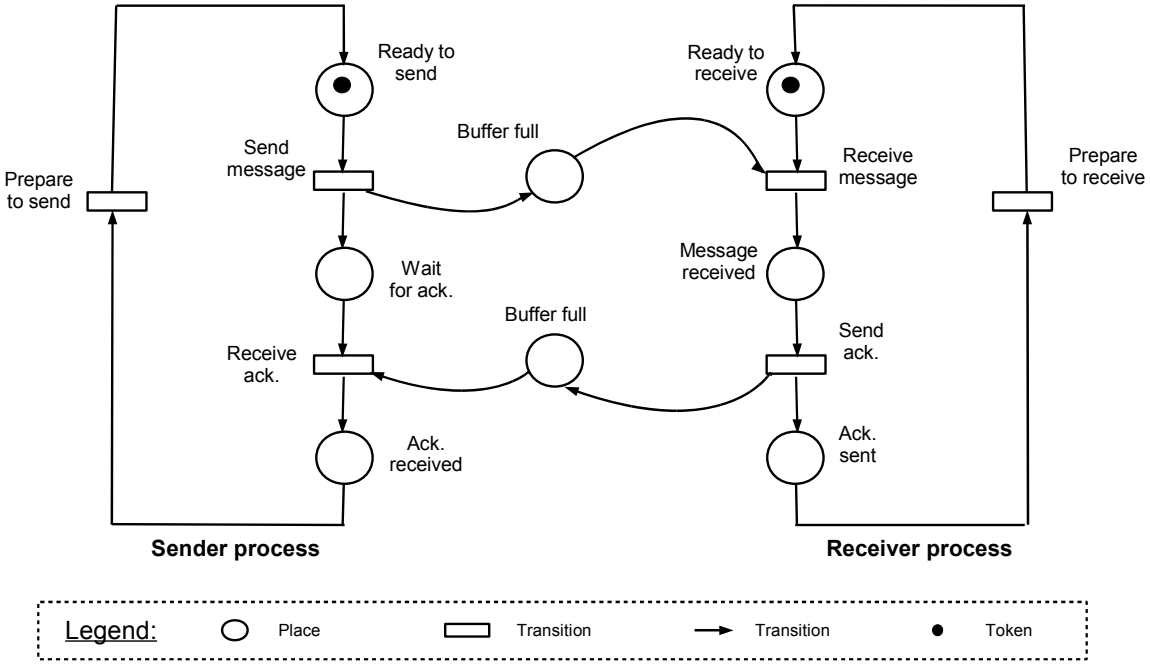


Figure 20: Petri Nets example for modelling Sender/Receiver (adapted from [Mur89])

Formally, a Petri Net is a 5-tuple (P, T, F, W, M_0), where

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places;
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation);
- $W : F \to \{1, 2, 3, \ldots\}$ is a weight function;

- $M_0 : P \rightarrow \{0, 1, 2, 3, \dots\}$ is the initial marking;
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

Petri nets have been used to model concurrent processes and distributed systems. Various kinds of Petri net classes with numerous features and analysis methods have been proposed in literature (among many others [Bra80], [BRR87a], [BRR87b]) for different purposes and application areas. The fact that Petri nets are widely used and are still considered to be an important topic in research, shows the usefulness and the power of this formalism. Petri nets have been used to model various kinds of dynamic event-driven systems like computers networks [Liu98], communication systems [Wan07], manufacturing plants [ZD89], command and control systems [AL88], real-time computing systems [TYC95], logistic networks [LB02], and workflows [LIRM02] to mention only a few important examples.

Interpretations of Petri Nets. Petri Nets are a generic modelling method since input places, output places, and transitions can be interpreted in different manners as shown below [Mur89]:

- input places can be seen as preconditions, transitions as events, and output places as postconditions. In this case, Petri Nets are like state automata.
- input places can be seen as input data, transitions as computation steps, and output places as output data. In this case, Petri Nets are like data flow languages.

3.7.2 Specification Techniques Comprised in the Paradigm

Standard Petri Nets do not have time and transitions are taken immediately after their activation. There are several timed extensions of the standard Petri Nets [Mur89] by considering time delays associated with places or transitions.

An ordinary Petri net has only one kind of tokens and no modules. Coloured Petri Nets define use data types and complex data manipulation [Jen81]. Each token has attached a data value called the token colour, which can be investigated and modified by the occurring transitions. Besides a large model can be obtained by combining a set of submodels.

3.7.3 Classification of the Specification Techniques

In the following Petri Nets are classified according to the criteria introduced in Section 2.

Covered system views. Petri Nets are typically used to model the behaviour of a system. Analysis of the Petri net can then, reveal important information about the structure and dynamic behaviour of the modelled system. This information can then be used to evaluate the modelled system and suggest improvements or changes.

Underlying model of computation. Below we detail the aspects of models of computation for Petri Nets (see also Table 2):

- *Time.* Petri Nets support the discrete time (e. g., Timed Petri Nets).
- *Clock.* Local as well as global (e. g., Timed Petri Nets) clocks are supported by Petri Nets.
- *Concurrency.* Petri nets are asynchronous, concurrent models. Events can happen at any time and there exists a partial order of events.
- *Behaviour.* Time- (e. g., Timed Petri Nets) as well as event-triggered behaviour is supported.
- *Flow.* Petri Nets specify a data flow.
- *Determinism.* Petri Nets may be non-deterministic.
- *Data.* Petri Nets can model continuous and discrete data.

Addressed stage of the development process. Petri Nets are typically used for the specification of requirements with focus on sequencing of activities, parallelism, concurrency, and branching. Since Petri Nets are typically represented through diagrams, they can be simulated and therefore are useful for requirements validation. Petri Nets can be used also for system design, for detailed design, or for system verification.

Supported abstraction layers. Petri Nets are a graphical language that is usually used in the logical view. In fact, Petri nets are intuitive and can be used to describe the behaviour of components. They were devised for use in the modelling of a specific class of problems, the class of discrete-event systems with concurrent or parallel events.

Compositionality. Modelling large real-world systems with Petri nets is feasible when using structuring mechanisms as the hierarchical modelling. In this model it is possible to inspect the modelled system at varying levels of detail, to visualise selected parts of the system, and to facilitate the multiple (re-)use of parts of the model. There are many extensions to Petri nets, one important extension is hierarchy: approaches as Coloured Petri Nets [Jen81] and hierarchical Petri net [Feh93] provide the semantic for combining a set of subcomponent Petri nets in an hierarchical model.

Available tool support. There is a common infrastructure the Petri Net Kernel used to construct Petri Net tools, which supports all kind of Petri Nets, provides basic modification and save operations and a graphical user interface. CPN Tools [JKW07][10] is a tool for editing, simulating and analysing untimed and timed, hierarchical Coloured Petri nets. It is freely available and maintained by the CPN Group at the University of Aarhus in Denmark. Most of the Petri Nets tool can be found at the so-called Petri Nets Tool Database[11]. Unfortunately, many of the tools described on the database or in literature are not longer maintained or available. Model checkers for Petri Nets are described in next paragraph and Appendix A.4

[10] http://wiki.daimi.au.dk/cpntools/cpntools.wiki
[11] http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html

Analysis techniques. Inspections, reviews and walkthroughs can be applied to Petri nets for informal verification and validation. Due to the well founded mathematics of Petri nets, automatic verification of models is possible. The techniques developed in this area range from simple deadlock checks to sophisticated model checking tools. We can apply testing, formal verification and runtime verification. In the appendix (A.4) is described a tool for PERS+ which supports model checking.

Typical notations. Petri Nets are typically represented graphically (Figure 20).

3.8 Sequence Diagrams

3.8.1 General Description of the Paradigm

Sequence diagrams designates a graphical and textual language for the description and specification of the interactions between system components, in particular as an overview specification of the communication behaviour of systems, be they, e. g., real-time or telecommunication switching systems. More precisely, a sequence diagram is a scenario description of the interaction –prominently communication, among other events– between a number of message-passing instances and their environment. Additionally, the language allows for expressing restrictions on transmitted data values and on the timing of events. Sequence diagrams may be used for requirement specification, simulation and validation, test-case specification and documentation of systems.

3.8.2 Specification Techniques comprised in the Paradigm

Message Sequence Charts (MSC). The first specification technique representative of this paradigm is MSC (Message Sequence Charts) released in 1992, that emerged from SDL (Specification and Description Language). The MSC standard was added references, ordering and inlining expressions concepts, and introduced HMSC (High-level Message Sequence Charts) in 1996; this latter construct allows the expression of state diagrams. The latest MSC version of 2000 added object orientation, refined the use of data and time in diagrams, and added the concept of remote method calls; see [Hau01, Hau05].

Live Sequence Charts (LSC). A reported weakness of MSC is the fundamental meaning of a scenario: in early stages of development, scenarios constitute just sample examples of system behaviour, whereas in later stages and if the condition at the beginning of the chart (if any) becomes true, then the system has to behave as described by the scenario. That is, the initially existential interpretation becomes universal, and only the latter interpretation allows for the expression of liveness properties. This motivated the enhancement to LSC (Live Sequence Charts). An LSC diagram is in general constituted by a pre-chart and a chart. The live interpretation of an LSC diagram requires that the behaviour specified by the chart *must* be exhibited by a system whenever the system has shown the behaviour specified by the pre-chart. Live elements, called *hot* (indicating that progress is enforced), make it possible to define forbidden scenarios. Furthermore, also mandatory and possible conditions, invariants,

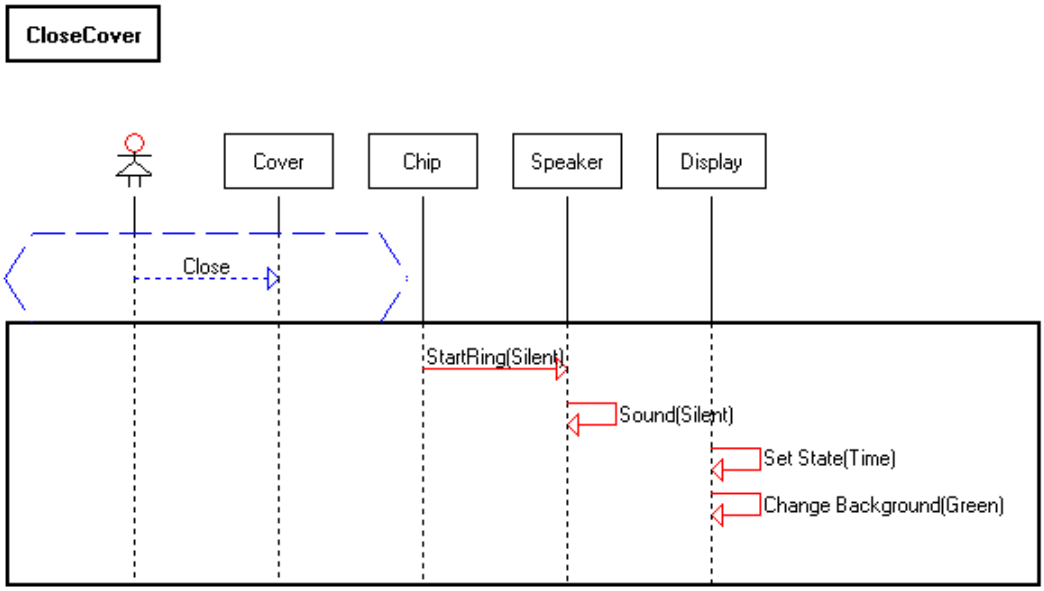


Figure 21: Sample LSC: universal chart [HMS08]

simultaneous regions and coregions, activation and quantification, may be specified; see [DH01]. Examples of LSC diagrams are shown in Figures 21 and 22.

Formally, let $\Theta = \{hot, cold\}$. An LSC body is a tuple

$$((\mathcal{L}, \preceq), I, \sim, B, V, X, Msg, Cond, LocInv, Z, Tmr)$$

where

- $(\mathcal{L}, \preceq)$ is a finite, non-empty, partially ordered set of locations, each $l \in \mathcal{L}$ associated with timing interval $T(l) \subseteq Q_0^+ \dot{\cup} \{\infty\}$, temperature $\theta(l) \in \Theta$, and with one of the finitely many instance lines $i_l \in I$,
- $\sim \subseteq \mathcal{L} \times \mathcal{L}$ is an equivalence relation on locations, the simultaneity relation,
- B is an alphabet, V a set of variables, X a set of clocks,
- $Msg \subseteq \mathcal{L} \times B \times \mathcal{L}$ is a set of instantaneous messages with $(l, b, l') \in Msg$ only if $l \sim l'$,
- $Cond \subseteq (2^{\mathcal{L}} \setminus \emptyset) \times \Phi(X, V) \times \Theta$ is a set of conditions with $(\mathcal{L}, \varphi, \theta) \in Cond$ only if $l \sim l'$ for all $l, l' \in \mathcal{L}$,
- $LocInv \subseteq \mathcal{L} \times \{\circ, \bullet\} \times \Phi(X, V) \times \Theta \times \mathcal{L} \times \{\circ, \bullet\}$ is a set of local invariants,
- $Tmr \subseteq \mathcal{L} \times Z \times (Q_0^+ \dot{\cup} \{reset, timeout\})$ is a set of timers with $(l, z, d) \in Tmr$ and $d \notin Q_0^+$ implies that there exist unique $l' \preceq l, d' \in Q_0^+$ such that $(l', z, d') \in Tmr$.

For each $l \in \mathcal{L}$, if l is the location of a condition, a local invariant, or a timer set or reset, then there exists l' equivalent to l which is the location of a message or an instance head. An LSC is a tuple $L = (b, ac, pch, amode, quant)$ with b the body of the LSC, ac the activation condition, pch the pre-chart, $amode \in \{initial, invariant, iterative\}$ the activation mode, and

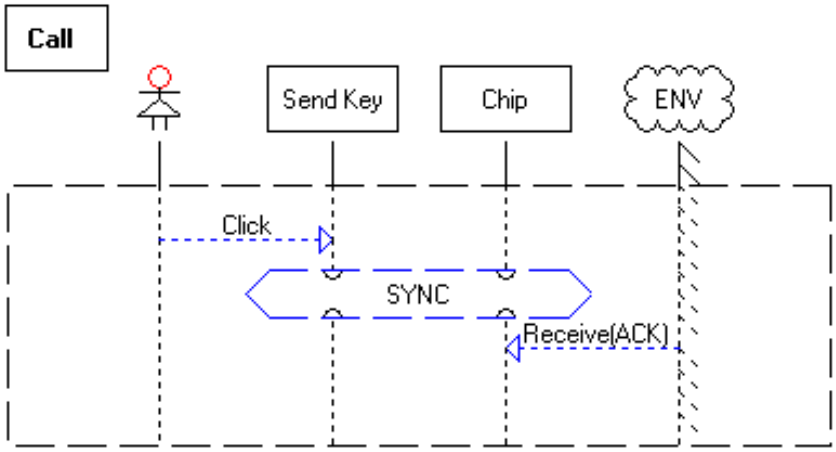


Figure 22: Sample LSC: existential chart [HMS08]

$quant \in \{existential, universal\}$ the quantification. The LSC formal semantics is based on the concept of timed Büchi automata; see [BDK+04].

UML Sequence Diagrams. UML interactions describe inter-object communication by message exchange patterns. Interactions can be expressed by means of sequence diagrams, communication diagrams (formerly called collaboration diagrams), interaction overview diagrams, or timing diagrams; see [OMG09]. These languages are comparable with respect to their expressive power; see [CG09]. Moreover, UML sequence diagrams and MSC have influenced each other and there is no noteworthy difference between them; see [Hau05]. Examples of UML sequence diagrams are depicted in Figures 23 and 24.

The abstract syntax of UML sequence diagrams is as follows:

$$
\begin{array}{rcl}
\mathit{Basic} & ::= & \mathsf{skip} \mid \mathbb{E} \mid \mathbb{A} \\
\mathit{SDiagram} & ::= & \mathit{Basic} \\
 & \mid & \mathit{CombinedFragment} \\
\mathit{CombinedFragment} & ::= & \mathsf{strict}(\mathit{SDiagram}, \mathit{SDiagram}) \\
 & \mid & \mathsf{seq}(\mathit{SDiagram}, \mathit{SDiagram}) \\
 & \mid & \mathsf{par}(\mathit{SDiagram}, \mathit{SDiagram}) \\
 & \mid & \mathsf{loop}(\mathbb{N}, (\mathbb{N} \cup \infty), \mathit{SDiagram}) \\
 & \mid & \mathsf{ignore}(\mathbb{M}, \mathit{SDiagram}) \\
 & \mid & \mathsf{alt}(\mathit{SDiagram}, \mathit{SDiagram}) \\
 & \mid & \mathsf{neg}(\mathit{SDiagram}) \\
 & \mid & \mathsf{assert}(\mathit{SDiagram}) \\
 & \mid & \mathsf{constraint}(\mathit{Term}, \mathit{SDiagram})
\end{array}
$$

where, for given sets of lifelines and $\mathbb{M}$ of messages, $\mathbb{E}$ is a set of send and receive events, $\mathbb{A}$ is a set of arrows (i. e., of synchronous send and receive events), $\mathbb{N}$ denotes the set of natural

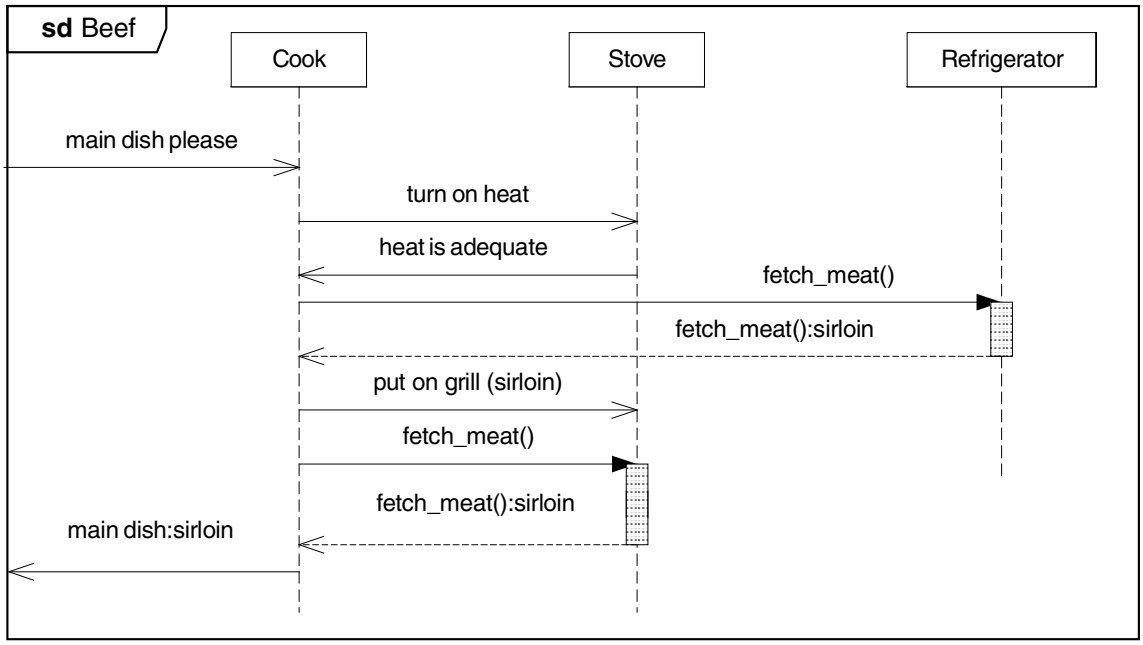


Figure 23: Sample basic sequence diagram [HS03]

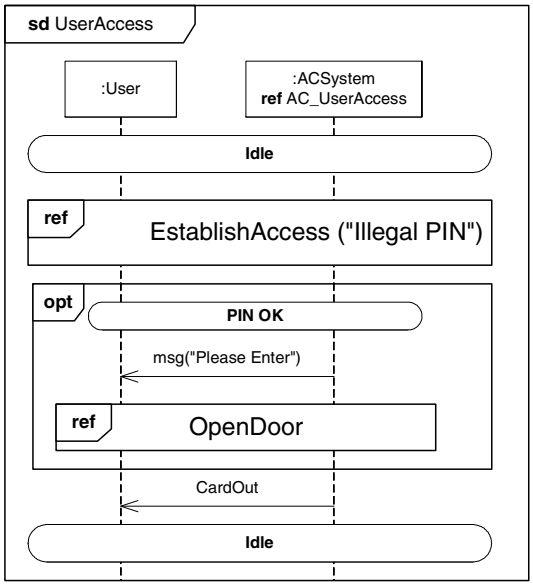


Figure 24: Sample sequence diagram: combined fragment [Hau05]

numbers, and $\overline{n}$ denotes a member of $\mathbb{N} \cup \{\infty\}$. The semantics of UML sequence diagrams is trace based and not necessarily a trace is either positive or negative for a given UML sequence diagram. Indeed, a trace may be inconclusive (see [KRHS05]), and moreover a trace may be both positive *and* negative for a UML sequence diagram; see also [CK04].

3.8.3 Classification of the Specification Techniques

Covered system views. Sequence diagrams are designed for the description of the interaction among sub-systems, that is, they are typically used for the specification of the behaviour view of a system.

Underlying model of computation. Below we detail the aspects of models of computation for sequence diagrams (see also Table 2):

- *Time and Clock.* In sequence diagrams, the time is event discrete and there are multiple clocks, whereby the precedence ordering among events is partial.
- *Concurrency.* Sequence diagrams describe concurrent systems with no synchronisation taking place besides maybe through communication, which typically is asynchronous (although mechanisms for the expression of synchronous message passing are put at disposal), non-deterministic, explicit, and it is abstracted away from dynamic vs. static communication.
- *Behaviour and Flow.* The behaviour of sequence diagrams is event triggered.
- *Flow.* The flow of computation is through data being exchanged among participant subsystems.
- *Determinism.* Sequence diagrams are non-deterministic.
- *Data.* They do not address data (i. e., no mechanism for data description is offered by the paradigm).

Addressed stage of the development process. Sequence diagrams are mainly used at early stages of development, i. e., for requirements elicitation and engineering as well as analysis. They can also be used during design, although allegedly the LSC specification technique appears to be more adequate for design than MSC and UML sequence diagrams.

Supported abstraction layers. Sequence diagrams are primarily used for the specification of the functional and logical views on a system.

Compositionality. There is no concept of operators on LSCs, i. e., two LSCs cannot be combined by means of, e. g., a union or sum operator. Thus, besides "putting scenarios together" there is no way to compose LSCs, and consequently the matter of compositionality does not apply.

On the contrary, MSCs and UML Sequence Diagrams can be composed as well as equipped with a compositional semantics.

Available tool support. UML sequence diagrams are supported by, e. g., Rational Rhapsody, ArgoUML, UMLet. LSC diagrams can be created using the Play Engine [HM03].

Analysis techniques. Reviews, inspections and walkthroughs can be used for the informal validation and verification of sequence diagrams. Sequence diagrams can be animated: TURTLE [ACLSS04] for UML sequence diagrams, and the Play Engine [HM03] for LSC diagrams.

Typical notations. Almost exclusively, sequence diagrams are graphically represented. Other representations via, e. g., words of an abstract syntax or XML encoding, can be used for the purposes of verification and/or exchange between tools.

3.9 Contraint-based Specification

3.9.1 General Description of the Paradigm

The constraint programming paradigm allows the declaration of relations between variables in form of constraints. Contrarily to imperative programming, constraints do not specify sequences of steps to execute, they rather specify properties of a solution, i. e., constraint programming is a declarative paradigm. Usually, constraint solvers are embedded in a programming language, which typically is but is not circumscribed to a logic programming language. The host language can also be for instance functional, imperative, and even term rewriting.

Constraints can be from different domains. Some popular ones are:

- boolean domains, e. g., $\varphi \vee \psi$ is true
- integer domains, rational domains
- linear domains, e. g., $x \leq y$, where only linear functions are described and analysed
- finite domains, where constraints are defined over finite sets
- mixed domains, involving two or more of the above

Finite domains is one of the most successful domains of constraint programming. Constraint solvers are often realised as a separate library in an imperative programming language.

3.9.2 Specification Techniques Comprised in the Paradigm

The above domains can be compiled and handled by use of the Constraint Handling Rules (CHR [Frü09]). CHR is a declarative programming language extension originally designed for developing (prototypes of) constraint programming systems. Although CHR is Turing complete, it was not devised as a programming language in its own right but rather to extend a host language with constraints. These host languages include Prolog, Java and Haskell, among others.

CHR is nowadays increasingly used as a high-level general-purpose programming language. Typical application domains of CHR are multi-agent systems, natural language processing,

compilation, scheduling, spatial-temporal reasoning, testing and verification, and type systems.

A CHR program, sometimes called a constraint handler, is a sequence of guarded rules for simplification, propagation, and "simpagation" (a mix of simplification and propagation) of conjunctions of constraints. The CHR constraint store is a multi-set. In contrast to Prolog, the rules are multi-headed and are executed in a committed-choice manner.

The Object Constraint Language (OCL [KW99]) provides a specification language for the definition of constraints and well-formedness requirements, like invariants and pre- and post-conditions, for models of the Unified Modeling Language (UML [OMG09]). OCL comprises a navigational expression language which can be used to compute object values in a UML model. For pre- and post-conditions associated with methods, which are akin to contracts and assumption/guarantee specifications, there exists the pre- and post-condition times as well as a time "in between" as provided by the action clause (see [KW02]).

3.9.3 Classification of the Specification Techniques

The classification is done over the domains of application of the constraints.

Covered system views. For *any* system view an adequate domain can be identified in which constraints of interest are expressible.

Underlying model of computation.

- *Time and Clock.* Depending on the domain, time might be relevant and one or more clocks might be present.
- *Concurrency.* Constraints are concurrent in character, since *all* of them have to be satisfied.
- *Flow.* Constraint solvers can be both event and time triggered.
- *Determinism.* They are typically non-deterministic.
- *Data.* Depending on the addressed domain the data they handle can be any of finite, discrete and even continuous nature.

Addressed stage of the development process. Constraints are better suited for (detailed) design, implementation and verification.

Supported abstraction layers. Constraints are suited for any abstraction layer.

Compositionality. CHR and OCL are compositional approaches.

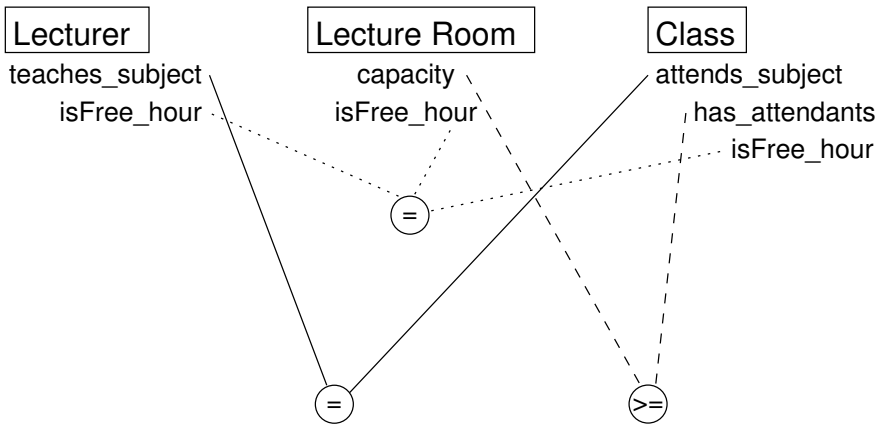


Figure 25: Constraints: graphical notation

Available tool support. CHR libraries are implemented for Prolog, Java, Haskell, Curry. Several implementations for OCL have been provided, among them the Bremen USE tool [RG01], the Dresden OCL tool [DW09], the KeY tool [ABHS07], and the pUML Meta-Modelling Framework (MMF [CEK02]).

Analysis techniques. There is not much literature on how constraint handlers are analysed. Obviously inspections, reviews and walkthroughs are suitable for the task. Certainly testing is also applied.

Typical notations. Constraints are usually expressed as a logic formula or a multi-headed Prolog clause. For instance,

$$(\forall \overline{x})(\exists \overline{y}) P_1 \wedge \cdots \wedge P_m \rightarrow Q_1 \vee \cdots \vee Q_n$$

where $\overline{x}$ are the variables occurring free in $Q_1, \ldots, Q_n$ and $\overline{y}$ are the other variables occurring free in $P_1, \ldots, P_m$, is classically denoted by

```
Q1;...;Qn :- P1,...,Pm.
```

Also a graphical notation can be used as exemplary depicted in Figure 25; this notation soon hinders more than eases the understanding of constraints, when many individual constraints are involved.

3.10 Algebraic Specification

3.10.1 General Description of the Paradigm

Algebraic specifications are a generalisation of the abstract data type concept. They concentrate on the functionality of a system and provide a systematic means for describing the properties of the system. Basically, a specification is composed of a signature and restrictions (also called axioms) associated with it. A signature consists of name declarations for data sets

and data operations, and possibly also for data relations. The associated restrictions are formulae over the signature; depending on the formalism chosen, they can be, e. g., equations or first-order formulae with equality. These restrictions must be fulfilled by any implementation of the specification.

Not one but a class of algebras or structures is denoted by an algebraic specification; the term structure is used when the signature is allowed to declare data relations. Moreover, the choice of those structures can be done by more than one criterion. The *initial* approach takes the initial semantics of a specification whose restrictions are expressed by equations; see [GTW78]. The *loose* approach assigns a class of (non-isomorphic) algebras to a specification whose restrictions are first-order formulae; see [SW83]. Of course the restrictions can be expressed in any logic as, e. g., temporal logic; see [FL90].

For larger systems, specifications may become quite unmanageable: specification-building operators are usually put at designer's disposal. These allow the construction of new, larger specifications (also called structured specifications) out of smaller ones. Moreover, new specification-building operators can be defined by constructs like parameterisation and lambda abstraction; see [Wir90, SST92, Cen94, Dim98].

In any case, algebraic specifications, be simple ones or structured (i. e., obtained by use of specification-building operators), imply further properties of the system-to-be other than the ones expressed by their axioms, and are pairwise related by a so-called refinement relation. Typically, refinement is semantically equivalent to model class inclusion. A formalisation of the kind offered by algebraic specifications should be accompanied by means for performing proofs and deriving the refinement relation; see [BCH99].

3.10.2 Specification Techniques Comprised in the Paradigm

Algebraic specification present programs as algebras consisting of datatypes and operations (signature). The intended behaviour of a program is specified by means of formulas (say, equations) concerning these operations (restrictions/axioms). There are several kinds of algebraic specifications comprising different algebras and different formalisms for introducing restrictions. One of the most prominent representative is the Common Algebraic Specification Language (CASL). CASL is a general-purpose specification language based on first-order logic with induction. Partial functions and subsorting are also supported. Basic specifications in CASL denote classes of partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed. Axioms are first-order formulae built from definedness assertions and both strong and existential equations. Datatype declarations are provided for concise specification of sorts equipped with some constructors and (optional) selectors, including enumerations and products. Later, we present two interesting CASL's extensions: HOL-CASL and CASL-LTL.

3.10.3 Classification of the Specification Techniques

In the following, algebraic specifications, namely HOL-CASL and CASL-LTL, are classified according to the criteria introduced in Section 2.

Covered system views. All algebraic specifications cover data system view through signatures which represent the data structure of the system and their interdependancies. Some of algebraic specifications also support structure system view, e. g., HOL-CASL and CASL-LTL. A structured specification in CASL is formed by combining specifications in various ways, starting from basic specifications. The structure of a specification is not reflected in its models: it is used only to present the specification in a modular style. Through restrictions, algebraic specification define a desired behaviour of a modelled system, i. e., they also support behaviour system view.

Underlying model of computation. The criterion is relevant only for axioms or restrictions, since signatures describe static aspects of a system. Thus, the classification depends on the concrete formalism chosen for introducing restrictions. In the case of HOL-CASL, this is high-order logic. Below we detail the aspects of models of computation for HOL (see also Table 3):

- *Time.* No time.
- *Clock.* No Clock.
- *Concurrency.* No notion to introduce concurrency.
- *Behaviour.* No notion to express time-triggered (periodic) behavoiur.
- *Flow.* Not covered.
- *Determinism.* Not covered.
- *Data.* Depends of a chosen signature.

In the case of CASL-LTL, the formalism used to define restrictions is Linear Temporal Logic. Below we detail the aspects of models of computation for LTL (see also Table 3):

- *Time.* No metric time. Only precedence relationship is defined.
- *Clock.* No Clock.
- *Concurrency.* No notion to introduce concurrency.
- *Behaviour.* No notion to express time-triggered (periodic) behavoiur.
- *Flow.* Not covered.
- *Determinism.* Not covered.
- *Data.* Depends on a chosen signature.

Addressed stage of the development process. Algebraic specifications are typically used for specifying requirements and design of conventional software packages, i. e., the addressed software development process stage are Requirements engineering and System and software design.

Supported abstraction layers. The functional view is supported because the restrictions allow specifying functional properties of a system. The logical view is also supported since the data structures can be specified for components first and combined further in various ways thus building a structure.

Compositionality. HOL-CASL and CALS-LTL are compositional since they support specification-building operators (structured specifications) which allow the construction of larger specifications out of smaller ones.

Available tool support. Axioms for HOL-CASL can be written in, e. g., Isabelle/HOL. Axioms for LTL-CASL can be written in any tool supporting LTL, e. g., Maude LTL model checker [EMS03].

Analysis techniques. Inspections, reviews and walkthroughs can be applied to algebraic specification for performing informal verifications. We can apply formal verification by static analysis techniques, for checking the syntactic and semantic correctness. In high-order logic representation, i. e., HOL-CASL, it is possible to verify correctness with a theorem prover, e. g., Isabelle. The Heterogeneous Tool Set supports this latter techniques for the CASL language; see Sect. A.5 below. There exist multiple tools (so called model checkers) supporting verification of temporal logic formula, among others LTL, e.g. Maude LTL model checker [EMS03].

Typical notations. Typical notation for algebraic specification is textual.

3.11 Process Algebra

3.11.1 General Description of the Paradigm

Process algebra denotes any approach devised to formally modelling concurrent systems that, given a set of names, allow the definition of terms out of these names by means of a number of operators, typically

- parallel composition,
- sequential composition,
- data communication,
- hiding or scope restriction, and
- recursion or process replication.

Names stand for (basic) processes or channels, whose purpose is to provide means of communication. Channels may have rich internal structure, in particular some implementations take advantage of an internal structure in order to improve efficiency, but this is abstracted away in most theoretic models.

Process algebras are calculi, that is, they also provide algebraic laws that allow process descriptions to be manipulated and analysed, and permit formal reasoning about equivalences between processes (e. g., using bisimulation [Par81, Mil89]).

More formally, a process algebra consists of a set of operators and syntactic rules for constructing process terms, a semantic mapping assigning meaning to every process term, and a notion of equivalence and/or partial order between processes. Equality for the process algebra is also a congruence relation and thus permits the substitution of one component with another equal

component in large systems. Therefore correctness can be modularly proved since, moreover, a large system can be partitioned into simpler sub-systems. A hiding or restriction operator allows one to abstract away unnecessary details.

3.11.2 Specification Techniques comprised in the Paradigm

The first sustainable approach was the Calculus of Communicating Systems (CCS, [Mil80]), followed by the Communicating Sequential Processes (CSP, [Hoa85]) and by the Algebra of Communicating Processes (ACP, [BK85]), which coined the term process algebra.

Calculus of Communicating Systems Let τ denote the *silent* action and $\mathcal{N}$ be a set of process names. Let A be a set of action names and $\overline{A} = \{\overline{a} : a \in A\}$ be the set of co-names; $A \cup \overline{A}$ represent the externally *visible* actions. Let $\mathcal{A}$ denote the set $A \cup \overline{A} \cup \{\tau\}$. The set of CCS processes is defined by the following BNF grammar:

$$P \ ::= \ \emptyset \ | \ \alpha.P_1 \ | \ N \ | \ P_1 + P_2 \ | \ P_1|P_2 \ | \ P_1[b/a] \ | \ P_1\backslash a$$

where $\alpha \in \mathcal{A}$, $a \in \mathcal{A} \setminus \{\tau\}$, and $N \in \mathcal{N}$.

The informal semantics of the expressions generated by the above semantics can be summarised as follows:

empty process the empty process $\emptyset$ is a valid CCS process

action the process $a.P_1$ can perform an action a and continue as the process P_1

process identifier write $N \stackrel{\text{def}}{=} P_1$ to use the identifier N to refer to the process P_1

choice the process $P_1 + P_2$ can proceed either as the process P_1 or the process P_2

parallel composition processes P_1 and P_2 exist simultaneously

renaming $P_1[b/a]$ is the process P_1 with all actions named a renamed as b

restriction $P_1\backslash a$ is the process P_1 without action a

The expressions of the language are interpreted as a labelled transition system. Between these models, bisimilarity is used as a semantic equivalence.

The operational semantics defines a relation with judgements of the form $P \stackrel{\alpha}{\rightarrow} P'$ defined by

the following axiom and inference rules:

$$\alpha.P \xrightarrow{\alpha} P$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'} \qquad \frac{\begin{array}{c} P \xrightarrow{a} P' \\ Q \xrightarrow{\overline{a}} Q' \end{array}}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{b} P'}{P\backslash a \xrightarrow{b} P'\backslash a}\ b \notin \{a, \overline{a}\}$$

$$\frac{P \xrightarrow{a} P'}{P[b/a] \xrightarrow{b} P'[b/a]} \qquad \frac{P \xrightarrow{c} P'}{P[b/a] \xrightarrow{c} P'[b/a]}\ c \neq a$$

The reflexive and transitive closure of the above relation, denoted by $P \xrightarrow{\sigma} Q$ and defined by $P \xrightarrow{\epsilon} P$ plus $P \xrightarrow{a \cdot \sigma} Q$ if there exists P' with $P \xrightarrow{a} P'$ and $P' \xrightarrow{\sigma} Q$, is the basis on which a *trace semantics* of CCS is defined. Trace equivalence is termed *observational* if occurrences of the silent action are removed from the trace.

CCS is useful for evaluating the qualitative correctness of properties of a system such as deadlock or livelock. The expressions of the language are interpreted as a labelled transition system, as induced by the inference system of above. Between these models, bisimilarity is used as a semantic equivalence. Specification languages and formalisms based on CCS include the π-calculus [MPW92a, MPW92b], which provides mobility of communication links by allowing processes to communicate the names of communication channels themselves, the Performance Evaluation Process Algebra (PEPA, [GH94]), which introduces activity timing and probabilistic choice thus allowing performance metrics to be evaluated, the Calculus of Broadcasting Systems (CBS, [Pra95]), where handshake communication is replaced by broadcast communication which implies different observationally meaningful laws, and the Language Of Temporal Ordering Specification (LOTOS, [EDV89]), a formal specification language based on temporal ordering used for protocol specification in ISO OSI standards.

Communicating Sequential Processes CSP was influential in the development of the occam programming language [RH88], and has been practically applied in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems, such as a secure ecommerce system. The theory of CSP itself is also the subject of active research, including work to increase its range of practical applicability (e. g., increasing the scale of the systems that can be tractably analyzed). CSP allows the description of systems in terms of component processes that operate independently, and interact with each other solely through

message-passing communication. The formal syntax of CSP-terms in its core is as follows:

P	$::= SKIP$	successful terminating process
	$\mid \quad STOP$	deadlock process
	$\mid \quad \alpha \rightarrow P_1$	prefixing
	$\mid \quad P_1; P_2$	sequential composition
	$\mid \quad \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2$	boolean conditional
	$\mid \quad P_1 \Box P_2$	external choice
	$\mid \quad P_1 \sqcap P_2$	non-deterministic choice
	$\mid \quad P_1 \mid\mid\mid P_2$	interleaving
	$\mid \quad P_1 \mid[X]\mid P_2$	interface parallel
	$\mid \quad P_1 \backslash X$	hiding
	$\mid \quad P_1 \triangle P_2$	P_1 interrupted by P_2
	$\mid \quad \mu X.F(X)$	process X s.t. $X = F(X)$

where α can be, among others, a variable assignment of the form $x := e$ for x a variable and e an expression, an output of the form $c!e$ meaning "on channel c output the value of e," or an input of the form $c?x$ menaning "on variable x store the value input on channel c." Modern CSP allows those component processes to be sequential as well as obtained by parallel composition of more primitive ones. The theory of CSP includes mutually consistent denotational semantics, algebraic semantics, and operational semantics. The three major denotational models of CSP are the traces model, the stable failures model, and the failures/divergences model. The traces model defines the meaning of a process expression as the set of sequences of events (traces) that the process can be observed to perform.

Specification languages and formalisms based on the classic untimed CSP include Timed CSP [SDJ+92], which incorporates timing information for reasoning about real-time systems, Receptive Process Theory [Jos92], a specialisation of CSP that assumes an asynchronous (i. e., nonblocking) send operation, CSPP [Law01], an extension of CSP that includes priority and addresses systems which are massively parallel, widely distributed, implemented in either hardware or software or both, HCSP [Law02], a superset of CSPP which captures the semantics of hardware compilation and can thus describe both hardware and software and so is useful for co-design, Wright [All97], an architecture description language based on the formal description of the abstract behaviour of architectural components and connectors that provides a practical formal basis for the description of both architectural configurations and of architectural styles, TCOZ [MD98], an integration of Timed CSP and Object Z, Circus [WC02], an integration of CSP and Z based on the Unifying Theories of Programming, and CSPC-ASL [Rog06], an extension of CASL that integrates CSP.

Algebra of Communicating Processes ACP was initially developed as part of an effort to investigate the solutions of unguarded recursive equations. More so than CCS and CSP, the development of ACP focused on the algebra of processes, and sought to create an abstract, generalised axiomatic system for processes. ACP is fundamentally an algebra, in the sense of universal algebra, which provides a way to describe systems in terms of algebraic process expressions that define compositions of other processes, or of certain primitive elements. ACP fundamentally adopts an axiomatic, algebraic approach to the formal definition of its various

operators. A process algebra over a set of atomic actions A is a structure

$$\mathcal{A} = \langle \mathsf{A}, \delta, (\partial_H)_{H \subseteq A}, +, \cdot, \lfloor\!\lfloor \rangle$$

where A is a set containing A, δ (deadlock) is a constant in A, ∂_H (encapsulation) is a unary operator on A, and $+$ (alternative composition), $\cdot$ (sequential composition), and $\lfloor\!\lfloor$ (left merge) are binary operators on A, such that for all $x, y, z \in \mathsf{A}$, $H \subseteq A$, and $a \in A$ the following axioms are satisfied:

$$\begin{aligned}
x + y &= y + x && \text{(A1)}\\
x + (y + z) &= (x + y) + z && \text{(A2)}\\
x + x &= x && \text{(A3)}\\
x \cdot (y \cdot z) &= (x \cdot y) \cdot z && \text{(A4)}\\
(x + y) \cdot z &= x \cdot z + y \cdot z && \text{(A5)}\\
\\
x + \delta &= x && \text{(A6)}\\
\delta \cdot x &= \delta && \text{(A7)}\\
\\
a \mathbin{\lfloor\!\lfloor} x &= a \cdot x && \text{(M1)}\\
a \cdot x \mathbin{\lfloor\!\lfloor} y &= a \cdot (x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x) && \text{(M2)}\\
(x + y) \mathbin{\lfloor\!\lfloor} z &= x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z && \text{(M3)}\\
\\
\partial_H(a) &= a \text{ if } a \notin H && \text{(D1)}\\
\partial_H(a) &= \delta \text{ if } a \in H && \text{(D2)}\\
\partial_H(x + y) &= \partial_H(x) + \partial_H(y) && \text{(D3)}\\
\partial_H(x \cdot y) &= \partial_H(x) \cdot \partial_H(y) && \text{(D4)}
\end{aligned}$$

Parallel composition is a derived operator defined by $x \| y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x$ and thus the axiom (M2) can be rewritten as $a \cdot x \mathbin{\lfloor\!\lfloor} y = a \cdot (x \| y)$. A description of this algebra can be found in [BK84], that includes comparisons to CCS.

ACP has served as the basis for some other formalisms that can be used to describe and analyse concurrent systems, including: the Process Specification Formalism (PSF, [MV90]), suitable for the specification of all kinds of processes based on ACP and Algebraic Specification of data (ASF), μCRL [GP95], especially developed to take account of data in the analysis of communicating processes and basically intended to study description and analysis techniques for (large) distributed systems, mCRL2 [GMR+07], a specification language for describing concurrent discrete event systems, accompanied with a toolset suitable for simulation, analysis and visualisation of behaviour, which is based on ACP whereas data is based on abstract equational data types extended with higher-order functions, and HyPA [CR05], a process algebra for hybrid systems that extends ACP, includes the disrupt operator from LOTOS as well as flow clauses and re-initialisation clauses for the description of continuous behaviour and discontinuities. A brief summary of the history of process algebras can be found in [Bae05].

3.11.3 Classification of the Specification Techniques

Covered system views. Process algebras are typically designed for the description of the communication between sub-systems and thus usually used for the specification of both the

behaviour view and the process view of a system. Some dialects as, e. g., Wright may besides allow the specification of the structural view. Some dialects as, e. g., PSF and μCRL may also permit the specification of the data view.

Underlying model of computation. Below we detail the aspects of models of computation for sequence diagrams (see also Table 3):

- *Time and Clock.* Process algebra terms behave event discrete with multiple clocks, thus the precedence ordering between events is partial.
- *Concurrency.* Process algebras address concurrent systems with handshaking communication.
- *Behaviour.* The behaviour of terms in a process algebra is usually event triggered.
- *Flow.* The flow of computation denoted by a term of a pure process algebra is given by data exchanged between participant processes or sent to the environment.
- *Determinism.* Terms of any process algebra typically denote non-deterministic computation(s).
- *Data.* In pure form, a process algebra does not take care of data.

Addressed stage of the development process. Process algebras can be used both at early stages of development and, more suitably, for verification purposes.

Supported abstraction layers. Process algebras are primarily used for the specification of the functional and logical views on a system.

Compositionality. All the above approaches support stepwise modelling and are moreover provided with compositional semantics.

Available tool support. As pointed out above, there is an overpopulation of variants and dialects of process algebras, many of them accompanied by tool support with diverse foci.

Analysis techniques. Inspections, reviews and walkthroughs techniques can be applied to the process algebra specifications. Also formal verification as well as testing and simulation methods have been used. This implentations have been constructed on different process algebra dialects considering their specific characteristics. In appendix are described the mCRL2 toolset A.2 which implements model checking, theorem proving and simulation and the model checker FDR (Failures-Divergence Refinement) A.4. The CSP-Prover [IR05] is an interactive theorem prover dedicated to refinement proofs within the process algebra CSP, that specifically aims at proofs on infinite state systems, which may also involve infinite non-determinism. The CSP-Prover focuses on the stable failures model as the underlying denotational semantics of CSP.

Typical notations. Process algebra terms are typically expressed as words of a language presented using a BNF as done above for CCS.

4 Conclusion

Model-based development assumes the use of adequate models in each development phase. In practice, the models are built with the help of modelling tools that provide their users a multitude of concepts, analysis functionality or model editors. Behind the scene, modelling tools use modelling languages whose concepts, well hidden by the tool, can be difficult to grasp by end-users if they are visible at all.

By taking a closer look at modelling languages, we notice that they are built from a handful of specification techniques. Being aware of this fact helps us to gain a broader view over different modelling languages, about their interoperability and powerfulness. Each specification technique is based on a certain set of concepts that defines its capabilities and limitations. Depending on the supported specification technique, a modelling language can be adequate or not to model a certain situation, to support a certain kind of analyses, or to enable the (partial) interoperability of models developed in different modelling languages.

In this document we presented a catalogue of criteria for the classification of specification techniques. We focused on practical aspects like: the system view typically covered by the technique, the underlying model of computation assumed by the technique, the typical development process stage in which a technique is used, the abstraction layer where the technique can be used, tool support for analyzing the models, analyses supported by the specification techniques, and typical notations used in practice to build the models. We used our criteria to classify different specification techniques. Our longer term goal is to develop a comprehensive set of criteria that is capable to realise a unifying view over different specification techniques and their dialects.

References

[ABHS07] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. KeY: A Formal Method for Object-Oriented Systems. In M.M. Bonsangue and E.B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems (9th FMOODS, Proceedings)*, volume 4468 of *Lecture Notes in Computer Science*. Springer, 2007.

[ACLSS04] Ludovic Apvrille, Jean-Pierre Courtiat, Christophe Lohr, and Pierre De Saqui-Sannes. TURTLE: A real-time UML profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering*, 30(7):473–487, 2004.

[AL88] Stamos K. Andreadakis and Alexander H. Levis. Synthesis of Distributed Command and Control for the Outer Air Battle. In *Symposium on C^2 Research (Proceedings)*. Science Applications International Corporation (SAIC), 1988.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[Alu99] Rajeev Alur. Timed Automata. In Nicolas Halbwachs and Doron Peled, editors, *11th International Conference on Computer Aided Verification (CAV'99, Proceedings)*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999.

[Amb04] Scott W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, March 2004.

[AR79] I. I. Amitan and I. V. Romanovskii. Signal language to describe the interaction of parallel processes. *Cybernetics and Systems Analysis*, 15(1):82–89, 1979.

[Bae05] Jos C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2–3):131–146, 2005.

[BBK+04] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *6th International Conference on Formal Engineering Methods (ICFEM'04, Proceedings)*, volume 3308 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2004. http://www.pst.ifi.lmu.de/veroeffentlichungen/balser-et-al:icfem:2004.pdf(26/04/10).

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The Synchronous Languages 12 Years Later. In *Proceedings of the IEEE*, volume 91, pages 64–83, 2003.

[BCH99] Michel Bidoit, María Victoria Cengarle, and Rolf Hennicker. Proof systems for structured specifications and their refinements. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*. Springer, 1999.

[BDK+04] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Live Sequence Charts: An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification. In Hartmut Ehrig, Werner Damm,

Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 374–399. Springer, 2004.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04, Proceedings)*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer, September 2004.

[Bel04] Donald Bell. UML Basics: The component diagram. Technical report, IBM Corporation, 2004. http://www.ibm.com/developerworks/rational/library/dec04/bell/(26/04/10).

[BH09] Jewgenij Botaschanjan and Alexander Harhurin. Property-Driven Scenario Integration. In *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, November 2009.

[BHSV+96] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *8th International Conference on Computer Aided Verification (CAV'96, Proceedings)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.

[Bit02] Kurt Bittner. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, 1984.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[Bra80] Wilfried Brauer, editor. *Net Theory and Applications, Proceedings of the Advanced Course on General Net Theory of Processes and Systems*, volume 84 of *Lecture Notes in Computer Science*. Springer, 1980.

[Bro95] Manfred Broy. Mathematics of Software Engineering. In Bernhard Möller, editor, *Mathematics of Program Construction (MPC'95, Proceedings)*, volume 947 of *Lecture Notes in Computer Science*, pages 18–48. Springer, 1995.

[Bro07] Manfred Broy. Two Sides of Structuring Multi-Functional Software Systems: Function Hierarchy and Component Architecture. In *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA'07, Proceedings)*, pages 3–12. IEEE Computer Society, 2007.

[BRR87a] Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors. *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I (Pro-*

ceedings of an Advanced Course), volume 254 of *Lecture Notes in Computer Science*. Springer, 1987.

[BRR87b] Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors. *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I (Proceedings of an Advanced Course)*, volume 255 of *Lecture Notes in Computer Science*. Springer, 1987.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *International Conference on Computer-Aided Verification (CAV'02, Proceedings)*, volume 2404 of *Lecture Notes in Computer Science*. Springer, July 2002.

[CEK02] Tony Clark, Andy Evans, and Stuart Kent. Engineering Modelling Languages: A Precise Meta-Modelling Approach. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering (FASE'02, Proceedings)*, volume 2306 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2002.

[Cen94] María Victoria Cengarle. *Formal Specifications with Higher-Order Parameterization*. PhD thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 1994.

[CEP00] Luis Alejandro Cortés, Petru Eles, and Zebo Peng. Verification of Embedded Systems using a Petri Net based Representation. In *13th International Symposium on System Synthesis (ISSS'00, Proceedings)*, pages 149–156. IEEE Computer Society, 2000.

[CG09] María Victoria Cengarle and Hans Grönniger. System Model Semantics of Interactions. Technical Report TUM-I0932, Institut für Informatik, Technische Universität München, 2009.

[CK04] María Victoria Cengarle and Alexander Knapp. UML 2.0 Interactions: Semantics and Refinement. In Jan Jürjens, Eduardo B. Fernandez, Robert France, and Bernhard Rumpe, editors, *3rd International Workshop on Critical Systems Development with UML (CSDUML'04, Proceedings)*, pages 85–99. Technical Report TUM-I0415, Institut für Informatik, Technische Universität München, 2004.

[CLB03] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, 2003.

[CM04] Séverine Colin and Leonardo Mariani. Run-Time Verification. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Cor96] Steve Cornett. Code coverage analysis. Technical report, Bullseye Testing Technology, 1996. http://www.bullseye.com/coverage.html.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: a declarative language for real-time programming. In *14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87, Proceedings)*, pages 178–188, New York, 1987. ACM.

[CR05] Pieter J. L. Cuijpers and Michel A. Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191–245, 2005.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[DeM79] Tom DeMarco. Structured analysis and system specification. pages 409–424, 1979.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[Dim98] Theodosis Dimitrakos. Parameterising (Algebraic) Specifications on Diagrams. In *13th IEEE International Conference on Automated Software Engineering (ASE'98, Proceedings)*, pages 221–224, Washington, DC, USA, 1998. IEEE Computer Society.

[DVM+05] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. Boosting Re-use of Embedded Automotive Applications Through Rich Components. In *Foundations of Interface Technologies (FIT'05, Proceedings)*, 2005.

[DW09] Birgit Demuth and Claas Wilke. Model and Object Verification by Using Dresden OCL. In *Russian-German Workshop on Innovation Information Technologies: Theory and Practice (Proceedings)*, page 81, 2009.

[DY95] Conrado Daws and Sergio Yovine. Two examples of verification of multirate timed automata with Kronos. In *IEEE Real-Time Systems Symposium (RTSS'95, Proceedings)*, pages 66–75. IEEE Computer Society Press, 1995.

[EDV89] Peter H. J. van Eijk, Michel Diaz, and Chris A. Vissers, editors. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science, 1989.

[EMS03] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In Thomas Ball and Sriram K. Rajamani, editors, *10th International Conference on Model Checking Software (SPIN'03, Proceedings)*, volume 2648 of *Lecture Notes in Computer Science*, pages 230–234. Springer, 2003.

[Est05] The Esterel v7 Reference Manual. Technical report, Esterel Technologies, 679 av. Dr. J. Lefebvre, 06270 Villeneuve-Loubet, France, November 2005. http://www.esterel-technologies.com/files/Esterel-Language-v7-Ref-Man.pdf(13/04/10).

[EW01] Rik Eshuis and Roel Wieringa. A Formal Semantics for UML Activity Diagrams – Formalising Workflow Models. CTIT technical reports series 01-04, University of Twente, 2001. http://doc.utwente.nl/37504/(26/04/10).

[EW04] Rik Eshuis and Roel Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

[Fag02] Michael Fagan. Design and code inspections to reduce errors in program development. In Manfred Broy and Ernst Denert, editors, *Software Pioneers: Contributions to Software Engineering*, pages 575–607. Springer, 2002.

[Feh93] Rainer Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In Grzegorz Rozenberg, editor, *Applications and Theory of Petri Nets (Advances in Petri Nets 1993, 12th International Conference on Applications and Theory of Petri Nets, Proceedings)*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 1993.

[FL90] Yulin Feng and Junbo Liu. A Temporal Approach to Algebraic Specifications. In Jos C. M. Baeten and Jan Willem Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR'90, Proceedings)*, volume 458 of *Lecture Notes in Computer Science*, pages 216–229. Springer, 1990.

[Frü09] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.

[GH94] Stephen Gilmore and Jane Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In Günter Haring and Gabriele Kotsis, editors, *7th International Conference on Computer Performance Evaluation: Modeling Techniques and Tools (Proceedings)*, volume 794 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 1994.

[GMR+07] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck van Weerdenburg. The Formal Specification Language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[GP95] Jan Friso Groote and Alban Ponse. The syntax and semantics of μCRL. pages 26–62, 1995.

[GS77] Chris Gane and Trish Sarson. *Structured Systems Analysis: Tools and Techniques*. McDonnell Douglas Systems Integration Company, 1977.

[GTW78] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. Prentice-Hall, 1978.

[GW05] Jan Friso Groote and Tim A. C. Willemse. Parameterised Boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005.

[Han09] Dexter A. Hansen. Flowcharting help page (Tutorial), December 2009. http://home.att.net/~dexter.a.hansen/flowchart/flowchart.htm.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hau01] Øystein Haugen. MSC-2000 interaction diagrams for the new millennium. *Computer Networks*, 35(6):721–732, 2001.

[Hau05] Øystein Haugen. Comparing UML 2.0 Interactions and MSC-2000. In Daniel Amyot and Alan W. Williams, editors, *System Analysis and Modeling, 4th International SDL and MSCWorkshop (SAM'04, Revised Selected Papers)*, volume 3319 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2005.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, September 1991.

[Hen96] Thomas A. Henzinger. The Theory of Hybrid Automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[Hil99] Rich Hilliard. Views and Viewpoints in Software Systems Architecture. In *First Working IFIP Conference on Software Architecture (WICSA 1, Proceedings)*, February 1999.

[HLL+03] Christopher Hylands, Edward Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the PTOLEMY project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, July 2003.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[HMS08] David Harel, Shahar Maoz, and Itai Segall. Some Results on the Expressive Power and Complexity of LSCs. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2008.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HS03] Øystein Haugen and Ketil Stølen. STAIRS: Steps to Analyze Interactions with Refinement Semantics. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *The Unified Modeling Language, Modeling Languages and Applications (UML 2003 Proceedings)*, volume 2863 of *Lecture Notes in Computer Science*, pages 388–402. Springer, 2003.

[IEE97] IEEE 1028-1997: Standard for Software Reviews. Institute of Electrical and Electronics Engineers (IEEE), 1997.

[IR05] Yoshinao Isobe and Markus Roggenbach. A Generic Theorem Prover of CSP Refinement. In Nicolas Halbwachs and Lenore Zuck, editors, *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05, Proceedings)*, volume 3440 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2005.

[Jac75] Michael A. Jackson. *Principles of Program Design.* Academic Press, Orlando, FL, USA, 1975.

[Jac04] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Jac05] Hugh Jack. Automating Manufacturing Systems with PLCs, April 2005.

[JDJ+06] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, and Ivan Porres. Model Checking Dynamic and Hierarchical UML State Machines. In Benoît Baudry, David Hearnden, Nicolas Rapin, and Jörn Guy Süß, editors, *3rd International Workshop on Model Development, Validation and Verification (MoDeV²a'06, Proceedings)*, pages 94–110, 2006.

[Jen81] Kurt Jensen. Coloured Petri Nets and the Invariant-Method. *Theoretical Computer Science*, 14:317–336, 1981.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, 2007.

[Jos92] Mark B. Josephs. Receptive Process Theory. *Acta Informatica*, 29(1):17–31, 1992.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, 1974.

[KLSV03] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In *24th IEEE Real-Time Systems Symposium (RTSS 2003, Proceedings)*, pages 166–177. IEEE Computer Society, 2003.

[KPDRW97] Georg Kösters, Bernd-Uwe Pagel, Thomas De Ridder, and Mario Winter. Animated Requirements Walkthroughs based on Business Scenarios. In *5th European Conference on Software Testing, Analysis & Review (euroSTAR'97, Proceedings)*, 1997.

[KRHS05] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. Refining UML Interactions with Underspecification and Nondeterminism. *Nordic Journal of Computing*, 12(2):157–188, 2005.

[KSW01] Georg Kösters, Hans-Werner Six, and Mario Winter. Validation and Verification of Use Cases and Class Models. In *7th International Workshop on Requirements Engineering: Foundations forSoftware Quality (REFSQ'2001, Proceedings)*, 2001.

[KW99] Anneke Kleppe and Jos Warmer. *The Object Constraint Language: Precise Modeling with UML.* Object Technology. Addison Wesley, 1999.

[KW02] Anneke Kleppe and Jos Warmer. The Semantics of the OCL Action Clause. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2002.

[Law01] Adrian E. Lawrence. CSPP and Event Priority. In Alan G. Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures*, volume 59 of *Concurrent Systems Engineering*, pages 67–92. IOS Press, September 2001.

[Law02] Adrian E. Lawrence. HCSP: Imperative State and True Concurrency. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures*, volume 60 of *Concurrent Systems Engineering*, pages 39–56. IOS Press, October 2002.

[LB02] Rik Van Landeghem and Carmen-Veronica Bobeanu. Formal Modelling of Supply Chain: An Incremental Approach Using Petri Nets. In A. Verbraeck and W. Krug, editors, *14th European Simulation Symposium and Exhibition: Simulation in Industry – Modeling, Simulation and Optimization (ESS 2002, Proceedings)*, pages 323–327, 2002.

[LIRM02] Chuang Lin, Yang Qu II, Fengyuan Ren, and Dan C. Marinescu. Performance Equivalent Analysis of Workflow Systems Based on Stochastic Petri Net Models. In Yanbo Han, Stefan Tai, and Dietmar Wikarski, editors, *1st International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002, Proceedings)*, volume 2480 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2002.

[Liu98] Zhen Liu. Performance analysis of stochastic timed Petri Nets using linear programming approach. *IEEE Transactions on Software Engineering*, 24(11):1014–1030, 1998.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, V11(6):637–664, December 1999.

[Lon95] Jim Long. Relationships Between Common Graphical Representations used in Systems Engineering. In *5th Annual International Symposium of the International Council on Systems Engineering (Proceedings)*, 1995.

[LSV03] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O Automata. *Information and Computation*, 185(1):105–157, 2003.

[LT89] Nancy Lynch and Mark Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[MD98] Brendan P. Mahony and Jin Song Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *20th International Conference on Software Engineering (ICSE'98, Proceedings)*, pages 95–104. IEEE Computer Society Press, 1998.

[Mea09] Mealy machine. From Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Mealy_machine, Accessed 17.11.2009.

[MHST03] Till Mossakowski, Anne Haxthausen, Donald Sannella, and Andrzej Tarlecki. CASL - The Common Algebraic Specification Language: semantics and proof theory. In *New Methods in Language Processing, Studies in Computational Linguistics*, 2003.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MM07] Nicolae Marian and Yue Ma. Translation of Simulink Models to Component-based Software Models. In *8th International Workshop on Research and Education in Mechatronics (REM'2007, Proceedings)*, pages 262–267, June 2007.

[Moo56] Edward F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.

[Mos05] Till Mossakowski. Heterogeneous theories and the heterogeneous tool set. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, editors, *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes (Part I). *Information and Computattion*, 100(1):1–40, 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes (Part II). *Information and Computation*, 100(1):41–77, 1992.

[Mur89] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[MV90] S. Mauw and G. J. Veltink. A Process Specification Formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[NAS95] Techniques of Functional Analysis. Systems Engineering Handbook SP-610S, NASA, June 1995.

[NB09] Erzsébet Németh and Tamás Bartha. Formal Verification of Safety Functions by Reinterpretation of Functional Block Based Specifications. In Darren D. Cofer and Alessandro Fantechi, editors, *13th International on Formal Methods for Industrial Critical Systems, (FMICS 2008, Revised Selected Papers)*, volume 5596 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2009.

[NS73] Isaac Nassi and Ben Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8):12–26, 1973.

[OMG08] OMG Systems Modeling Language (SysML), v1.1. Specification, Object Management Group, 2008. http://www.omg.org/cgi-bin/doc?formal/08-11-02.pdf(26/04/10).

[OMG09] Unified Modeling Language (UML): Superstructure, Version 2.2. Specification, Object Management Group, February 2009. http://www.omg.org/spec/UML/2.2/Superstructure/PDF/(13/12/09).

[Par81] David Michael Ritchie Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science (5th GI-Conference, Proceedings)*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[Pea09] Jon Pearce. Object-oriented analysis, December 2009. http://www.cs.sjsu.edu/faculty/pearce/ooa/front.htm.

[PM05] Gergely Pintér and István Majzik. Runtime Verification of Statechart Implementations. In Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky,

editors, *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 148–172. Springer, 2005.

[Pra95] K. V. S. Prasad. A Calculus of Broadcasting Systems. *Science of Computer Programming*, 25(2-3):285–327, 1995.

[RG01] Mark Richters and Martin Gogolla. OCL – Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*, pages 43–69, Berlin, 2001. Springer.

[RH88] A. W. Roscoe and C. A. R. Hoare. The Laws of Occam Programming. *Theoretical Computer Science*, 60(2):177–229, 1988.

[Rid04] J. W. Rider. System diagram essentials. Technical report, J. W. Rider Consulting, October 2004. http://www.jwrider.com/lib/DiagramEssentials.htm[27/04/2010].

[Rog06] Markus Roggenbach. CSP-CASL: A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[RS99] Doug Rosenberg and Kendall Scott. *Use Case driven object modeling with UML: a practical approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[RST09] Daniel Ratiu, Wolfgang Schwitzer, and Judith Thyssen. A System of Abstraction Layers for the Seamless Development of Embedded Software Systems. Technical Report TUM-I0928, Technische Universität München, 2009.

[RTC82] DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA), 1982.

[SDJ+92] Steve Schneider, Jim Davies, D. M. Jackson, George M. Reed, Joy N. Reed, and A. W. Roscoe. Timed CSP: Theory and Practice. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice (REX Workshop, Proceedings)*, volume 600 of *Lecture Notes in Computer Science*, pages 640–675. Springer, 1992.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-Based Development of Embedded Systems. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems (OOIS 2002 Workshops, Proceedings)*, volume 2426 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2002.

[SST92] Donald Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Parameterisation Revisited. *Acta Informatica*, 29(8):689–736, 1992.

[Ste03] Alan B. Sterneckert. *Critical Incident Management: A Methodology for Implementing and Maintaining Information Security.* Auerbach Publications, 2003.

[STMW04] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In *2nd International Conference on Software Engineering and Formal Methods (SEFM'04, Proceedings)*, pages 174–183. IEEE Computer Society, 2004.

[SW83] Donald Sannella and Martin Wirsing. A Kernel Language for Algebraic Specification and Implementation. In *Foundations of Computation Theory (Proceedings of the International FCT-Conference, 1983)*, volume 158 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 1983.

[SZS08] Igor Siveroni, Andrea Zisman, and George Spanoudakis. Property Specification and Static Verification of UML Models. In *3rd International Conference on Availability, Reliability and Security (ARES 2008, Proceedings)*, pages 96–103. IEEE Computer Society, 2008.

[TYC95] Jeffrey J. P. Tsai, Steve Jennhwa Yang, and Yao-Hsiung Chang. Timing Constraint Petri Nets and Their Application to Schedulability Analysis of Real-Time System Specifications. *IEEE Transactions on Software Engineering*, 21(1):32–49, 1995.

[VHD09] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, February 2009.

[Wan07] Jiacun Wang. Charging information collection modeling and analysis of GPRS networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 37(4):473–481, 2007.

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Formal Specification and Development in Z and B (ZB 2002, Proceedings)*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

[Wir90] Martin Wirsing. Algebraic Specification. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 675–788. Elsevier Science Publishers B. V., 1990.

[WM86] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *21st International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2008, Proceedings)*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.

[You89] Edward Yourdon. *Modern structured analysis*. Yourdon Press, Upper Saddle River, NJ, USA, 1989.

[You09] Edward Yourdon. Yourdon structured analysis wiki, December 2009. http://yourdon.com/strucanalysis/wiki/index.php?title=Introduction.

[YzYzYf04] Liang Yi-zhi, Wang Yan-zhang, and Liu Yun-fei. The formal semantics of an UML activity diagram. *Journal of Shanghai University (English Edition)*, 8(3):322–327, 2004.

[ZD89] MengChu Zhou and Frank DiCesare. Adaptive Design of Petri Net Controllers for Error Recovery in Automated Manufacturing Systems. *IEEE Transactions on Systems, Man and Cybernetics*, 19(5):963–973, September 1989.

[ZLL07] Yang Zhao, Jie Liu, and Edward A. Lee. A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *13th IEEE Real-Time and*

Embedded Technology and Applications Symposium (RTAS 2007, Proceedings), pages 259–268. IEEE Computer Society, 2007.

A Analysis Tool Support

A.1 Inspections, Reviews and Walkthroughs

Rational Rose In [KSW01] is described a UML based tool implemented in Rational Rose[12], comprising modelling, validation, and verification. The analysis is performed on UML class, use case and activity diagrams. The validation is focused on inspections and walkthroughs by a two level inspection process:

1. In the first phase analysts and testers pre-inspect the use case diagram and the domain class model to detect violations of the UML syntax and modelling rules defined in [KSW01]. For each use case, we determine the set of domain classes with instances involved in the execution of the work unit (i. e., which are created, retrieved, modified, or deleted). The set of classes is called the class scope of the use case. Analogously, we derive the class scope for each action of the according activity graph.
2. On the second phase, a series of walkthroughs business scenarios is jointly conducted by all parties involved in the development [KPDRW97]. In the course of the walkthroughs, domain experts and users focus on the textual descriptions of the actions, check their correctness.

In the verification phase the class model is verified against the use cases and activity graphs diagrams. The operations are performed in two steps:

1. On the first step, the verification starts with detailed inspections of the formal parts of the use case model (e.g. pre- and post-conditions and edge-guards of the activity graphs). Then the (refined) class model is checked to detect incomplete, inconsistent, and/or ambiguous specifications and/or missing items.
2. On the second step, analysts and testers walkthrough the scenarios (re-used from the validation phase) and investigate the thread of executions of operations in the class model.

A.2 Testing and Simulation

ASCET. ASCET[13] is a tool family for developing embedded system software by a model based ECU software development process, in the framework of the automotive industry. In ASCET model components can be specified with block diagrams and state machines. In the tools there are instruments for performing testing and simulation.

Rational Rhapsody. Rational Rhapsody[14] is a tool used to develop real-time or embedded systems based on UML and SysML. In Rational Rhapsody development environment it is possible to apply model driven testing with visualisation of test cases and automatic test execution, monitoring, and generation. Rational Rhapsody enables the visualisation of the model through simulation. Simulation is the execution of behaviours and associated definitions

[12] http://www-01.ibm.com/software/awdtools/developer/rose/
[13] http://www.etas.com/en/products/ascet_software_products.php
[14] http://www-01.ibm.com/software/awdtools/rhapsody/

in the model. Rational Rhapsody simulates the behaviour of your model by executing its behaviours captured in statecharts, activity diagrams, and textual behaviour specifications.

SCADE. SCADE (Safety Critical Application Development Environment)[15] is both a language and a toolset for the development of critical embedded software. The SCADE language is a graphical synchronous data flow specification language. The data flow structure fits the block diagram approach. Besides constructs based on data flow it´s also possible to define state machine. The code automatic generated is qualified to the strictest level of the Civilian Avionics Standard DO-178B, Level A. SCADE contains a fully functional Simulator module enabling simulation for all language constructs and state machines.

MATLAB/Simulink. Simulink[16] is an extension of Mathworks Matlab, for designing and simulating embedded systems, which are modelled as block diagrams consisting of blocks from the Simulink block library. There are Simulink extensions which implements testing and simulation:

- Simulink Verification and Validation aids model verification by creating designs and component test cases. It provides six model coverage analysis metrics 2.7.2. Modelling standards checks provided are based on The MathWorks Automotive Advisory Board (MAAB) Style Guidelines.
- Simulink Design Verifier generates test cases and property-proving analysis reports. It proves model properties and generates examples of violations. Model coverage test objectives include MC/DC coverage, which is required by safety critical standards, such as DO-178B.

TURTLE Tool. TURTLE Tool[17] is an open source toolkit (based on TURTLE [ACLSS04]) which provides simulation techniques. The simulation is performed translating a UML analysis diagrams (interaction overview and sequence diagrams) or UML design diagrams (class and activity diagrams), into an RT-LOTOS specification [BK85] and executing it in the RT-LOTOS verification tool.

WinA&D WinA&D[18] is a design tool which supports UML use case diagrams and descriptions, class and package diagrams, state diagrams, collaboration and sequence style interaction diagrams, activity diagrams, and deployment diagrams. WinA&D performs tests for checking reveal errors, inconsistencies, or incomplete data, and the flow of information between diagram levels.

[15] http://www.esterel-technologies.com/products/scade-suite/
[16] http://www.mathworks.com/products/simulink/
[17] http://labsoc.comelec.enst.fr/turtle/
[18] http://www.excelsoftware.com/wina&dproducts.html

mCRL2 The mCRL2[19] language is a specification language for describing communication behaviour among systems. The behavioural part of the language is based on the Algebra of Communicating Processes (ACP) [BK84] which is extended to include data and time. The language is supported by a toolset enabling simulation, visualisation, behavioural reduction and verification of software requirements. Validation of specification is supported by simulation which can be used for an insight into the behaviour of a system.

A.3 Formal Verification

Simulink Design Verifier Simulink Design Verifier[20] is an extension for Simulink which uses formal analysis techniques provided by Prover Plug-In. It is developed and maintained by Prover Technology. It performs exhaustive formal analysis of Simulink models to confirm the correctness with respect to given properties. The user can specify these as assertions. To prove these assertion, Simulink Design Verifier searches for all possible values for a Simulink function in order to find a simulation that satisfies it. It will compute all possible input signals automatically. Simulink Design Verifier can only check assertions and not temporal logic properties as made for instance from model checkers.

SCADE Design Verifier SCADE Design Verifier[21] is a formal proof engine within the SCADE tool. It uses formal methods for all the language constructs and state machines models. This component allows the formal verification of certain properties of models by Prover Plug-In as already explained for Simulink Design Verifier.

A.4 Model Checking

Assessment Studio The Assessment Studio[22] for ASCET, can check ASCET-MD models against company-specific modelling and implementation guidelines and also style conventions such as AUTOSAR Styleguide for ASCET or check models for MISRA compliance.

Toolkit for Conceptual Modelling In [EW04] is described a tool, an extension of Toolkit for Conceptual Modelling[23] (TCM), that supports verification of models specified in UML activity diagrams. The tool translates an activity diagram into a format for NuSMV [CCG+02] model checker. Safety requirements are checked translating them in an extension of LTL temporal logic. If a requirement fails an error trace is returned by NuSMV, which is presented in the TCM extended tool, by highlighting a corresponding path in the activity diagram.

PRISE PRISE [NB09] is a tool defined for Function Block Diagram specifications. It performs model checking translating the diagrams in a Petri net representation, the properties verified are based on CTL temporal logic.

[19] http://www.mcrl2.org
[20] http://www.mathworks.com/products/sldesignverifier/
[21] http://www.esterel-technologies.com/products/scade-suite/design-verifier
[22] http://www.match-technologies.com/
[23] http://wwwhome.cs.utwente.nl/~tcm/

TURTLE Tool TURTLE Tool[24] provides also model checking for the same specification techniques described in subsection A.2. TURTLE Tool translates the specification for Kronos [DY95] a model checker for real-time system. The properties to be verified are in Timed CTL an extension of the temporal logic CTL.

Papyrus UML The Static Verification Tool [SZS08] performs model checking on UML models composed of UML class diagrams which define the structure of the model and UML statechart diagrams which specify the behaviour of each of the defined classes. It translates model to verify in PROMELA language and so uses SPIN as model checker, with an extension of LTL temporal logic for specifying the properties. The Static Verification Tool is implemented in Java and is packaged as an Eclipse plug-in that runs along the Papyrus UML[25] graphical modeler.

Hugo/RT Hugo/RT[26] [BBK+04] translates a UML model in the following way: UML statechart diagrams are associated with UML class diagram to specify the model to verify and communication diagrams describe how the objects of a model may interact. The tool generates input for two model checkers: SPIN and UPPAAL [BDL04]. The model checkers can verify whether the interactions expressed by a UML communication diagram are realised by the state machines.

PROCO In PROCO[27] [JDJ+06] a system to be verified is modelled using UML class diagrams and statecharts. These models with assert specifications are translated and verified in SPIN model checker.

Rhapsody in C++ In [STMW04] is reported a verification environment for UML models, that has been integrated within Rhapsody in C++[28]. The specifications to be verified can be specified using temporal patterns or with a graphical specification formalism called Life Sequence Charts [DH01]. The verification is applied to UML models composed by UML statechart diagrams and UML class diagram. They are transformed in a format for VIS model checker [BHSV+96]: a finite state machine description of the model and a temporal logic CTL formula for the property to verify.

PRES+ Petri-net based Representation for Embedded Systems (PRES+) [CEP00] is an extension to the classical Petri nets model for representing embedded systems. PRES+ explicitly captures timing information, allows systems to be represented at different levels of granularity. In [CEP00] is presented an approach for verifying PRES+ specification using model checking.

[24] http://labsoc.comelec.enst.fr/turtle/
[25] http://www.papyrusuml.org/
[26] http://www.pst.ifi.lmu.de/projekte/hugo/
[27] http://www.tcs.hut.fi/SMUML/
[28] http://www-01.ibm.com/software/awdtools/rhapsody/

PRES+ model is translated into an hybrid automata and then is used an existing verification tool, namely HyTech [29], to check properties expressed as CTL and Timed CTL formulas, therefore it is possible to validate design properties and timing requirements.

There are several types of analysis that can be performed on systems represented in PRES+. A given marking, i. e., absence or presence of tokens in places of the net, may represent the state of the system in the dynamic behaviour of the net. Based on this, different properties can be studied. For instance, the designer could be interested in proving that the system eventually reaches a certain state whose marking represents the completion of a task, so a reachability analysis. In many embedded applications, time is an essential factor. Therefore, it is needed not only to check that a certain state will eventually be reached but also to ensure that this will occur within some bound on time. In PRES+, time information is attached to tokens so that we can analyze quantitative timing properties: we may prove that a given place will be eventually marked and that its time stamp will be less than a certain time value that represents a temporal constraint.

Failures-Divergences Refinement Hoare's Communicating SequenStial Processes (CSP) [Hoa85] is a well known Process algebra implementation based around the theory of concurrency. FDR (Failures-Divergences Refinement)[30] is a refinement checker for the process algebra CSP. In common with many other model checkers, it works by "determinising" (or normalising) a specification and enumerating states in the cartesian product of this and the implementation. Unlike most, the specification and implementation are written in the same language. Adaptations of FDR have been, or are being made, to accommodate other input notations such as UML. FDR has been much used in industrial work in areas such as computer security, safety-critical systems, communications networks and telecommunications.

mCRL2 The mCRL2[31] language is a specification language based on process algebra as described in subsection A.2. To apply model checking, a modal formula must be provided that states some functional requirement on the mCRL2 specification. In mCRL2 modal formulae are specified in a variant of the modal μ-calculus extended with regular expressions, data and time. In combination with the specification this formula is transformed into a parameterised boolean equation system (PBES) [GW05]. Several tools are available for solving PBESs.

A.5 Theorem Proving

Heterogeneous Tool Set Common Algebraic Specification Language (CASL) [MHST03] is a tool for algebraic specification. Heterogeneous Tool Set[32] [Mos05] is a tool used for CASL. Heterogeneous Tool Set performs the verification in two phases: a parser checks the syntactic correctness of a specification according to the CASL grammar and a static checker checks the semantic correctness. It is possible to translate the CASL model to high-order logic which

[29] http://www-cad.eecs.berkeley.edu/ tah/HyTech/
[30] http://www.fsel.com/software.html
[31] http://www.mcrl2.org
[32] http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/index_e.htm

allows to re-use existing theorem proving, for instance there is an interface from CASL to Isabelle/HOL [WPN08] theorem prover.

mCRL2 The mCRL2[33] toolset (described in subsection A.2) supports theorem proving. Every analysis of an mCRL2 specification starts by a transformation of the specification into linear form. This is achieved by the lineariser, which transforms a restricted yet practical subset of mCRL2 specifications to linear process specifications (LPSs). With theorem proving technology it is possible to check invariants on an LPS. Validity of boolean data expressions can also be checked using external SAT solvers.

A.6 Runtime Verification

Tool for Statechart Diagrams Work in [PM05] introduces a runtime verification framework for monitoring of applications specified by UML statechart diagrams. The verification is made in two phases:

1. Firstly, are formally defined correctness temporal requirements by a temporal logic language for UML statecharts. A corresponding runtime verifier component verifies it.
2. During the subsequent model refinement steps the developers prepare the fully behavioural model of the system by manual programming or automatic source code generation. The final behavioural model can be directly used for runtime verification of deviations from the behaviour specified by the statechart. UML statecharts are transformed to Extended Hierarchical Automata (EHA [LMM99]), that is used as a reference model for a statechart-level (EHA-level) runtime verifier component. This component is capable of detecting behavioural anomalies and operational errors throughout the entire life cycle of the observed object: proper initialisation (entering the states belonging to the initial configuration), event processing (selecting transitions to be fired) and the firing of transitions (leaving source states, performing the action associated to the transition and entering the target states) according to the UML semantics.

[33] http://www.mcrl2.org