Institut für Informatik
der Technischen Universität München

RBSLA: Rule-Based Service Level Agreements

Knowledge Representation for Automated e-Contract, SLA and Policy Management

Adrian Paschke

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Torsten Grust
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Martin Bichler
	2.	Univ.-Prof. Bernd Brügge, Ph.D.

Die Dissertation wurde am 26.06.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.10.2007 angenommen.

Acknowledgements

First, I would like to thank my referees Prof. Dr. Martin Bichler and Prof. Dr. Bernd Brügge. Prof. Bichler gave me the opportunity to work on this exciting subject under his supervision. He generously supported my RBSLA project and gave me the encouragement and the freedom I needed to complete the maturing of the ideas to a practical relevant project.

A big "thank you" goes to Alexander Kozlenkov from the City University London (now Betfair Ltd.) and to Dr. Jens Dietrich from the Massey University in New Zeeland for countless invaluable discussions and constructive comments. They tirelessly motivated and inspired me through all the stages of this dissertation. It is an honor to collaborate with them in the precious spirit of open source development and extreme programming. I am grateful for the time and energy they devoted to the supervision of this work and even happier to have them as friends.

Special thanks go to Prof. Dr. Harold Boley who invited me as a guest researcher at the National Research Council in Canada and who gave me the unique opportunity to contribute my RBSLA work to the RuleML standardization initiative as a Steering Committee member and as a Co-Chair of Reaction RuleML. He also made it possible for me, as a non W3C member, to present parts of my works in terms use cases to the newly formed W3C Rule Interchange Format working group. I would like to thank him, Michael Kifer, Said Tabet, Mike Dean and the other RuleML and Reaction RuleML standardization members for their constructive comments and feedback during our weekly telephone conferences and face to face meetings and it is an honor to work together with them.

Further thanks go to Prof. Dr. Gerd Wagner who invited me as a guest researcher to his group at the Technical University Brandenburg and to Prof. Dr. Hoelldobler from the TU Dresden with whom I had fruitful discussion about Situation Calculus vs. Event Calculus during the ICCL Summer School "Logic-based Knowledge Representation" in Dresden.

Further thanks go to all my colleagues at the institute and in particular the Internet Based Information Systems (IBIS) group for providing an excellent working environment. I would also like to thank all industry partners and collaborators such as Siemens SBS, Siemens AG, IBM, HP, Wacker Chemie, Lufthansa, BMW, O2, Deutsche Telekom for giving an insight into IT service centers and IT service management practices, for providing valuable material such as real world SLAs and for giving feedback on the practical and industrial relevance and usability of my project.

Special thanks go to my family, my wife Aira and my daughter Aureelia as well as my parents and sisters, for their love, understanding and encouragement. They have been a constant source of joy and strength during the ups and downs of the last years and I know that it was really a hard to time for them.

Abstract

The emergence of service oriented computing (SOC) together with novel business models such as On-Demand or Utility Computing leverage an open environment based upon loosely coupled and distributed services and dynamic service provider and service consumer relationships. Dynamic service supply chains (a.k.a. business services networks) based upon service-oriented and event-driven architectures (SOAs and EDAs) and technology standards such as Web Services, Grid Computing and the Semantic Web ease the outsourcing of complex IT infrastructure to IT service providers. IT service providers must be able to fulfil their policy and service-quality commitments based upon Service Level Agreements (SLAs) with their service customers. They need to manage, execute and maintain thousands of contracts for different customers and different types of services while facing rapidly changing business and system environments, a wide range of domain knowledge due to diverse organizational boundaries, huge amounts of scattered data managed in possibly distributed heterogeneous data sources and a great variety of more or less cooperative entities (human or automated agents) involved during the contracts life cycles. Moreover, they have to accommodate high requirements regarding correctness, robustness and traceability of drawn results and triggered reactions from the service contracts in order to establish trust and fulfil legal compliance rules.

While past research in service oriented computing has focused on the fairly static functional description and the operators of services - including publication, discovery, selection, binding and composition of basic services - the top layer in the SOA pyramid providing support for IT service management (ITSM) and in particular IT Service Level Management based upon SLAs describing non-functional service properties such as quality of service (QoS) and business policies has not been as thoroughly explored. I argue that the prevailing IT service level management tools and the actual SLA and contract languages such as WSLA, WS-Agreement or WS-Policy which mainly focus on the syntactical description of electronic contracts and policies using XML markup are not suitable to solve this task. The complexity of contractual logic in SLAs in an open, dynamic service oriented environment such as the Semantic Web requires new forms of knowledge representation and new technical solutions with a high degree of agility and flexibility in order to efficiently engineer, manage and continuously monitor and enforce large amounts of complex and distributed IT service contracts.

I propose a declarative rule-based approach for SLA representation and service level management (SLM) of IT services, where sophisticated logical knowledge representation (KR) concepts and artificial intelligence (AI) techniques are used to describe the contracts in a generic way. Using rules for SLA representation has several advantages:

1. reasoning with rules is based on a semantics of formal logic - a variation of first order predicate logic - enabling automated rule chaining by resolution and variable unification, which alleviates the burden of having to

implement extensive control flows as in imperative programming and allows for easy extensibility without affecting the underlying mechanisms and architectures,

2. it allows for a compact and comprehensible human and machine-oriented representation and high levels of automation and flexibility to adapt to rapidly changing requirements, and

3. it is relatively easy for the end user to write rules and hence rapidly engineer and maintain SLAs.

The basic idea is, that business practitioners employ rules - which might be predefined as templates by experts - to describe their SLAs, the responsibility to interpret and execute these contracts and to decide on how to do this is delegated to a standard interpreter - a rule engine. Along with providing the possibility to describe complex contract structures as modular rule sets which can be collaboratively engineered, maintained and interchanged in an open distributed environment and which can be automatically executed and enforced by a rule engine, my formal-logical rule based approach provides means to:

- integrate external systems, data sources and domain-specific Semantic Web ontologies into rule executions,
- dynamically change and update the rule-based SLA specifications at runtime, and
- automatically verify, and validate consistency and correctness of the rule sets.

The proposed research is on the interplay of common software engineering (SE) providing well-established methodologies, imperative programming and database solutions, on the one hand and artificial intelligence and knowledge representation following the declarative rule based programming paradigm, on the other. A third important research dimension is added due to the inherent business-driven character of the SLA domain, the indispensable compliance of contracts with legal regulations and the deep impact on all operational, tactical and strategic IT service management (ITSM) processes. The objective for the proposed research is to adapt the values, principles and practices in the addressed fields and bring together the different requirements within one practical framework. Although, rule based systems have been investigated comprehensively in the realms of declarative logic programming and expert systems over the last decades and some development on specialized problems already exists in the domain of contract and policy management a comprehensive and coherent KR for rule-based SLA and policy representation is missing. This dissertation aims to identify and implement adequate knowledge representation concepts from the area of artificial intelligence (AI) and knowledge representation (KR). A particular interest is in the investigation and combination of adequate first order logic subsets such as description logics as basis for Semantic Web ontology languages and expressive non-monotonic logic programming techniques and logical formalisms such as defeasible logic, deontic logic, temporal event and action

logics, transaction and update logics as a means for deriving formal declarative contract specifications. In particular, the goal of this dissertation is to develop a practical SLA management framework that

1. includes an expressive, efficient and coherent KR for declaratively formalizing modular contract structures and integrating external domain-specific (Semantic Web) vocabularies and external systems, data and functionalities;
2. aids in collaborative development, verification, maintenance, interchange and execution of contracts in open distributed environments such as the Semantic Web by a superimposed declarative rule based SLA markup and interchange language and translations from this language into the execution KR in the target environment;
3. provides a stable, efficient and highly scalable infrastructure for distributed rule-based contract management, rule interchange and rule-based complex event processing
4. SLM support via tool based contract engineering, management and runtime visualization, alerting and explanation in service dashboard views;
5. provides integration in and interoperation with standard enterprise application architectures and distributed web architectures

The primary contribution of this dissertation is a declarative, compact and highly expressive KR proposal, the ContractLog KR, consisting of selected adequate formalisms for SLA representation and a rule based Service Level Agreement markup language, called RBSLA, used to serialize SLAs in a interchangeable format in XML. The approach has a clear logical semantics, is computationally efficient for larger SLAs and data sets, reliable and traceable even in case of incomplete or contradicting knowledge, provides support for validation and verification of contractual specifications, is flexible in a way that allows to quickly alter the SLA specification, supports declarative programming (not just syntactical specification) of arbitrary functionalities and decision procedures, provides means for serialization and interchange based on the superimposed RBSLA mark-up language and enables reuse and integration of external ontologies, data, systems and object-oriented procedures through hybrid semantic and object-oriented integration techniques and enterprise service technologies, e.g. an Enterprise Service Bus. Compared to traditional contract management approaches my rule-based approach has the following major advantages:

1. contract rules are externalized and easily shared among multiple applications and domains (avoiding vendor lock-in and facilitating modularization and information hiding);
2. encourages reuse, shortens development time and safeguards the engineering process;
3. changes can be made faster and with less risk; lowers cost incurred in the modification of contractual business logic;

4. it provides are more human way to represent SLAs while at the same time enabling machine-interpretation and automated execution;
5. results and reactions are highly reliable, traceable and verifiable;
6. integrates and interoperates with enterprise application architectures and distributed service-oriented web technologies;

While most recent key developments in research on logic-based knowledge representation have been of the more theoretical sort, in this dissertation I follow a constructivistic, SE-oriented methodology and adopt the Design Science Research approach as described by Hevner et al. [HMPR04]. With my new alternative design artefact I try to overcome real-world problems which are of high relevance and importance for IT service provider such as insufficient automation of IT service level management based on SLAs, slow change cycles of contractual agreements in rapidly changing, highly-distributed and loosely coupled service oriented environments with business models such as on-demand computing or utility computing and several new regulations and compliance rules (Sarbanes Oxley, Basel II). This substantial system development effort made it clear that moving a KR idea into real practice is not just a matter of designing a new KR theory and investigation of theoretical results, but needs significant research to seriously implement adequate logics in a coherent framework even after the basic theory is in place. It turns out that there is a long and difficult road to travel from the pristine clarity of an initial logical formalism to a practical system that really works. To evaluate the adequacy, utility and practicality of the proposed rule based SLM approach and the implementations I use established methodologies in SE, KR and LP reaching from requirements and adequacy analysis, theoretical worst-case analysis, experimental performance benchmark tests and simulations as well as use case implementations in proof-of-concept settings and contributions to open-source projects (Prova, Mandarax) and standardization initiatives (RuleML, Reaction RuleML, W3C Rule Interchange Format RIF Use Cases).

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Problem Statement

Outsourcing of complex IT infrastructure and applications to IT service providers has become increasingly popular [Dav03] and led to much recent development in open, dynamic and distributed systems such as the semantic web, grid computing systems, or pervasive computing environments and multi-agent systems. Flexibility in dynamically composing new business processes and integrating heterogenous information systems (HIS) enabling ad-hoc cooperations is one of the main aims of the recent service oriented computing (SOC) paradigm which uses (semantic) web services on top of the Semantic Web. This service-orientation enables novel business models such as on demand computing or utility computing which charge resource usage on a per-use basis. In this open service-oriented scenario based upon loosely coupled and distributed services and dynamic service provider and service consumer relationships, service level agreements (SLAs) defining the (business) policies and performance criteria an IT service provider promises to meet while delivering a service are of vital importance with a deep impact on the operational, strategic and organizational processes in the enterprizes and the reliability and trust relationships across multiple partners. SLAs are fundamental for any kind of service supply chain and assume a central position in popular IT service management standards such as IT Infrastructure Library (ITIL) (www.itil.co.uk) and BS15000 (www.bs15000.org.uk). A well-defined and effective SLA correctly fulfils the expectations of all participants and defines the quality attributes and guarantees that a service is required to process. It provides metrics for measuring the performance of the agreed upon Service Level Objectives (SLOs) and defines the rules used to monitor service execution and detect violations of SLOs. It typically also sets out the remedial actions and any penalties that will take effect if performance falls below the promised service levels. During the monitoring and enforcement phase the SLA rules will be used to detect violations to the promised service levels and to derive consequential activities in terms of actions, rights and obligations. They play a key role in metering, accounting and reporting of IT services and provide data for further refinement of SLAs and for optimizing IT service management (ITSM) on an operational, tactical and strategic level.

In practice, in the upcoming service oriented landscape IT service providers need to manage, execute and maintain thousands of often short-termed SLAs which are possibly scattered among business partners, organizations or departments and which are individualized for different customers and different types of services. Manually maintaining and executing large numbers of interlinked

and modular service contracts which are just defined in natural language as in the traditional long running outsourcing and application service providing (ASP) models becomes impossible. Commercial IT service level management (SLM) tools such as IBM Tivoli, HP OpenView, CA Unicenter, BMC Patrol, or Microsoft Application Center typically store selected Quality of Service (QoS) parameters such as availability or response time directly as parameters in the application code or database tiers. This approach is restricted to simple, static SLA rules which are often hard-coded in classical imperative programming languages and replicated by different applications. To dynamically change the SLAs from time to time in order to meet new requirements of the business partner, adapt to a changed environment or just to keep a service alive would force an administrator to change the application code in a time consuming and costly imperative programming style. Even when the tool offers possibilities to customize parameters that can be adapted by non-programmers, situations can be easily found where this kind of limited parameter customization is not powerful enough to keep step with the requirements and changes in modern IT service centered environments. Existing declarative SLA specification languages such as WSLA [DDK+04], WSOL [TPP+03] or WS-Agreement [ACD+05] are pure syntactical serialization languages defining a mark-up syntax without a precise logical semantics. They need specialized procedural interpreters, have a restricted expressiveness to describe complex decision and contractual logic in terms of rules using only plain material truth implication and provide no capabilities to declaratively implement new functionalities and decision procedures. Moreover, due to their lack of a precise mathematically grounded semantics verification and validation of the correctness of the SLA specifications as well as predictability, robustness and traceability of results at runtime is not warranted.

In a nutshell, the current technology and tools in IT Service Level Management and SLA representation are not powerful enough to satisfy the requirements of modern service oriented environments and novel service centered business models. Accordingly, a more sophisticated declarative knowledge representation (KR) for representing IT service contracts is needed with a high degree of automation, agility, flexibility and a precise formal semantics. In the following section I will further elaborate on the general requirements for such a KR and an associated IT service level management solution.

1.2 Requirements and Research Issues

Requirements for a practical IT service level management (SLM) tool and a declarative SLA representation language arise from the interplay of common software engineering (SE) for developing a system in an open distributed environment, on the one hand and from knowledge representation of different types of knowledge such as domain models/ontologies, extensional data from various data sources or contractual rules, on the other. A third research dimension is added due to the inherent business-driven character of the SLA domain, the indispensable compliance of contracts with legal regulations and the deep im-

pact on all operational, tactical and strategic IT service management (ITSM) processes.

From a software-engineering point of view, the *functional requirements* for a SLM system capture what the system must do, which can be summarized as providing support for IT service management according to the ITIL ITSM processes, in particular ITIL's service level management (see section 2.1.1), i.e., the establishment of services with appropriate service levels, the implementation of SLAs and the continual identification, monitoring, enforcing and reviewing of the optimally agreed service levels of IT services as required by the business. Beside these functional properties the *non-functional requirements* of the system are of high importance. A distinction can be made between requirements wrt to the *development-time qualities* which influence the efforts and costs IT service provider have to implement SLAs with new service levels, and requirements wrt *run-time qualities* which describe the observable quality properties of the functional requirements during run-time. [Ben97]. The following SE development-time requirements are the most important ones:

- individualizability/customizability: ability to make adaptations and differentiations of contracts (individual SLAs) e.g., first, second or third degree price discrimination policies, local (regional) or system specific differences or domain specific adaptations (using different terminologies/ontologies);
- composability: ability to compose contract hierarchies from sub-contracts which are composed from modules (rule sets) and further submodules;
- interoperability: ability to cooperate with existing external tools, data sources and functionalities
- declarative implementability, modifiability and evolvability: ability to declaratively implement, add or modify (unspecified) future functionality;
- reusability and interchangeability: ability to interchange and (re)-use contracts and contract modules in different target environments and by different entities (collaborative humans, agents, services);

The important SE run-time qualities are:

- active adaptability, reactivity and user-defined configurability: ability to actively react and adapt to changed situations and occurred events and passively (re-)configure / react according to user queries;
- usability: ability for the users (humans or machines) to easily use the SLA specifications and tools wrt to the environment (i.e., in an open distributed environment);
- understandability and explanation: ability of the users (humans or machines) to easily understand produced results;
- efficiency: equated with performance;

- scalability: ability of the SLA system consisting of monitored contracts and external tools, data and users to grow in size while maintaining all other properties and qualities, in particular, efficiency;
- correctness: absolute ability to derive the correct conclusions and reactions from a contract wrt the functional requirements;
- robustness: "reasonable" behavior in unforeseen circumstances and incomplete knowledge;
- safety: fault and conflict tolerance, information hiding (need-to-know principle) etc.;
- verifiability and validation: ability to analyze the correctness;

From the KR point of view (in Artificial Intelligence) the realization of a knowledge based system involves two primary requirements. The first requirement involves precisely characterizing the type of knowledge to be specified (in a knowledge base) and clearly defining the reasoning services including inferences the system needs to provide. The second requirement consists of providing a powerful development environment which makes the interaction of the user (human or machine) with the system more effective. That is, KR focusses on the design of formalisms that are adequate for expressing knowledge and reasoning in a particular domain. Adequacy criteria [MH69] can be used to derive the requirements and assess the usability of a particular KR formalism for SLA representation:

- epistemological adequacy: ability to represent all relevant knowledge;
- heuristical adequacy: ability to execute all inferences with limited resources;
- algorithmic adequacy: complexity and efficiency;
- logic-formal adequacy: soundness, completeness and decidability;
- psychological adequacy: fault-tolerance wrt contradictions, possible failures and incomplete knowledge with an intuitive meaning, understandable and traceable by humans;
- ergonomically adequacy: ability to easily and efficiently use the KR language;

According to these general adequacy criteria, a KR language should be usable to both human and machines. This typically requires to build syntactical language variants with a human-readable syntax and a machine-readable and interpretable syntax and semantics, which can be mapped to each other. As it has been often discussed in literature expressiveness in a KR language trades off against computational complexity [LB87]. The SLA domain faces a huge amount of intensional and extensional knowledge which demands for a complete inference system, which should be worst-case tractable, but at the same

time needs high expressiveness to represent different kinds of knowledge (ontological, relational, object-oriented etc.) and allow sophisticated reasoning in an open and distributed domain such as the Semantic Web. To ensure qualities such as correctness and verifiability, which are crucial requirements in order to fulfil legal regulations and compliance rules and establish trust with the service consumers, a precise semantics with a mathematical basis is of vital importance.

The KR language should be clear, compact, precise and easily adaptable. It should fulfil typical criteria for good language design [Cod71] such as *minimality*, *symmetry* and *orthogonality*:

- Minimality means that the language provides only a small set of needed language constructs, i.e., the same meaning cannot be expressed by different language constructs.
- Symmetry is fulfilled if the same language construct always expresses the same semantics regardless of the context it is used in.
- Orthogonality permits every meaningful combination of a language constructs to be applicable.

An important property which refers to the development-time SE qualities is the extensibility of the language and the interoperability with other representation formats. A completely predefined specification language with no means to declaratively program new functionalities and data structures is only of limited use in the SLA domain due to the frequently changing requirements and the dynamism of the domain (see section 1.1 and section 2.5).

From a business-oriented point of view requirements can be postulated according to short-term business goals relating to the operational management and active decision making process and long-term business goals relating to strategic and organizational decisions. While run-time qualities have to do with the conformity with the execution environment and the direct value they provide for the users, the development-time qualities refer to the cost savings in the development and maintenance phase, thus creating long-term business value e.g., due to a more maintainable and flexible SLM architecture. One of the most important requirement businesses do have nowadays is to overcome the restricting nature of slow IT change cycles and change their business rules including the contractual agreements in order to adapt to a rapidly changing business environment.

The central research question which follows from this general discussion of requirements and adequacy/quality criteria for a flexible SLM solution and SLA KR is:

Which kind of knowledge representation is adequate for an efficient, flexible and distributed design, management, monitoring and enforcement of SLAs?

Three important research issues arise for the KR:

1. representation problem: design an adequate knowledge representation to describe the problem and the answers

2. search problem: efficiently derive answers from the given knowledge by the KR inference formalisms, e.g., goal-driven without visiting solutions twice
3. inference problem: use complete or incomplete knowledge for inferencing; allow to withdraw conclusion or not; automatically resolve conflicts; allow external open knowledge or not; allow constructive quantifications such as temporal or priority based quantifications

1.3 Proposed Approach and Research Methodology

In [PSG06] I have analyzed a large number of text-only real-world IT service contracts. An SLA typically consists of (1) a static part consisting of e.g., the involved parties, the contract validity period, the functional service definitions and (2) a more or less dynamic part with the QoS definitions stated in terms of SLA rules specifying service level guarantees and appropriated actions to be taken if a contract violation has been detected according to measured performance values. Various types of complex rules can be found which need to be represented, monitored and enforced by IT service providers such as graduate rules, dependent rules, reactive rules, normative rules, default rules, exception rules. To represent rules there are different possibilities, e.g., if-then constructs in procedural programming languages such as Java or C/C++ (with control flow), decision tables/trees, truth-functional constructs based on material implication, implications with constraints (e.g., OCL), triggers and effectors (e.g., SQL trigger) or logical knowledge representation (KR) approaches based on subsets of first order predicate logic such as logic programming (LP) techniques.

In this dissertation I propose a declarative rule-based KR framework for SLA and policy representation and management. I will show the use of logic programming as an appropriate declarative rule language for SLA representation enabling declarative programming of SLA rules and making inferences with these rules. Whereas, existing approaches to SLA representation are based on procedural or simple implicational truth logic, I draw on logic programming (LP) and related FOL based knowledge representation (KR) concepts which I combine in a hybrid way with existing object-oriented programming techniques in order to exploit the benefits of both worlds. This representation approach follows the declarative "separation of concerns" principle, i.e., the contractual logic is decoupled from the application or database domain and represented in terms of explicit rules which accordingly can be much easier maintained, adapted/updated and managed:

"*Expressing information in declarative sentences is far more modular than expressing it in segments of computer programs or in tables. Sentences can be true in a much wider context than specific programs can be used. The supplier of a fact does not have to understand much about how the receiver functions or how or whether the receiver will use it. The same fact can be used for many purposes, because the logical consequences of collections of facts can be available.*" [McC59]

The main advantages of using logic programming for SLA rule representation are:

- Natural and widely-accepted for representing and automated reasoning with rules
- Theoretically well-understood and uncontroversial for several logic classes with mature theory and technology
- Compact and declarative representation, no need to implement control flow as in imperative languages
- Operational reading and declarative reading with well-defined semantics (e.g., well-founded semantics, stable model semantics)
- Rich KR expressiveness and automated inference power (rule chaining via resolution + unification, variables, quantification, (non-monotonic) negation, closed world assumption)
- Traceability and verifiability due to well-defined logical semantics
- Efficient complexity results (polynomial data complexity under common restrictions)
- Highly flexible and adaptable enabling combination with other first order logic based formalisms such as ontologies (description logics) and practical extensions, e.g., type systems, procedural attachments.
- Widely implemented and deployed, e.g., Prolog derivatives, deductive databases, rule engines

Logic programming allows for a compact representation of SLA rules and for automated rule chaining by resolution and variable unification, which alleviates the burden of having to implement extensive control flows as in imperative programming languages and allows for easy extensibility in a declarative programming style. Moreover, the declarative semantics as a subset of classical first-order logic ensures correctness and traceability of produced results. However, further expressive logical formalisms than standard Horn clauses are needed for adequate, automated SLA management. Different types of rules such as derivation rules, integrity constraints, and reaction rules are needed in combination with modalities and rule qualifications in order to express deontic assignments, rule priorities, scopes, dynamic modular updates and non-monotonic features such as default and explicit negations.

To deal with the practical requirements in IT SLM a SLA rule language should also pay special attention to existing enterprise applications, IT management solutions, tools and implementations such as SQL based databases/data warehouses, EJBs, ESBs, Web Services or system and network management tools. Their optimized and specialized functionalities should be directly integrated into the SLA rules' execution by calling their procedural APIs or exchanging messages with their communication interfaces via common message protocols

such as JMS or SOAP. Accordingly, a SLA language should adopt practical language constructs such as procedural attachments and integrate external object-centered type systems as well as Semantic Web ontologies / meta data vocabularies used in the modern Internet. It should take account of the distributed environment and provide sophisticated management support for bundling rules to rule sets, managing and maintaining them during runtime as modules including updates and enabling interchange and translation into different target formats (execution syntax, mark-up syntax, human-oriented syntax). This also amount for a stable and highly scalable middleware to manage distributed contract rule bases and communicate and interchange complex event requests and answers between the systems using a common interchange format and typical transport protocols (e.g. HTTP, JMS, SOAP).

My work follows a constructivist, SE-oriented methodology where I adopt the Design Science Research approach, as described in Hevner et al. [HMPR04]. To satisfy the requirements I propose a highly expressive, efficient and general KR framework, called ContractLog, and a rule based XML markup language, called RBSLA, as a new design artifact combining adequate KR concepts and selected logical formalisms which uses extended logic programs [LW92] as its basis. Based on meta-programming techniques the KR implements adequate logical formalisms such as integrity constraints, event / action logics, deontic logic or defeasible rules. Essential non-classical inference features for external information processing and constructing views on external data (e.g., SQL queries on relational data), integration of external vocabularies (e.g., Semantic Web ontologies) or procedural code (Java objects and methods) are crucial, practical extensions of the underlying rule language and rule engine. For the distributed management and interchange RBSLA is used as an XML-based interchange format and an enterprise service bus (ESB) is used as communication middleware between the RBSLM (Rule-Based Service Level Management) services and external components. Event messages are transported by the ESB based on a broad spectrum of selectable transport protocols such as HTTP, JMS, Web Service protocols (SOAP) or agent communication languages (JADE). With my alternative rule-based approach, exploiting logic programming in a hybrid combination with object-oriented, relational and ontological concepts, I provide new levels of flexibility and automation which are not available in the current technologies and tools in the SLA domain. I try to overcome real-world problems which are of high relevance and importance for SLA representation such as rapidly changing, highly-distributed and loosely coupled service oriented environments, slow contractual change cycles, and support for new business models (on-demand, utility computing) as well as several new regulations and laws with compliance rules (Sarbanes Oxley, Basel II etc.).

While most research on logic-based knowledge representation have been of the more theoretical sort, this dissertation is more on the application-oriented utilization and implementation of selected logical formalisms for SLA representation. My practical system development effort made it clear that moving a KR idea into real practice is not just a matter of designing a new KR theory and investigation of theoretical results, but needs significant research to seriously

implement adequate logics in a coherent LP framework. It turns out that there is a long and difficult road to travel from the pristine clarity of an initial logical formalism to a practical system that really works. To evaluate the adequacy of my rule-based approach, in particular wrt to expressiveness and computational efficiency, I use established methodologies in SE, KR and LP such as theoretical worst-case analysis, experimental performance benchmark tests and real-world SLA simulations as well as use cases implementations in proof-of-concept settings which have been submitted to open-source projects (Mandarax, Prova, RBSLA) and standardization initiatives (RuleML, W3C Rule Interchange Format RIF Use Cases). Anticipating the results: the rule-based approach fulfils both primary demands of Design Science, namely relevance and rigor and there is large evidence that the core research question, whether KR techniques with logic programming on their basis can be used to adequately represent and automate SLAs, can be answered positively. In accordance with my constructivistic, implementation oriented design science approach, the focus in this dissertation is more on the implementation, utilization and practical analysis of different sorts of logics for the representation and enforcement of various SLA rules, since a comprehensive and pure theoretical logical proof of such a complex KR combining so many logical formalisms is practically hardly feasible and the theoretical results would be still questionable from the point of view of the real-world application domain.

1.4 Main Contributions

This dissertation contributes with a declarative, compact, efficient and highly expressive KR proposal, called ContractLog, for formalizing SLA specifications, which has the following main features:

- a configurable distributed and web-based inference service
  (1) built on top of an enterprise service bus (Mule [Mul06]) used as a scalable, highly distributable object broker and communication middleware,
  (2) integrated into a derivation rule engine, called Prova [KPS06], with selectable semantics variants reaching from standard SLDNF resolution and different variants to evaluations of well-founded semantics for generalized and extended LPs based on sound and complete linear resolution semantics with (sub-)goal memoization, loop-prevention and highly efficient memory structures.
- a testing methodology for logic programming adopting SE extreme programming techniques to verify and validate the correctness of ContractLog based SLA formalizations.
- a typed logic with a polymorphic order-sorted typed unification as operational semantics supporting Semantic Web ontology languages such as RDFS or OWL and Java class hierarchies with expressive procedural attachments (dynamic Java object instantiations and variable bindings enabling calls to externa boolean and object-valued Java methods).

- an active Event-Condition-Action rule interpreter which combines reactive rules and derivation rules and supports parallel execution of extended ECA rules, (transactional) ID-based updates and active rules, sensing and triggering of external functions/systems via procedural attachments and complex interval-based event /action processing via an event calculus based event/action algebra.
- an interval-based Event Calculus variant with support for temporal reasoning about transient and non-transient events and their effects on the (dynamic) knowledge systems.
- a temporal deontic logic for deriving role-based, temporal norms such as permissions, prohibitions, obligations as well as exceptions and (contrary-to-duty) violations
- support for integrity constraints and ID-based knowledge updates which enables scoped reasoning, transactional updates with validation against constraints and ID-based (bulk) updates of new knowledge, (external) modules, i.e., revision/updating and modularity of ID based rule sets.
- support for rule interchange via attached test cases, meta test suits for verification of inference engines and expressive integrity constraints
- defeasible reasoning on prioritized logic programs in combination with integrity constraints to handle un-known and conflicting knowledge and define default, exceptional, priority relations between rules and complete rule sets (modules) in order to overcome rule conflicts.
- a meta data annotated labelled logic to annotate rules with meta data such as rule names, module names, Dublin Core annotations etc. and bundle rules and facts to modules (rule / clause sets) which are managed by their module IDs and which can be used to explicitly close open environments such as the Semantic Web for scoped reasoning on parts of the distributed knowledge.
- a rich library of useful predicate functions for mathematical, date, time and interval based functions and operators.

The ContractLog KR is mainly written as a collection of LP scripts implementing the respective formalisms and meta programs, which can be easily imported on a per-need basis to the knowledge base and transferred to different rule systems. Remarkably, the carefully selected and implemented formalisms in the ContractLog KR, although primarily designed for SLA representation, also qualify to be an appropriate tool for declaratively implementing Semantic Web applications in general.

The second major contribution of this dissertation is a declarative Rule Based SLA (RBSLA) mark-up language provided as modular XML schema and a EBNF syntax description. It is implemented as an extension to the emerging Semantic Web rule standard, the Rule Markup Language (RuleML) [WTB03] in order to address interoperability with other rule languages and tool support.

That is, in contrast to existing XML markup languages in the SLA domain such as WSLA [DDK+04], WSOL [TPP+03], WS-Agreement [ACD+05], RBSLA is not a pure syntactical specification language with predefined language constructs, but is a declarative rule-based programming language with an operational and declarative logic based semantics for formalizing and implementing arbitrary contract related functionalities and specifications. This declarative implementation oriented design provides maximum flexibility and extensibility for SLA representation enabling the use of different contract vocabularies written as Semantic Web ontologies such as WSMO [RKL+05], WS-Policy Ontology [PKH05, VAG05], OWL-S [OS03] or KAoS [JCJ+03] or other ontologies such as OWL time [PH04]. The external vocabularies can be integrated into the logical SLA rules as external type systems, giving them a domain-specific meaning. RBSLA adds additional modelling power and expressiveness to RuleML to specify higher-level policies and SLAs. In a nutshell, it adds the following features to RuleML:

- typed logic constructs with "webized" external types (e.g., OWL ontologies) and input / output modes
- labelled logic constructs with rule sets and module object ids and meta data annotations
- meta-data scoped queries/goals
- procedural attachments, i.e., calls to external procedural functions
- integration of external data sources / facts and imports of external modules (external URI-based rule bases)
- Reaction RuleML: a sublanguage for reactive rules with complex events and action language constructs, complex event/action processing with interval based event / action algebra constructs and state changes a la event calculus
- ID-based update primitives and module support with "need-to-know principle"
- deontic norm constructs for describing normative rules
- defeasible rules and rule / module priorities
- various formula for computations, equalities, aggregation, lists etc.
- syntax for test cases and integrity constraints for verification, validation and integrity testing of rule bases (V&V&I)

The RBSLA language simplifies machine-processing and interchanging SLAs / policies by means of an interchangeable mark-up language for serializing SLA / policy rules and facilitates the reuse of existing XML tools and ontological webized ontologies. It provides means for optimizing/refactoring and validation of rule sets during the transformation into an executable rule language, e.g., into the ContractLog KR via a XSLT based transformations.

Based on ContractLog and RBSLA I have implemented a rule based Service Level Management (RBSLM) tool which contributes with

- integration of Prova [KPS06] as execution environment for rule-based SLAs
- integration of the Mule ESB [Mul06] as a scalable, highly distributable object broker and communication middleware to manage, integrate and seamlessly handle the interactions between distributed rule-based SLM services using disparate transport and messaging technologies
- a graphical user interface (GUI) for collaborative and role-centred developing and deploying SLAs for IT services in distributed environments such as the Semantic Web.

In summary the major advantages of my rule-based approach to SLA representation, management and enforcement are:

- it allows to continuously adapt SLAs to a rapidly changing business environment, and overcomes the restricting nature of slow change cycles
- it lowers the cost incurred in the modification of contractual / business logic
- it shortens development time of SLA specifications and safeguards the engineering process
- rules are externalized and easily shared among multiple applications and contractual partners
- changes can be made faster and with less risk
- it provides a more human way to describe, manage and maintain formalized SLAs as compared to implicit procedural implementations
- it provides highly correct, reliable and traceable results derived by generic inference engines with logical semantics based on a solid mathematical basis
- it allows to declaratively program highly sophisticated SLA logic and business management policies and provides means to integrate external functionalities and ontologies

Some of the material and contributions presented in this dissertation have been published and presented elsewhere. Selected refereed publications are:

- Paschke, A., Kozlenkov, A., Boley, H.: A Homogenous Reaction Rules Language for Complex Event Processing, International Workshop on Event Drive Architecture for Complex Event Process (EDA-PS 2007) at VLDB'07, Vienna, Austria, 2007.
- Paschke, A. and Bichler, M.: Knowledge Representation Concepts for Automated SLA Management, Int. Journal of Decision Support Systems (DSS), submitted June 2006, revised January 2007.

- Paschke, A.: Reaction RuleML Tutorial, Int. Conf. of Rule Markup Languages (RuleML'06), Athens, Georgia, USA, 2006, available at: http://2006.ruleml.org/slides/reaction-ruleml.pdf.
- Paschke, A., Dietrich, J., Giurca, A., Wagner, G., Lukichev, S.: On Self-Validating Rule Bases, Int. Semantic Web Enabled Software Engineering Workshop (SWESE'06 at ISWC'06), Athens, Georgia, USA.
- Paschke, A.: A Typed Hybrid Description Logic Programming Language with Polymorphic Order-Sorted DL-Typed Unification for Semantic Web Type Systems, OWL-2006 (OWLED'06), Athens, Georgia, USA, 2006.
- Paschke, A.: Verification, Validation and Integrity of Distributed and Interchanged Rule Based Policies and Contracts in the Semantic Web, Int. Semantic Web and Policy Workshop (SWPW at ISWC 06), Athens, Georgia, USA, 2006.
- Paschke, A.: ECA-RuleML/ECA-LP: A Homogeneous Event-Condition-Action Logic Programming Language, Int. Conf. of Rule Markup Languages (RuleML'06), Athens, Georgia, USA, 2006.
- Paschke, A.: The Mandarax RDF / RDFS/ OWL / DLP Module - Integration of Semantic Web Data into the Rule Engine Manadarax. International Workshop on Rule-Based Modeling and Simulation of Interacting Systems and Agents (AORML), Cottbus, Germany, Feb. 2006.
- Paschke, A.: The Rule Based Service Level Agreement Project (RBSLA). International Workshop on Rule-Based Modelling and Simulation of Interacting Systems and Agents (AORML), Cottbus, Germany, Feb. 2006.
- Paschke, A.: RBSLA - A declarative Rule-based Service Level Agreement Language based on RuleML, International Conference on Intelligent Agents, Web Technology and Internet Commerce (IAWTIC 2005), Vienna, Austria, 2005.
- Paschke, A., Bichler, M., Dietrich, J.: ContractLog: An Approach to Rule Based Monitoring and Execution of Service Level Agreements, International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML 2005), Galway, Ireland, 2005.
- Paschke, A., Dietrich, J., Kuhla, K.: A Logic Based SLA Management Framework, Semantic Web and Policy Workshop (SWPW), 4th Semantic Web Conference (ISWC 2005), Galway, Ireland, 2005.
- Jens Dietrich and Adrian Paschke, On the Test-Driven Development and Validation of Business Rules, 4th International Conference on Information Systems Technology and its Applications (ISTA 2005), New Zealand, May 2005.
- Paschke, A., Bichler, M.: SLA Representation, Management and Enforcement - Combining Event Calculus, Deontic Logic, Horn Logic and Event

Condition Action Rules, E-Technology, E-Commerce, E-Service Conference (EEE05), Hong Kong, 2005

For a complete list of all publications of the RBSLA project see:

`http://ibis.in.tum.de/research/projects/rbsla/index.php#infos`

The RBSLA project is hosted at Sourceforge:

main page: `http://ibis.in.tum.de/projects/rbsla/index.php`
download: `https://sourceforge.net/projects/rbsla`

Parts of my work in the RBSLA project have been contributed to other open source projects and standardization initiatives:

1. Sourceforge Prova - The Prova Rule Engine with ContractLog and RBSLA: http://www.prova.ws/
2. Sourceforge Mandarax - Mandarax RDF/OWL API 1.1 and Mandarax Event Calculus EC 1.0: http://sourceforge.net/project/showfiles.php?group_id=50817
3. Reaction RuleML - The Reaction Rules Markup Language is evolved from the RBSLA langauge: http://ibis.in.tum.de/research/ReactionRuleML/
4. RuleML - Various contributions to RuleML: validators, translators (XSLT), GUI-based editors, language extensions etc.
5. W3C Rule Markup Language Initiative(RIF): Use Cases for the W3C RIF WG: http://www.w3.org/2005/rules/wg/wiki/Use_Cases

1.5 Thesis Organization

The remainder of this dissertation is organized as follows:

Chapter 2 gives an insight into IT Service Management (ITSM) and relevant ITSM standard such as ITIL. It provides the relevant background in Service Oriented Computing (SOC) and IT Service Level Management and analyzes different types of service contracts, presents the main component parts and defines the relevant terminologies in order to reach a common understanding. A use case based on real-world industrial SLAs is introduced and related work is discussed.

Chapter 3 recalls the basics in rule-based knowledge representation based on first order logic and in particular the two relevant subsets, namely logic programming extended with non-monotonic features and description logics. It also gives an overview on knowledge representation in the context of reactive rule and event/action processing and defines relevant concepts in this domain.

Finally, it further details the requirements for a declarative, logic-based contract representation language.

Chapter 4 elaborates on the ContractLog KR which is an expressive and computational efficient KR framework consisting of selected, adequate logical knowledge representation concepts for the formalization and automated execution of electronic contracts such as SLAs or higher-level policies. It combines selected logical formalisms which are mainly implemented on the basis of declarative logic programs and meta programming techniques.

Chapter 5 describes the superimposed Rule Based Service Level Agreement language (RBSLA) which is a mark-up language to serialize rule-based policy- and contract specifications such as SLAs in XML. It is implemented as an extension to the emerging XML-based Rule Markup language (RuleML v. 0.91) and Reaction RuleML (Reaction RuleML v. 0.1) in order to address interoperability with other rule languages, XML-based persistent serialization and tool support.

Chapter 6 introduces the superimposed rule-based service level management tool (RBSLM) which serves as a proof-of-concept implementation for my rule-based SLA management approach. The RBSLM tool divides into the contract manager and the service dashboard. The contract manager is an engineering tool for SLA rule bases written either in RBSLA or directly as ContractLog scripts. IT provides support by means of template repositories and test suites for different roles such as business practitioners and domain experts. The service dash board is a runtime environment to monitor, execute and visualize service contracts. ContractLog/Prova is used for the distributed inference services (rule engine) and Mule as ESB middleware infrastructure.

Chapter 7 presents theoretical and experimental evaluations and illustrates the rule-based formalization of SLAs rules in the ContractLog KR based on a real-world use case example. It discusses adequacy of the design artifacts, namely the ContractLog KR, the RBSLA markup language, for SLA representation, management and enforcement, and the RBSLM prototype.

Chapter 8 concludes with a summary of the dissertation and a discussion of practical implications.

2 IT Service Level Management

This chapter gives an insight into IT Service Management and the relevant concepts, practices and technologies in Service Oriented Computing and IT Service Level Management. It categorizes different types of IT service contracts, presents the main component parts and defines the goals in order to reach a common understanding. An use case is presented and related work is discussed.

2.1 IT Service Management

IT Service Management (ITSM) describes the change of information technology (IT) towards service and customer orientation. IT Service Management is commonly defined [You04] as:

Definition 1 *(**IT Service Management**) a set of processes that cooperate to ensure the quality of IT services, according to the levels of service agreed to by the customer. It is superimposed on management domains such as systems management, network management, systems development, and on many process domains like change management, asset management and problem management.*

Software producers often subsume different extensions to the IT infrastructure and system management by a service layer under the term ITSM. [BDF+04] According to [HP03, Pet03] ITSM should be distinguished from IT infrastructure management (ITIM) and IT business value management (IT Governance [Gre04]):

Definition 2 *(**IT Infrastructure Management, IT Service Management and IT Governance**)*

IT infrastructure management (ITIM) focuses on optimizing the management of the infrastructure, i.e., the devices it contains and the data it creates.

IT service management (ITSM) focuses on planning and efficient and effective delivering of IT services and products while meeting availability, performance, and security requirements. Of importance are the management, monitoring and enforcement of service-level agreements, both internally and externally, to meet agreed-upon quality and cost targets.

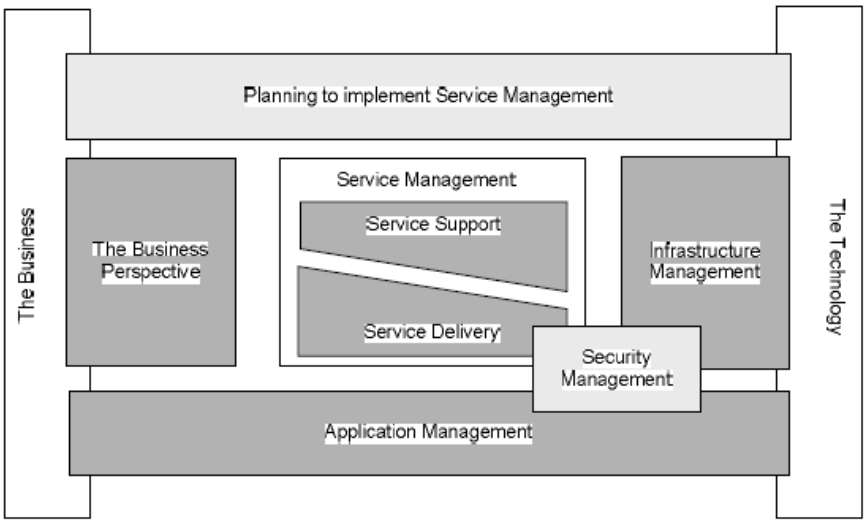

Figure 2.1: ITIL Service Management Processes [Sal04]

IT Governance [Gre04] (a.k.a. IT business value management related to strategic alignment) is an integral part of enterprise governance and consists of the leadership and organizational structures and processes that ensure that the organization's IT sustains and extends the organization's strategy and objectives.

In this dissertation, I mainly focus on ITSM. But, remarkably, ITIM applications such as system and network management tools typically provide measurement and monitoring results which are aggregated to service level states which underpin ITSM and SLA enforcement. IT Governance then draws on the results from ITSM and in particular from IT Service Level Management (see ITIL section 2.1.1) and SLA monitoring and enforcement.

[Sal04] reviews different open and industrial frameworks for IT governance (CobiT [ITG98]) and IT management (ITIL [itS04a],HP ITSM [HP03], Microsoft MOF [Mic02]. At present, especially the de-facto standard ITIL together with the newly released standard BS 15000 receives great attention of IT management. In this dissertation I will mainly focus my attention on a particular process of the ITIL Service Delivery domain, namely Service Level Management.

2.1.1 ITIL

The *Information Technology Infrastructure Library* (ITIL) [OGC00a] is a documentation of IT management concepts, processes and methods supported and provided by the international IT-Service-Management-Forum [ITS04b] as a series of books. It is organized into the business perspective, application management, service delivery, service support, and infrastructure management supplemented with security management, as shown in figure 2.1 [Sal04].

The core of the frameworks is the *IT Service Management* (ITSM) which is concerned with the management and monitoring of IT services [Ker98]. ITIL describes ITSM as a set of strategic, tactical and operational practices, processes and methods. ITIL's ITSM is divided into the domains *Service Support* [OGC00b] and *Service Delivery* [OGC01]. Service Support is divided into Incident Mgt., Problem Mgt., Change Mgt., Release Mgt., and Configuration Mgt. Service Delivery is divided into Service Level Mgt., Financial Mgt., IT-Service Continuity Mgt., Capacity Mgt., Availability Mgt. Table 2.1 gives a high level overview.

Table 2.1: The ITIL processes

Description	Position	Task
Service Desk	Function	Group of specialists, inquiry -, treatment of disturbances
Incident Management	Process	Support user, problem acceptance, assistance, monitoring service level
Problem Management	Process	Treatment of losses, cause identifying, recommendations at Change Mgmt., improvement of productive resources use
Configuration Management	Process	Process control of the inventory (components hard -, software....)
Change Management	Process	Change process
SLM	Process	Formulate and enforce SLA
Release Management	Process	Storage of authorized software, release in productive environment, distribution to remote bases, implementation to start-up
Capacity Management	Process	Correct and cost-related-justifiable IT capacity provision analysis, prognosis; Capacity plans
Availability Management	Process	Optimization IT resources use, foreseeing and calculation of losses, safety guidelines to monitor SLAs, Security, Serviceability, Reliability, Maintainability, Resilience
Service-Continuity-Management	Process	Re-establishment of services, replacement in case of failure
Financial Management	Process	Process investment strategy, definition that-achievement-aims, those-brought achievement to measurement

In section 2.3 I will focus on Service Level Management which is of highly practical relevance for all other processes such as Availability Mgt. or Incident

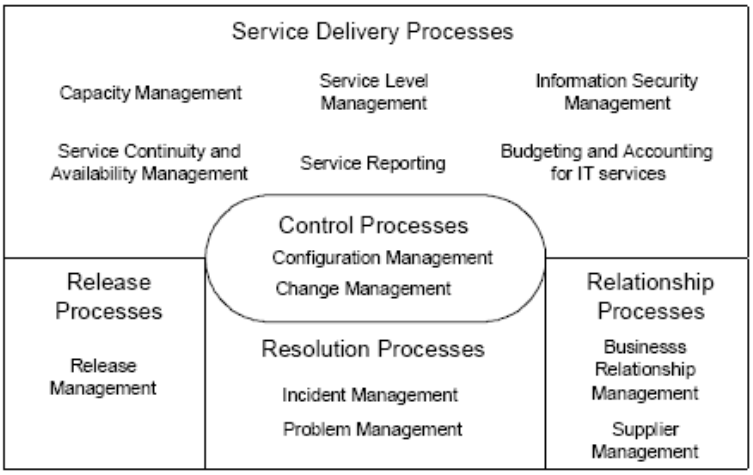

Figure 2.2: BS15000 Service Management Processes [Sal04]

Mgt. in ITIL. For a detailed description of the other domains and processes see e.g., [HC02, OGC00b, OGC01].

2.1.2 BS15000

With BS 15000 a standard which implements the particular IT service management processes of ITIL exists. BS1500 was published in November 2000 by the British Standards Institutes (BSI) [BSI02]. It addresses the effective management and implementation of IT service by an integrated process approach and defines high level requirements for an ITSM system that also includes policies. The IT processes are organized into five categories as presented in figure 2.2 [Sal04].

BS15000 gives recommendations for planning, implementing, monitoring and reviewing/continous improve service management. BS15000 in cooperation with ITIL will be integrated into the ISO-9000 standard.

2.2 Service Oriented Computing

The computing paradigm that utilizes services for developing applications in open distributed environments such as the Web is known as Service Oriented Computing (SOC). The vision is to build large scale service supply chains (a.k.a. business services networks) which enable enterprizes to define and execute web services based transactions and business processes across multiple business entities and domain boundaries using standardized (web) protocols. This idea is not completely new, but evolves from earlier approaches such as multi-agent systems (MAS), heterogenous information systems (HIS), enterprise application integration (EAI) and technologies such J2EE, DCOM or Corba. But, web services,

which emerged in the last time as the prevailing technology for implementing IT services on the web, have received a great commitment from industry and research. In the following I will introduce the major web services related terminology and concepts; although, it should be noted that the work presented in this dissertation is not restricted to web services but applies to IT services and contracts / policies for IT services in general. For details I refer to the overwhelming amount of literature about service-oriented computing and web services. Most web service related specifications are standardized at OASIS and the W3C.

According to [SOA06] the term service-oriented architecture (SOA) expresses a perspective of software architecture that defines the use of loosely coupled software services to support the requirements of the business processes and software users. In an SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation. A SOA is not tied to a specific technology. It may be implemented using a wide range of technologies, including REST, RPC, DCOM, CORBA or Web Services. Web services can be used to implement a Service Oriented Architecture. Service Component Architecture (SCA) is a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture. Service Component Architectures (SCAs) extend and complement this approaches to implementing services. That is, SCA builds on open standards such as Web services. The W3C defines a Web service as [Web04]

Definition 3 *(**Web Service**) A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

The W3C Web service definition encompasses many different systems, but in common usage the term refers to those services that use SOAP-formatted XML envelopes and have their interfaces described by WSDL:

- SOAP: An XML-based, extensible message envelope format, with "bindings" to underlying protocols (e.g., HTTP, SMTP and XMPP).
- WSDL: An XML format that allows service interfaces to be described, along with the details of their bindings to specific protocols. Typically used to generate server and client code, and for configuration.
- UDDI: A protocol for publishing and discovering metadata about Web services, to enable applications to find Web services, either at design time or runtime.

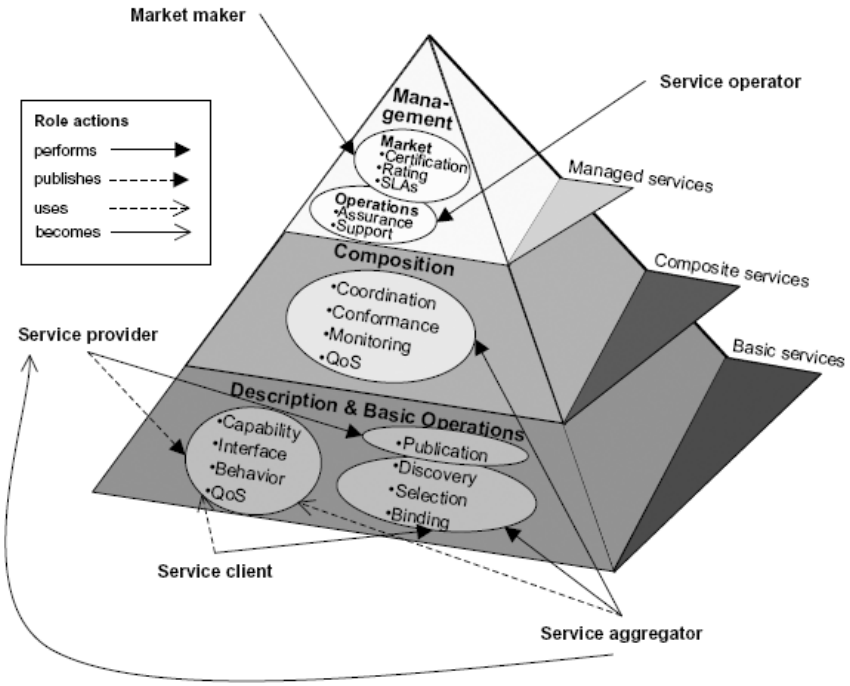

Figure 2.3: Extended Service Oriented Architecture [PG03]

Recently semantic web services (SWS) which provide an approach for representing the functionality of Web Services with the help of Semantic Web ontologies and BPEL (business process execution language) for web service choreography attracts much attention from industry and research (see section 2.5).

OASIS (the Organization for the Advancement of Structured Information Standards) defines SOA as the following [Or06]:

Definition 4 *(**Service Oriented Architecture**) A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.*

An extended SOA for SOC has been proposed in [PG03] - see figure 2.3

While large parts of the description / basic operators and the composition layer have been already standardized and provide rich tool support the management layer in this extended SOA still comprises a lot of open challenges and research questions. In this dissertation I contribute to this layer with an adequate representation and management artifact for service contracts such as Service Level Agreements or policies. SLAs are crucial in all phases of the service life cycle:

1. design and search/negotiation phase, where SLAs contribute e.g., to find / negotiate appropriate services in the open Web which provide acceptable service levels and quality of service (QoS) guarantees;
2. in the monitoring and enforcement phase where they are used monitor the service levels at runtime, enforce the agreed rights and obligations and provision the service resources according to the measured values and the agreed prices and penalties (or bonuses)
3. in the analysis phase where the SLAs and the services are redefined and renegotiated according to the results in the monitoring and enforcement phase. This process of design / implementation, monitoring/enforcement and redefinition of SLA is often subsumed as Service Level Management or SLA Management.

2.3 Service Level Management and Service Level Agreements

According to ITIL *Service Level Management* (SLM) ensures continual identification, monitoring and reviewing of the optimally agreed service levels of IT services as required by the business. That is, SLM deals with defining, monitoring and enforcing of Service Level Agreements (SLAs). SLM is central in ITIL Service Delivery and an essential precondition for all other processes in ITIL. The high level activities of the SLM process are [Sup01]: establish services/-functions with service levels which can be measured and monitored, implement SLAs, manage/monitor/enforce ongoing processes, review periodically, continuously refine and optimize the service levels. SLM has close connections to the operational processes in ITIL such as incident or problem management.

Definition 5 ***(Service Level Agreement)*** *A SLA is a contract that documents the goals and responsibilities, the service level indicators (SLIs), the service level objectives (SLOs), and the applicable rewards and penalties.*

A SLA is an essential component of the legal contract between a service consumer and the provider. They help to assure the total system costs and improve relationships and establish trust between the service providers and consumers.

Definition 6 ***(Service Level Indicators, Service Level Objectives and Guarantees)*** *Service Level Indicators (SLIs) (a.k.a. key performance indicators) clearly and consistently indicate the performance of a service.*
Service Level Objectives (SLO) defined in terms of SLA rules represent the promises and guarantees wrt graduated high/low SLI ranges, e.g., average availability range [low: 95% , high: 99%, median: 97%], so that it can be evaluated whether the measured SLIs resp. SLA metrics exceed, meet or fall below the defined service levels at a certain time point or in a certain validity period.

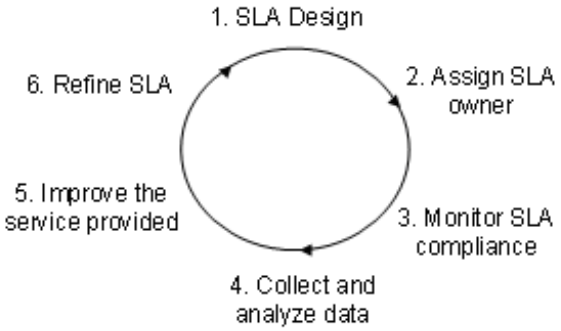

Figure 2.4: SLA Life Cycle

SLOs and guarantees can be informally represented as if-then rules expressing graduations, complex policies and conditional guarantees, e.g., conditional rights and obligation: "*If the average service availability during on month is below 95% then the service provider is obliged to pay a penalty of 20%.*".

Definition 7 ***(SLA metrics)*** *SLA metrics are used to measure the performance and quality characteristics of the service on an operational, infrastructure level. They are either retrieved directly from the managed and monitored resources such as middleware, servers or instrumented applications or are created by aggregating such direct metrics into higher level composite metrics.*

Typical examples of direct metrics are e.g., the MIB variables of the IETF Structure of Management Information (SMI) [MIB04] such as number of invocations, system uptime, outage period or technical network performance metrics such as loss, delay, utilization etc. which are collected via measurement directives such as management interfaces, protocol messages, URIs etc. Composite metrics use a specific function averaging one or more metrics over a specific amount of time, e.g., average availability, or breaking them down according to certain criteria, e.g., maximum response time, minimum throughput, top 5% etc.

In an industrial study with close to three dozen IT service providers from small-and medium-sized enterprises to big companies I have analyzed nearly fifty state-of-the-art SLAs currently used throughout the industry in the areas of IT outsourcing, Application Service Provision-ing (ASP), Hardware Hosting, IT Service Suppliers. Appendix E shows the develop categorization scheme for IT metrics. [PSG06]

According to the Hurwitz Group [SMJ00] the life cycle of a SLA is defined as shown in figure 2.4.

The rule-based SLA representation approach proposed in this dissertation contributes in all phases of the SLA life cycle. Remarkably, due to the formalized, machine-readable and semantically understandable representation SLAs

can be directly used in automated service discovery and negotiation (see e.g., [Pas05d] for an overview of different coordination and negotiation mechanisms). However, a particular focus is on the runtime management, maintenance and enforcement phase, because here (from the view of a service provider) the cost savings due to the automation of SLAs and the flexibility wrt to dynamic changes will be highest, since it is well-known in software projects that maintenance makes up to 40% of the overall costs and is often much higher than the development costs.

The objectives of SLAs are manifold. In a nutshell the substantial goals are: [Pas04b]

- Verifiable, objective agreements
- Know risk distribution
- Trust and reduction of opportunistic behavior
- Fixed rights and obligations
- Support of short and long term planning and further SLM processes
- Decision Support: Quality signal (e.g., assessment of new market participants)
- Guaranteed quality of service

According to their intended purpose, their scope of application or their versatility SLAs can be grouped into different (contract) categories - see table 2.2.

A particular service contract might belong to more than one category, e.g., an Operation Level Agreement (OLA) might also be an individual in-house agreement. Several service contracts can be organized in a unitized structure according to a taxonomical hierarchy - see figure 2.5.

Service Level Agreements come in several varieties and comprise different technical, organizational or legal components. Table 2.3 lists some typical contents.

Although the characteristics and clauses may differ considerably among different contracts, they all include more or less static parts such as the involved parties, the contract validity period, the service definitions but also dynamic parts which are more likely to change, such as the QoS definitions stated as SLA rules specifying service level objectives and guarantees e.g., in terms of appropriated actions to be taken if a contract violation has been detected. The representation of the static part of an SLA is straightforward. From the point

Table 2.2: SLA categorization

Intended Purpose	
Basic Agreement	defines the general framework for the contractual relationship and is the basis for all subsequent SLAs.
Service Agreement	Subsumes all components which apply to several subordinated SLAs.
Service Level Agreement	Normal Service Level Agreement
Operation Level Agreement (OLA)	A contract with internal operational partners, which are needed to fulfil a superior SLA.
Underpinning Contract (UC)	A contract with external operational partner, which are needed to fulfil a superior SLA.
Scope of Application	(according to [Binder2001])
Internal Agreement	Rather an informal agreement than a legal contract
In-House Agreement	Between internal department or divisions
External Agreement	Between the service provider and an external service consumer
Multi-tiered Agreement	Including third parties up to a multitude of parties
Versatility	(according to [Binder2001])
Standard Agreement	Standard contract without special agreements
Extensible Agreement	Standard contract with additional specific agreements
Individual Agreement	Customized, individual agreements
Flexible Agreement	Mixture of standard and individual contract

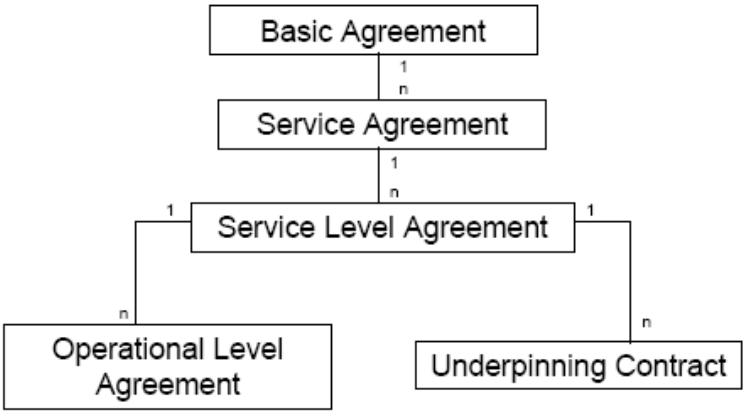

Figure 2.5: Hierarchical Contract Structure

of view of a rule-based contract logic they can be easily represented as facts managed e.g., in the knowledge base or an external database or a Semantic Web ontology which can be integrated into the rule execution at runtime. In this dissertation I focus on the dynamic part of an SLA - the SLA rules - which imposes high requirements regarding flexibility, expressiveness, verifiability and traceability, interoperability and automation on the representation language.

2.4 Use Case Example

In order to better illustrate the requirements of SLA management I will describe an use case derived from real-world SLA examples from an industry partner. The SLA defines three monitoring schedules, "*Prime*", "*Standard*" and "*Maintenance*" as shown in table 2.4

During prime time the average availability has a low value of 98%, a median of 99% and a high value of 100% and a response time which must be below 4 seconds. The service metrics are calculated via a ping every 10 seconds. During standard time the average availability is {high:99%;low:95%;median:97%} and

Table 2.3: Categorization of SLA contents

Technical Components	Organizational Components	Legal Components
- Service Description	- Liability and liability limitations	- Obligations to co-operate
- Service Objects	- Level of escalation	- Legal responsibilities
- SLA/QoS Parameter	- Maintenance / Service periods	- Proprietary rights
- Metrics	- Monitoring and Reporting	- Modes of invoicing and payment
- Actions	- Change Management	
...	...	...

Table 2.4: Monitoring schedules

Schedule	Time	Availability	Response Time
Prime	8 a.m. - 18 p.m.	98%[99%]100%; pinged every 10 sec.	4 sec.; pinged every 10s
Standard	18 p.m. - 8 a.m.	95%[97%]99%; pinged every min.	10[14]16 sec.; pinged every min.
Maintenance	0 a.m. - 4 a.m.*	20%[50%]80%; pinged every 10 min	No monitoring

response time {high:10sec.;low:16sec.;median:14sec.} monitored via a ping every minute. Maintenance is permitted to take place between midnight and 4 a.m. During this period the average availability is {high:80%; low:20%; median:50%} monitored every 10 minutes. Response time will not be monitored in this case.

Further the SLA defines a "bonus-malus" policy as shown in table 2.5.

Table 2.5: Bonus malus price policy

Price	Base	Malus	Bonus
Prime	p_{prime}	$p_{prime} + p_{prime} * x_{bonus}\%$	$p_{prime} - p_{prime} * x_{malus}\%$
Standard	$p_{standard}$	$p_{standard} + p_{standard} * x_{bonus}\%$	$p_{standard} - p_{standard} * x_{malus}\%$
Maintenance	$p_{maintenance}$	$p_{maintenance} + p_{maintenance} * x_{bonus}\%$	$p_{maintenance} - p_{maintenance} * x_{malus}\%$
Incident Penalty	$p_{incidentn}$		

According to the monitoring schedules a differentiated base price is defined if the service levels are met. If the service levels are exceeded (median to high) a dependent bonus is added and if they fall below the agreed upon service levels (median to low) a discount is deducted. The bonus and malus are defined as a percentage value of the base price. If a service level is missed, i.e., the actual value falls below the low service level (<low) an additional penalty has to be paid which increases exponentially with the number of incidents during the accounting period.

In case of outages/incidents the SLA defines two escalation levels - see table 2.6

Table 2.6: Escalation levels with role models and associated rights and obligations

Level	Role	Time-to-Repair (TTR)	Rights/Obligations
1	Process Manager	10 Min.	Start / Stop Service
2	Quality Manager	Max. Time-to-Repair (MTTR)	Change Service Levels

Each escalation level defines clear responsibilities in terms of associated roles which have certain rights and are obliged to do certain remedial actions in case of incidents which initiate the respective escalation level. In the SLAs' escalation level 1 the process manager is obliged to restart an unavailable service within 10 minutes. Accordingly, she has the right (permission) to start and stop the

service. If she fails to do so, escalation level 2 is triggered and the quality manager is informed. The quality manager has more rights, e.g., the right (permission) to adapt/change the SLA management systems respectively the service levels. The quality manager might discuss the time needed to repair with the process manager and extend it up to a maximum time to repair level (change request). In case of very critical incidents the system might directly proceed to escalation level 2 and skip level 1.

2.5 Related Works

2.5.1 Commercial SLA/SLM Tools

The tendency in commercial SLA/SLM tools such as IBM Tivoli SLA, HP OpenView, CA Unicenter, BMC Patrol, Remedy Service Management is to allow standardized specifications of QoS parameters (e.g., availability, response time) with high/low bounds. Typically, these parameters are directly encoded in the application code or database tier, which makes it hard to dynamically extend the SLA logic and describe more complex conditions than simple bounds. Hence, this approach is restricted to more or less standardized, static rules with only a limited set of parameters which can be changed at runtime. More complex conditionals where parameters depend upon other parameters or conditional contract states are not expressible. For instance, a high response time level may be necessary during prime time, a lower availability might be necessary under higher server loads or at least response time must be high when availability falls below a certain value. Due to the implicit procedural encoding of the SLA logic into the application code the SLAs are hard to manage, maintain and adapt to new requirements, since this would require heavy time and cost intensive refactorings of the application code and database schemas. Often the SLM tools are only simple extensions to system- and network management tools which allow monitoring of agreed subsystems such as the network or the help desk. Integrated benchmarks with respect to the natural language defined SLA are only possible by manually joining measured values using e.g., Excel tables. Obviously this process is fault-prone, inflexible, time-consuming and costly.

2.5.2 SLA XML Markup Language

There are several XML based mark-up approaches towards syntactical specification and management languages for SLAs such as the IBM Web Service Level Agreements (WSLA) [DDK+04], the HP Web Service Management Language (WSML) [SAM01], the Web Service Offering Language (WSOL) [TPP+03], SLAng [LSE03], WS-QoS [TGN+03] or the WS-Agreement language [ACD+05]. These languages include syntactical XML mark-up definitions of the involved parties (signatory and supporting parties), references to the operational service descriptions (e.g., WSDL), the SLA parameters to be monitored and the metrics and algorithms used to compute SLA parameters from raw metrics collected

by measurement directives from external sources. They allow the specification of QoS guarantees, constraints imposed on SLA parameters and compensating activities in case these constraints are violated. Typically, these constraints and activities are expressed in terms of implicational clauses with simple Boolean-valued evaluation functions and explicit connectives such as And, OR. As a result, these rules have only a very limited expressiveness which is restricted to truth/false implications without e.g., rule chaining, variable backtracking or non-monotonic reasoning inferences, as in logic based rule languages. They are to some extend comparable to strictly ordered hierarchical if-then rules in a procedural language. But in fact, they are much less expressive, because they are missing almost any required programming language feature such as data abstraction, modularity, recursion, complex object structures etc. A formal semantics of logical rule inference is completely missing. Hence, these syntactical languages can not express more than simple qualifying conditions in the sense of material truth implications. For instance, a rule set "*if given p and if p then q and if q then r*" is only expressible as a nested truth implication rule $(p \supseteq q) \supseteq r)$, but not as an automatically chained global rule set as in LP rule languages. Obviously, this approach is not modular and large, sophisticated rule sets can not be expressed. Moreover, these languages are not based on a logical semantics but are purely syntactic formalization approaches. In particular, semantics for temporal, deontic, athletic or state/situative quantifications are missing. For instance, WS-Agreement defines an agreement life-cycle which includes the creation, termination and monitoring of agreement states. However, the notion of a state in WS-Agreement is restricted to a predefined syntactical set of states such as "not ready", "ready", "running", "finished". A semantical event and time-based state model and capabilities to define and reason over arbitrary states and state transitions as effects of occurred events or actions, as e.g., in action languages, is missing. The WS-Agreement specification does not contemplate the possibility of changing an agreement wrt to contract state tracking at runtime.

In summary, although, these languages are syntactically rich, i.e., define a great variety of SLA related constructs, they need an extra procedural interpreter which must be adapted each time new expressiveness and functionalities are introduced into the XML language. For instance, Cremona [LDK04] is a purely procedural implementation framework for the creation and monitoring of interfaces of the WS-Agreement specification. In short, these markup languages are purely syntactic, they are not generic and flexible and they typically lack a precise formal logical semantics which has several drawbacks, e.g., wrt verification and validation of correctness and semantical conflict detection and resolution. They can be used only to describe more or less "standardized" SLAs which refer to the provided vocabulary of the language but they are not useable to *declaratively program* individual SLAs with a precise semantical interpretation enabling reasoning and semantics interpretation in standardized inference engines. In short, the main contribution of these syntactical approaches is: to provide a vocabulary syntax to define terms which refer to non-functional properties of (Web) services.

2.5.3 Policy Languages and Ontology-based Languages

In the area of policy specifications there are several proposals which address the definition of policies such as WS-Policy [HK03], KAoS [JCJ+03], Rei [Rei02, KFJ05] or Ponder [DDLS01]. WS-Policy [HK03] is a general policy framework which provide a basic XML grammar for expressing the capabilities, requirements, and general characteristics of entities in a XML Web service-based systems. The details of special policy aspects are intended to be defined in specialized sublanguages. Although, there exists several extensions to WS-Policy such as GlueQoS [WTM+04] or WS-Security Policy [LDa02] the approach mainly focuses on the syntactic mark-up description of policy information and hence is limited in its ability to express rich semantic meaning. Parsia et al. [PKH05] present and OWL based representation of WS-Policy.

Ponder [DDLS01] is a declarative object-oriented language that supports the specification of several types of management and security policies for distributed object systems. It provides structuring techniques for policies to manage the complexity of policy administration in large enterprise information systems. It distinguishes obligation and authorization policies and allows for conflict detection between deontic norms. Constraints are specified via a subset of OCL (Object Constraint Language). Roughly, in OCL "p implies q" means that the truth-functional proposition "if p then q" is true if either "p" is false or "q" is true in every instantiation of the model to which the constraint applies. This kind of implication is different from logical implication which means that an implication is always true if the prerequisite propositions are true under every possible interpretation, i.e., "p implies q" means "q" is true, if "p" is true under every possible interpretation. Moreover, OCL constraints can not explicitly express (athletic) modalities or non-monotonic rules. Since Ponder specifications are compiled into Java classes and are represented as Java objects at runtime, dynamic changes to the policy at runtime are not possible.

Rei [Rei02, KFJ05] is a policy language based in OWL-Lite (although originally in version 1.0 it was based on first-order logic) that allows policies (mainly trust and privacy policies) to be specified as constraints over allowable and obligated actions on resources in the environment. Rei also includes logic-like variables giving it the flexibility to specify relations like role value maps that are not directly possible in OWL. Recently the authors discuss the combination of OWL ontologies and SWRL rules [KFJ05].

KAoS [JCJ+03] uses OWL (former DAML) as basis for representing and reasoning about policies within Web Services, Grid Computing and Multi-Agent Systems (MAS). It exploits ontologies written in the Semantic Web ontology language OWL for reasoning about domain specific models describing actors and their positive or negative authorizations and obligations together with properties such as site of enforcement, priority or update time stamps. It allows updates to the policy models during runtime and provides means to detect conflicts between policies using the subsumption mechanisms between ontology classes. However, a pure OWL approach encounters some difficulties with regard to the definition

of e.g., non-monotonic reasoning needed e.g., to express defaults or negation-as-failure or parametric constraints, which are assigned a value only at deployment or run time. To compensate some of the limitations in the expressiveness of OWL role-value maps are added to later work of KAoS.

In summary these policy related approaches mainly focus on typical operational policies such as access control or security issues and only require/consider a very limited set of logical formalisms - mostly standard deontic norm reasoning to express access rights and obligations. In particular, they are missing expressive rules such as reactive rules, defeasible rules, normative rules with full temporal deontic logic capable of handling violations and exceptions such as contrary-to-duty obligations.

2.5.4 Semantic Web Services Languages

Closely related to the ontology-based approaches in the policy domain are semantic web services (SWS). Among these approaches are OWL-S [OS03] (former DAML-S), WSDL-S [SVSM03], WSMF [FB02], SWSF [SWS05], WSMO [WSM05b], Meteor-S MWSAF [MS03]. These approaches semantically annotate service descriptions with additional meta data and concepts from ontologies which describe the functional and non-functional properties of a service in a machine-understandable format. These semantic annotations enrich the service by mapping e.g., the standard WSDL description of a service with ontologies which provide a common understanding of the properties of a Web Service and hence form the basis for automatically searching and negotiating web services in the Semantic Web. Since all these approaches use ontologies, in particular OWL-based representations, similar limitations wrt expressiveness for general SLA representations apply as discusses above for the ontology-based policy languages.

Another approach towards SWS are UDDI extensions such as UX [CLTSBS03] or UDDIe [SRAAW03]; two approaches which deal with extending the functionality of UDDI by introducing an additional server of broker which provides QoS service data. UX [CLTSBS03] is an architecture providing QoS-aware and cross organizational support for UDDI. The main idea is to rate services with reputation measured through QoS feedback in form of client reports containing response time, reliability, cost, time stamp and report number. The collected reports are used to predict the services' future performance. A similar approach based on consumer ratings is proposed by [DSGF03]. UDDIe [SRAAW03] is implemented in the context of the G-QoSM framework for grid service discovery and extends UDDI. It enables service providers to associate their services with QoS properties such as bandwidth, CPU or memory requirements. It provides search functionalities which allow quantifications over the stated QoS values such as "equalto", "lessthan" or "greaterthan". In summary, these approaches focus on service discovery using additional meta data about QoS related properties of the service. The meta data values are provided as simple name value pairs and there is no support for descriptions of SLAs.

2.5.5 Formalization Approaches

There have been some proposals on using Petri-nets [MJSSW03] or Finite State Machines [Das00] to describe contracts as process flows. A predictive QoS model for workflows involving QoS properties is proposed in [CSM+04]. Other approaches propose contractual agreements formats and infrastructure to facilitate interaction and workflow-based coordination between parties, e.g., tpaML/BPF [DDK+01] and CrossFlow [[HFGL01]. In short, these approaches are best suited for contracts which follow a pre-defined protocol sequence.

In the context of the ODP Enterprise Language a model for the representation of contractual obligations has been proposed [ISO99] along with other work on the formalization of contractual obligations and rights [MM01]. The approach enables representation of contracts a deontic constraints. However, these deontic constraints are not based on a logical semantics as in deontic logic, but exploit a temporal verification of deontic consistency via a visual mechanism based on the concept of a role window [MM01]. In [MGL+04] a contract monitoring facility based on a model for expressing behavior and policies in a representation language called Business Contract Language BCL is presented. Bhoj et al. [BSC98] describes a contract to be defined by a triple (P,M,A), where P is a set of properties, A is the set of assertions and M is the set of methods available on the contract. In [AB01] Abrahams defines the E-Commerce Application Development and Execution Environment (EDEE) - an approach for representing contracts using occurrences.

2.5.6 Logic Based SLA Languages

Only little work has been done in using rules and in particular LP-based approaches for representing SLAs and QoS policies. Chomicki et al. [CLN00] describe a declarative policy description language "PDL" in which policies are represented as event-condition-action (ECA) rules which are translated into non-recursive Horn logic programs. The approach supports action constraints which are used to define, detect and resolve conflicting actions, but mainly focuses on event action sequences. In [KBGH06] an approach towards web service discovery and composition using constraint logic programming (CLP) is proposed. The services are described using the USDL (Universal Service-Semantics Description Language), a language for formally describing the semantics of Web services. Farrell et al. [FSS+04] developed the Contract Tracking XML language (CTXML) with a computational model based on the Event Calculus in order to facilitate the automated tracking of the contract state defined as contract norms that hold between contract parties. However, their approach is restricted to the core classical event calculus and does not deal with e.g., time intervals, complex events which occur over an interval, (re)active rules such as ECA rules or other rule types such as derivation rules or integrity constraints.

Several recent works in particular in the SWS domain integrate rules into their ontology models in order to overcome expressiveness restrictions of Se-

mantic Web ontology languages such as OWL (which are semantically based on description logics). Among these approaches are e.g., WSML [WSM05a], SWSF [SWS05], Rei [KFJ05] or the approach of [VAG05] who presented a policy matching approach extending a WS-Policy-based OWL ontology with SWRL rules. In contrast to my heterogeneous approach of combining rules (logic programming) with ontologies (description logics) (see section 4.2.2) SWRL and most other works are homogeneous approaches with relatively high complexity bounds for the ontology reasoning part (due to the fact that standard rule engines are not optimized for DL reasoning). Moreover, in these works rules are primarily used in specific contexts in order to overcome certain restrictions of the core OWL representation, but a comprehensive and coherent use of rules as in my approach is missing.

The work of Grosof et al. [GLC99], the Semantic Web Enabling Technology (SWEET) toolkit [SWE05] comprises the CommonRules syntax and also enables business rules to be represented in RuleML. Whilst their approach deals with contracts in a broader range namely e-commerce contracts and mainly supports rule priorities via Generalized Courteous Logic Programs (GCLP) [Gro99] and to some extend procedural attachments, my approach is focused on the specifics of Service Level Management. In contrast to SWEET, ContractLog incorporates additional logical concepts which are needed for adequate SLA representation such as contract states, explicit rights and obligations (deontic norms) supplemented with violations and exceptions of norms, integrity constraints, event processing facilities with active sensing, monitoring and triggering of actions in terms of ECA rules, full support for different type systems (e.g., Java, Semantic Web) and procedural attachments in order to integrate existing business object implementations, and SLA-specific contract vocabularies.

3 Knowledge Representation

Knowledge representation (KR) focuses on methods for describing the world in terms of high-level, abstracted models which can be used to build intelligent applications, i.e., it provides methods to find implicit consequences of explicitly represented knowledge. Approaches can be roughly divided into *logic based formalisms*, usually a variant of first-order predicate calculus and *non-logic based formalisms* such as semantic networks, frames or (early) production rule systems. Non-logic based approaches, which are often based on ad hoc data structures and graphical representations, typically lack a precise formal semantics which makes it hard to verify the correctness of drawn consequences. On the other hand, logic based approaches use the powerful and general semantics of first-order logic (FOL) (typically a decidable subset of FOL) which allows a precise characterization of the meaning of a world by expressing it as a knowledge base of statements in a language which has a truth theory. While the syntax may differ, the semantics of FOL KBs is typically given in a Tarski-style semantics.

3.1 Rule Based Knowledge Representation

Rule based systems have been investigated comprehensively in the realms of declarative programming and expert systems over the last two decades. Using (inference) rules has several advantages: reasoning with rules is based on a semantics of formal logic, usually a variation of first order predicate logic, and it is relatively easy for the end user to write rules. The basic idea is that users employ rules to express *what* they want, the responsibility to interpret this and to decide on *how* to do it is delegated to an interpreter (e.g., an inference engine or a just in time rule compiler). Traditionally, rule-based systems have been supported by two types of inferencing algorithms: forward-chaining and backward-chaining.

3.1.1 Forward Chaining Rule Systems

Forward chaining is one of the two main methods of reasoning when using "if-then" style inference rules in artificial intelligence. Forward chaining is data-driven. The inference engine makes inferences based on rules from given data. It starts with the available data and uses inference rules to extract more data until an optimal goal is reached. An inference engine using forward chaining searches the inference rules until it finds one where the *if clause* is known to be true.

When found it can conclude, or infer, the *then clause*, resulting in the addition of new information to its KB. The most common form of forward chaining is the Rete algorithm. In a nutshell, this algorithm keeps the derivation structure in memory and propagates changes in the fact and rule base. There are many forward-chaining implementations in the area of deductive databases and many well-known forward-reasoning engines for production rules ("if condition then action" rules) such as ILOG's commercial rule system or popular open source solutions such as CLIPS or Jess which are based on the RETE algorithm.

3.1.2 Backward Chaining Rule Systems

The other main reasoning method for "if ... then ..." rules is backward-chaining which is typically used in logic programming, where the rules are called derivation rules. Backward chaining starts with a list of goals (hypothesis) and works backwards to see if there are data available that will support any of these goals. Accordingly, backward chaining is goal-driven. An inference engine using backward chaining would search the inference rules until it finds one which has a *then clause* that matches a desired goal. If the *if clause* of that inference rule is not known to be true, then it is added to the list of goals. The common deductive computational model of logic programming uses backward-reasoning (goal-driven) *resolution* to instantiate the program clauses via goals and uses *unification* to determine the program clauses to be selected and the variables to be substituted by terms. The unification algorithm supports backtracking usually according to depth-first recursive backward-chaining, but forward-chaining bottom-up approaches are also possible.

3.1.3 Discussion Backward Chaining vs. Forward-Chaining in SLA Representation

Forward-chaining, e.g based on the Rete algorithm in production rules, can be very effective, e.g., if you just want to find out what new facts are true or when you have a small set of initial facts and when there tends to be lots of different rules which allow you to draw the same conclusion. However, I argue that production rule systems are not suitable in SLA representation and that backward-chaining in the sense of logic programming qualifies to be the better choice:

- In forward-reasoning additional software must propagate changes to the memory-based fact base which leads to a lot of redundancy and difficulties, e.g., a relational database normally does not propagate changes and for real-time access the fact base and the database must be synchronized.
- In the SLA domain large set of initial facts are provided which are likely to change. Using forward chaining, lots of rules would be eligible to fire in any cycle and a lot of irrelevant conclusions are drawn, while in backward-reasoning the knowledge base can be temporarily populated

with the needed facts from external systems to answer a particular goal at query time which can be discarded from the memory afterwards. In fact, forward-reasoning on the Web is well-suited only for closed scopes, e.g., firing rules when certain events occur.

- The open-distributed environments such as the web are usually based on a pull-model and most implementations of push-architectures (the push model relates to active event processing) are basically pull-concepts, i.e.„ the push functionality is simulated via frequently issuing queries, e.g., a mail client which queries the mailbox every second for new mails. Therefore, a goal-driven backward-reasoning system perfectly fits to those architectures.
- Forward-reasoning production rules have an operational semantics but no clear logical semantics and a restricted expressiveness, e.g., no recursion, disjunctions in the sense of "else" are dangerous in forward-reasoning and non-monotonic features such as default negation in unstratified production rule systems are not clear.

In this dissertation I defend the adequacy of goal-driven logic programming techniques for SLA representation and show how required KR formalisms can be expressed in terms of backward-chaining derivation rules. Since in SLA representation there is also a need for reactive rules with forward-driven operational semantics, I will show in section 4.7 how complex reaction rules can be implemented as a common add-on to backward-chaining rule engines enabling a tight combination of derivation rules and reaction rules. For a list of advantages of using logic programming in SLA representation see section 1.3.

3.2 First-Order Logic

This section recalls the definition of a first order logic (FOL) language and classical FOL models (structures) under Tarski semantics that I adopt from [Llo87, LMR92, Fit96]. Both interrelated concepts play a central role in logic and form a general basis in this dissertation that allows me to cover a wide range of logical formalisms for rule-based contract representation in the ContractLog KR (see chapter 4).

3.2.1 Syntax

This subsection defines the syntax of a first order language according to [Llo87, LMR92, Fit96].

Definition 8 *(**Signature**) S is a signature if S is a four-tuple $\langle \overline{P}, \overline{F}, arity, \overline{c} \rangle$ where:*

1. *$\overline{P}$ is a finite sequence of predicate symbols $\langle P_1, .., P_n \rangle$.*

2. *$\overline{F}$ is a finite sequence of function symbols $\langle F_1, .., F_m \rangle$*
3. *For each P_i respectively each F_j, $arity(P_i)$ resp. $arity(F_j)$ is a non-zero natural number denoting the arity of P_i resp. F_i.*
4. *$\overline{c} = \langle c1, .., c_o \rangle$ is a finite or infinite sequence of constant symbols.*

A signature is called function-free if $\overline{F} = \emptyset$.

Definition 9 ***(Alphabet)*** *An alphabet Σ consists of the following class of symbols:*

1. *A signature $S = \langle \overline{P}, \overline{F}, arity, \overline{c} \rangle$.*
2. *A collection of variables V which will be denoted by identifiers starting with a capital letter like U,V,X*
3. *Logical connectives / operators: $\neg$. (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\equiv$ (syntactical equivalent), $=$ (equivalence), $\bot$ (bottom), $\top$ (top).*
4. *Quantifier: $\forall$ (forall), $\exists$ (exists).*
5. *Parentheses and punctation symbols: "(", ")", ",".*

Definition 10 ***(Terms)*** *A term is defined inductively as follows:*

1. *A variable is a term.*
2. *A constant in $\overline{c}$ is a term.*
3. *If f is a function symbol with arity n and $t_1, .., t_n$ are terms, then $f(t_1, .., t_n)$.*

Function symbols are written in prefix notation whereby a function always precedes its terms. However, throughout the dissertation I sometimes write terms composed of both prefix and infixed symbols, e.g., $(f(2) - f(1)/f(1))$. There are standard ways of dealing with this issues which I will not discuss in this dissertation.

Definition 11 ***(Atom)*** *Let p be a predicate symbol with arity $n \in \aleph$. Let $t_1, .., t_n$ be terms, then $p(t_1, .., t_n)$ is an atomic formula of terms. A* ground atom *is a atomic formulae without variables.*

Definition 12 ***(Well-formed Formula)*** *A (well-formed) formula is defined as follows:*

1. *An atom is a formula.*

2. *If H and G are formula then*
   - *$\neg H$ is a formula (negation)*
   - *$(H \wedge G)$ is a formula (conjunction)*
   - *$(H \vee G)$ is a formula (disjunction)*
   - *$(H \rightarrow G)$ is a formula (implication)*
   - *$(H \equiv G)$ is a formula (equivalence)*
3. *If H is a formula and X is a variable, then $(\forall XH)$ and $(\exists XH)$ ar formulas.*

The following precedences are defined:

1. $\neg$, $\forall$, $\exists$
2. $\wedge$, $\vee$
3. $\rightarrow$, $\equiv$

Definition 13 ***(First-Order Language)*** *A* FOL language *is defined over an alphabet Σ where the signature S my vary from language to language. It consists of the set of all formulas that can be constructed according to the definitions of well-founded formulas using the symbols of Σ. A FOL language is called function-free if the signature is function-free.*

Thus, a language in addition to a signature also contains the logical symbols and a list of variables. The notion "first-order" refers to the fact that quantification is over individuals rather than classes (or functions).

Definition 14 ***(Scope of Variables)*** *Let X be a variable and H be a formula. The scope of $\forall X$ in $\forall XH$ and of $\exists X$ in $\exists XH$ is H. Combinations of $\forall X$ and $\exists X$ bind every occurrence of X in their scope. Any occurrences of variables that are not bound are called free.*

Definition 15 ***(Open and Closed Formula)*** *A formula is open if it has free variables. A formula is closed if it has no free variables.*

Definition 16 ***(Literal)*** *A literal L is an atom or the negation of an atom.*

Definition 17 ***(Complement)*** *Let L be a literal. The complement $-L$ of L is defined as follows:*

$$-L := \begin{cases} \neg A \text{ if } L \equiv A \\ A \text{ if } L \equiv \neg A \end{cases}$$

, where A is an atom.

Definition 18 ***(Theory)*** *A FOL theory Φ or FOL knowledge base is a set of formulae in a FOL language Σ: $\Phi \subseteq \Sigma$. The signature S of Φ is obtained from all the constant, function and predicate symbols which occur in Φ.*

Every finite FOL knowledge base (FOL KB) is equivalent to the conjunction of its elements, i.e., it might be equivalently written as a conjunction of formulas.

3.2.2 Interpretations and Models

This subsection is concerned with attributing meaning (or truth values) to sentences (well-formed formulae) in a FOL language. The definitions follow [Llo87, LMR92, Fit96]. Informally, the sentences are mapped to some statements about a chosen domain through a process known as interpretation. An interpretation which gives the value true to a sentence is said to satisfy the sentence. Such an interpretation is called a model for the sentence and an interpretation which does not satisfy a sentence is called a counter-model.

Definition 19 ***(Interpretation / Structure)*** *Let $S = \langle \overline{P}, \overline{F}, arity, \overline{c} \rangle$ be a signature. I is called an interpretation (or a structure) for S if $I = \langle |M|, \overline{P}^I, \overline{F}^I, \overline{c}^I \rangle$ consists of:*

1. *a non-empty set $|M|$ called the universe of I or the domain of the interpretation. The members of $|M|$ are called individuals of I.*
2. *$\overline{P}^I = \langle P_1^I, .., P_k^I \rangle$ associates with each predicate P_i in S of arity $n = arity(P_i)$ an n-ary relation P_i^I on $|M|$, i.e., $P_i^I \subseteq |M|^n$, where $|M|^n$ denotes the collection of all n-tuples from $|M|$.*
3. *$\overline{F}^I = \langle F_1^I, .., F_l^I \rangle$ is an interpretation for each function symbol F_j of arity m, where F_j^I is an m-place function $\overline{F}_j^I : |M|^m \rightarrow |M|$, i.e., F_j^I is defined on the set of m-tuples of individuals $|M|^m$ with values in $|M|$.*
4. *$\overline{c}^I = \langle c^I | c = constant \rangle$ is an interpretation for the constants of S: $c \in S$, where c^I is an individual of M: $c^I \in |M|$.*

Definition 20 ***(Assignment)***

1. Variable Assignment*: Let Σ be a FOL language with $\overline{X}$ its set of variables, and I an interpretation for Σ. An assignment is a function σ from $\overline{X}$ into the universe of Σ.*
2. Term Assignment*: Let I be an interpretation of a FOL language Σ with domain $|M|$ and variable assignment σ. The term assignment wrt σ of the term in Σ is defined as:*
   - Each variable is given its assignment according to σ.
   - Each constant is given its assignment according to I.
   - If $t'_1, .., t'_n$ are the term assignments of $t_1, .., t_n$ and f' is the assignment of the function symbols f with arity n, then $f'(t'_1, .., t'_n) \in |M|$ is the term assignment of $f(t_1, .., t_n)$.

That is, given an assignment σ, any variable term of the language that is in the domain of σ is given a constant value in $|M|$.

Definition 21 *(**Truth Values**) Let I be an interpretation of a FOL language Σ with domain $|M|$ and σ be a variable assignment. A formula $F \in \Sigma$ can be given a truth value "false" or "true" as follows:*

1. *If the formula is an atom $p(t_1, .., t_n)$ then the truth value is obtained by calculating the value of $p'(t'_1, .., t'_n)$ where p' is the mapping assigned to p by I and $t'_1, .., t'_n$ are the term assignments of $t_1, ..t_n$ wrt I and $\overline{X}$.*
2. *The truth values of the following formulas is given by the following table:*

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F = G$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

3. *If $\exists XF$, then the truth value of the formula is true if there exists $c \in |M|$ such that the formula F has truth value "true" wrt I and $\sigma(X/c)$; otherwise it is false.*
4. *If the formula has the form $\forall XF$, then the truth value of the formula is true if, for all $c \in |M|$ F is "true" wrt I and $\sigma(X/c)$; otherwise, its truth value is false.*

Definition 22 *(**Satisfaction**) If F is a formula and σ is an assignment to the interpretation I of a FOL language Σ, then the relation $I \models F[\sigma]$ means that F is true in I when there is a substitute for each free variable X of F with the value of $\sigma(X)$. The inductive requirements of "$\models$" are:*

1. *For any atomic formula of the form $p(t_1, ..t_n)[\sigma]$ iff $\langle t^\sigma_1, .., t^\sigma_n \rangle \in p^I$.*
2. *$I \models \neg F[\sigma]$ iff it is not the case that $I \models F[\sigma]$*
3. *$I \models (F \wedge G)[\sigma]$ iff both $I \models F[\sigma]$ and $I \models G[\sigma]$. Similarly, for the other statements.*
4. *$I \models \exists XF[\sigma]$ iff there exists some assignment σ' such that*
   - for every variable Y different from X $\sigma'(Y) = \sigma(Y)$
   - $\sigma'(X)$ is defined and $I \models F[\sigma']$
5. *$I \models \forall XF[\sigma]$ iff for any assignment σ', if $\sigma'(X)$ is defined and σ' is equal to σ on each variable different from X, then $I \models F[\sigma']$*

Accordingly, a formula F is satisfied by an interpretation I (F is true in I: $I \models F$) iff $I \models_\sigma F$ for all variable assignments σ. F is valid iff $I \models F$ for every interpretation I.

The satisfaction relation $\models$ goes back to A. Tarski and is a major achievement in logic.

Definition 23 *(**Model**) Let I be an interpretation of a FOL language Σ. Then I is a model of a closed formula F, if F is true wrt I. Further, I is a model of a set $\overline{F}$ of closed formulas, if I is a model of each formula of $\overline{F}$. I is a model of an FOL KB Φ iff $I \models F$ for every formula $F \in \Phi$: $I \models \Phi$.*

Definition 24 *(**Logical Consequence, Entailment, Logical Implication**) A formula $F \in \Sigma$ is a logical consequence of a FOL KB Φ written as $\Phi \models F$, i.e., Φ entails F iff for all models $I \in \Sigma$ for which $I \models \Phi$ also $I \models F$. For a fixed FOL language (and signature) Σ let Φ and Ψ be two sets of sentences (two KBs), then $\Phi \rightarrow \Psi$ means that for every interpretation I of Σ, if I is a model for Φ then it is also a model for Ψ.*
$\Phi \rightarrow \Psi$ is also meaningful when Φ and Ψ are sets of formulas with variables, i.e., for every interpretation I of Σ and every assignment σ in I, if $I[\sigma]$ satisfies every formula in Φ then it also satisfies every formula in Ψ.

3.3 Logic Programming

Full first-order logic is not suitable as a declarative programming language due to the following reasons:

- unrestricted FOL is in general undecidable
- finding (most general) unifier and solving formula is highly complex
- large search domains, which must be restricted using complex control structures
- danger of implementation incompleteness
- the results are not always unique

Hence, logic programming is based on a subset of FOL which deals with a specific class of well-formed formulas, so called statement clauses which consist of an antecedent part and a consequent. The declarative meaning for such clauses is that the consequent part is true, if the antecedents are true. The procedural meaning is, that the consequent is proven by reducing it to a set of sub-goals given by the antecedent part. The most common form of logic programming is based on Horn Logic where clauses in normal form only have one positive literal which is the consequent. Such programs are called definite LPs or Horn LPs. The semantics of definite Logic Programs (LPs) is based on minimal Herbrand models. Although Horn LPs are expressive enough to model hard problems the formulation is neither easy nor elegant. Hence, extensions to definite LPs like different forms of negations have been proposed. In this section I introduce relevant terms, concepts, syntax and semantics of different classes of logic programs (LPs) derived from [Llo87, LMR92, Fit96].

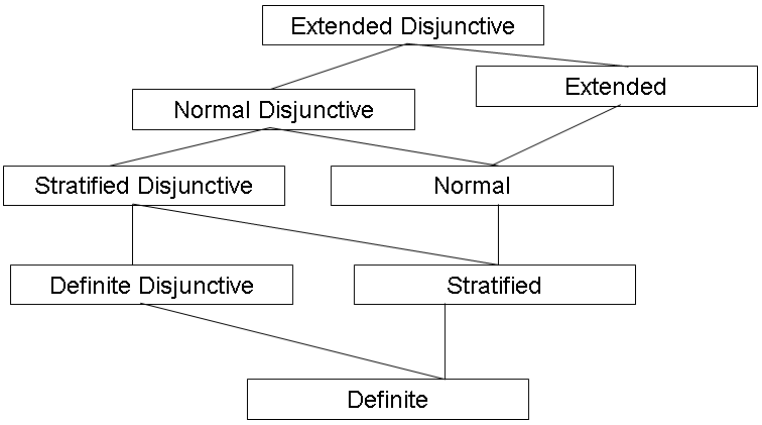

Figure 3.1: Classes of LPs

3.3.1 Syntax of Logic Programs

Definition 25 *(**Clause**) A clause is a formula such as* $\forall \overline{X}(L_1 \vee .. \vee L_m)$ *where each* L_i *is a literal and* $\overline{X} = \{X_1, .., X_n\}$ *are all the variables occurring in* $L_1 \vee .. \vee L_m$.

Throughout this thesis I will refer to different classes of clauses such as: *propositional clauses*, *Datalog clauses*, *definite Horn clauses*, *normal clauses*, *extended clauses*, *positive clauses*, *positive-disjunctive clauses*, *disjunctive clauses*, and *extended disjunctive clauses*. Associated with each type of clause is a class of logic programs: *propositional LP*, *Datalog LP*, *definite LP*, *stratified LP*, *normal LP* (aka general LP), *extended LP*, *disjunctive LP* and combinations of classes, with an increasing expressiveness as illustrated in figure 3.1 for several classes of LPs. Each class can be propositional (without terms), Datalog (without functions) or with terms and variables.

These LPs are defined as follows:

Definition 26 *(**Logic Programs and Rules**) Given a FOL language* Σ*, a (disjunctive extended) logic program* P *consists of logical rules (or program clause) of the form*

$$A_1, .., A_k \leftarrow B_1, .., B_m, not\, C_1, .., not\, C_n$$

or equivalently

$$\forall \overline{X}(A_1 \vee .. \vee A_k \leftarrow B_1 \wedge .. \wedge B_m \wedge not\, C_1 \wedge .. \wedge not\, C_n)$$

which is a convenient notation for a FOL clause where all variables $X_i \in \overline{X}$ *occurring in the literals* A_i, B_j, C_k *are universally quantified* $\forall X_1..\forall X_s$, *the commas in the antecedent denote conjunction and the commas in the consequent*

denote disjunction, and not denotes negation by default, rather than classical negation. For short a rule is denoted in set notation as:

$A \leftarrow B \wedge not\, C$

where $A = A_1 \vee .. \vee A_k$, $B = B_1 \wedge .. \wedge B_m$, $C = not\, C_1 \wedge .. \wedge not\, C_n$. *Note that* C *is a disjunction and according to De Morgan's law* $not\, C$ *is taken to be a conjunction. The* A *is called the* rule head *which consists of the set of head literals and* B *and* C *is called the* rule body *which consists of the set of body literals. Note that this set notation is legitime because the conjunction is commutative.*

A rule is called:

- *a* fact *if* $m = n = 0$, *i.e.,* $A \leftarrow \emptyset$
- *a* query *(or goal) if* $k = 0$, *i.e.,* $\leftarrow B \wedge C$. *A query or goal is called atomic if it consists of a single literal* B_1, *i.e.,* $m = 1$ *and* $n = 0$.
- *a* propositional rule *if the arity of all predicates is* 0, *i.e., all literals are propositional ones. If all rules in a program* P *are propositional the* P *is called a* propositional LP.
- *a* Datalog rule *if it contains no functions, i.e., is function-free and no predicate symbol of the input schema appears in the rule head. A* Datalog LP *(aka deductive database) is a function-free LP.*
- *a* definite or positive rule *(or* Horn clause*), if all literals are atoms,* $n = 0$ *and* $k = 1$, *i.e., it neither contains negation nor disjunction. The corresponding LP is called* positive or definite LP *(or* Horn Program*).*
- positive-disjunctive rule, *if all literals are atoms and* $n = 0$, *i.e., it does not contain negation. The corresponding LP is called* positive-disjunctive LP.
- normal rule, *if all literals are atoms and* $k = 1$, *i.e., it does not contain disjunction. The corresponding program is called a* normal LP.
- extended rule, *if* A_i, B_i *and* C_i *are literals, i.e., are atoms or explicitly negated atoms . The corresponding programm is called an* extended LP.
- disjunctive rule, *if* $k > 1$, *i.e., it does contain a disjunction. The corresponding program is called a* disjunctive LP.
- range-restricted *if all variable symbols occurring in the head also occur in the positive body.*
- ground *if no variables occur in it.*

3.3.2 Semantics of Logic Programs

Proof-theoretically the semantics of a logic program P is defined as a set of literals that is (syntactically) derivable from P using a particular derivation mecha-

nism such as SLDNF resolution. Model-theoretically, a semantics for a logic program P is concerned with attributing meaning (truth values) to clauses (rules). The properties of soundness and completeness establish a relation between the notions of syntactic ($\vdash$) and semantic ($\models$) entailment in logic programming. In this subsection I will review several approaches to define proof-theoretic and model-theoretic semantics for different types of logic programs which are relevant in the context of the ContractLog KR which will be introduced in chapter 4.

3.3.2.1 Substitution and Unification

I first introduce the concepts of substitution and unification from [Llo87] which are at the heart of *proof-theoretic semantics* of non-ground LPs.

Definition 27 *(**Substitution**) A substitution θ in a language Σ is a finite set of the form $\{X_1/t_1, .., X_n/t_n\}$, where each X_i is a variable in Σ, each t_i is a term in Σ distinct from X_i and the variables $X_1, .., X_n$ are pairwise distinct. Each element X_i/t_i is called a binding for X_i. θ is called a ground substitution if the t_i are all ground terms. θ is called a variable-pure substitution if the t_i are all variables.*

Definition 28 *(**Expression**) An expression E is either a term, a literal or a conjunction or disjunction of literals.*

Definition 29 *(**Instance**) Let $\theta = \{X_1/t_1, .., X_n/t_n\}$ be a substitution and E be an expression then $E\theta$ is the instance of E by θ is the expression obtained from E by simultaneously replacing each occurrence of the variable X_i in E by the term t_i for $i = 1, .., n$. if $E\theta$ is ground then $E\theta$ is called a ground instance of E.*

Definition 30 *(**Variant**) Let E and D be expressions. E and D are variants if there exists substitutions θ and σ such that $E = D\theta$ and $D = E\sigma$.*

Definition 31 *(**Renaming Substitution**) Let E be an expression and $\overline{X}$ be the set of variables occurring in E. A renaming substitution for E is a variable-pure substitution $\{X_1/Y_1, .., X_n/Y_n\}$ such that $\{X_1, .., X_n\} \subseteq \overline{X}$, the Y_i are pairwise distinct and $(\overline{X} \setminus \{X_1, .., X_n\}) \cap \{Y_1, .., Y_n\} = \emptyset$.*

Definition 32 *(**Composition**) Let $\theta = \{X_1/s_1, .., X_m/s_m\}$ and $\sigma = \{Y_1/t_1, .., Y_n/t_n\}$ be substitutions. The composition $\theta\sigma$ of θ and σ is the substitution obtained from the set*

$$\{X_1/s_1\sigma, .., X_m/s_m\sigma, Y_1/t_1, .., Y_n/t_n\}$$

by deleting any binding $X_i/s_i\sigma$ for which $X_i = s_i\sigma$ and deleting any binding Y_j/t_j for which $Y_j \in \{X_1, .., X_m\}$.

Definition 33 *(**Most General Unifier** (MGU)) Let $\overline{E}$ be a finite set of expressions. A substitution θ is called a unifier for $\overline{E}$ if $\overline{E}\theta$ is a singleton. An unifier for $\overline{E}$ is called most general unifier (mgu) for $\overline{E}$ if for each unifier σ of $\overline{E}$ there exists a substitution γ such that $\sigma = \theta\gamma$. $\overline{E}$ is called unifiable if there exists a unifier for $\overline{E}$.*

Note that a mgu for a set of expressions is unique modulo renaming if there exists a mgu at all.

3.3.2.2 Minimal Herbrand Model

For the *model-theoretic semantics* I first introduce the minimal or least Herbrand model semantics which is considered as the natural interpretation of a definite LP. I then extend the minimal Herbrand semantics for other subclasses of LPs and described other (declarative) semantics as well as their proof-theoretic counterparts for logic programming.

Definition 34 *(**Herbrand Universe**) The Herbrand universe of a program P defined over the alphabet Σ, denoted U_P, is the set of all ground terms which can be formed out of the constants and function symbols of the signature S of Σ.*

Definition 35 *(**Herbrand Base**) The Herbrand base of a program P, denoted B_P, is the set of all ground atomic literals which can be formed by using the predicate symbols in the signature S of Σ with the ground terms in U_P as arguments.*

Definition 36 *(**Herbrand Instantiation aka Grounding**) The Herbrand instantiation ground(P) of P consists of all ground instances of all rules in P wrt to the Herbrand universe U_P which can be obtained as follows: The ground instantiation of a rule r is the collection of all formulae $r[X_1/t_1, .., X_n/t_n]$ with $X_1, .., X_n$ denoting the variables which occur in r and $t_1, .., t_n$ ranging over all terms in U_P.*

Definition 37 *(**Herbrand Interpretation**) The Herbrand interpretation I^{Herb} of P is a consistent subset of B_P. The interpretation is given as follows:*

1. *The domain of the interpretation is the Herbrand universe U_P.*
2. *Constants are assigned themselves in U_P.*
3. *IF f is a function in P with arity n then the mapping $f' : U_P^n \mapsto U_P$ assigned to f is defined by $f'(t_1, ..t_n) := f(t_1, .., t_n)$.*

Note that, since the assignment to constant and function symbols is fixed for Herbrand interpretations, it is possible to identify a Herbrand interpretation with a subset of the Herbrand base. For any Herbrand interpretation, the corresponding subset of the Herbrand base is the set of all ground atoms which are true wrt the interpretation.

Definition 38 *(**Herbrand Model**) Let P be a positive / definite program. A Herbrand interpretation I^{Herb} of P is a model of P, denoted as M^{Herb}, iff for every rule $H \leftarrow B_1, .., B_n \in ground(P)$ the following holds: If $B_1, .., B_n \in I^{Herb}$ then $H \in I^{Herb}$.*

The Herbrand model M^{Herb} satisfies the *unique name assumption*, i.e., for any two distinct ground terms in B_P, their interpretations are distinct as well.

Definition 39 *(**Unique Name Assumption and Domain Closure Assumption**) Let Σ be a given language. The unique name assumption (UNA) restricts the model M^{Herb}, where syntactically different ground terms t_1, t_2 are interpreted as non-identical elements: $t_1^{M^{Herb}} \neq t_2^{M^{Herb}}$.*
The domain closure assumption (DCA) is a restriction to those models M^{Herb} where for any element a in M^{Herb} there is a term t that represents this element: $a = t^{M^{Herb}}$.

Model-theoretically the intended meaning of a LP is that a formula should be true if it is a logical consequence of the program, i.e., it is true in all models of the program. For definite LPs this intention leads to a semantics that coincides with the intuition because of the model intersection property.

Definition 40 *(**Model Intersection Property**) Let $\overline{M}^{Herb}$ be the set of all Herbrand models of a program P. The intersection of all Herbrand models $\bigcap \overline{M}^{Herb}(P)$ of a definite LP P is also a Herbrand model of P.*

Note that since every definite LP P has B_P as an Herbrand model, the set of all Herbrand models for P is always non-empty: $\bigcap \overline{M}^{Herb}(P) \neq \emptyset$.

Definition 41 *(**Minimal Herbrand Model**) Let P be a definite LP then the minimal or least Herbrand model M_P^{Herb} of P is the intersection of all Herbrand models for P.*

The constructive computational characterization of the minimal Herbrand model of a definite LP P is based on the least fixpoint of the immediate consequence operator of P. A detailed description of the theory of lattices and fixpoints can be found in [Llo87], I recall some relevant definitions.

Definition 42 *(**Immediate Consequence Operator**) Let P be a definite LP. Let $I^{Herb} \subseteq B_P$ be a set of atoms. The set of immediate consequences of I^{Herb} wrt P is defined as follows:*

$T_P(I^{Herb}) := \{A \mid \text{there is } A \leftarrow B \in ground(P) \text{ with } B \subseteq I^{Herb}\}$.

Definition 43 *(**Monotonic Mapping**) Let $T : P(U) \rightarrow P(U)$ be a mapping then T is monotonic if $T(X) \subseteq T(Y)$, whenever $X \subseteq Y$.*

Definition 44 *(**Ordinal Power of T**) Let $T : P(U) \rightarrow P(U)$ be a monotonic mapping then:*
$T \uparrow 0 = \emptyset$
$T \uparrow a = T(T \uparrow (a-1))$ *if a is a successor ordinal*
$T \uparrow a = \bigcup(T \uparrow b | b < a)$ *if a is a limit ordinal*

Definition 45 *(**Fixpoint of operators**) An operator T is a function $T : P(U) \rightarrow P(U)$, where $P(U)$ denotes the powerset of a countable set U. A set $X \subseteq U$ is called a fixpoint of the operator $T : P(U) \rightarrow P(U)$ iff $T(X) = X$*

Definition 46 *(**Least Fixpoint**) Let $T : P(U) \rightarrow P(U)$ be a mapping. An element $e \in P(U)$ is called a least fixpoint $lfp(T)$ iff e is a fixpoint of T and for all fixpoints f of T it is that $e \subseteq f$.*

According to the Fixpoint Theorem of Knaster and Tarski (see [KM97b] for more details) each monotonic operator T has a least fixpoint $lfp(T)$, which is the limit of the sequence $T^0 = \emptyset$, $T^{i+1} = T(T^i)$ for $i \geq 0$. It appears that for each set P of clauses $lfp(T)$ coincides with the unique least Herbrand model of P, where a model M^{Herb} is smaller than a model N^{Herb}, if $M^{Herb} \subset N^{Herb}$ [EK76].

Definition 47 *(**Fixpoints of Monotonic Mappings**) Let T be a monotonic mapping. Then T has a least fixpoint $lfp(T)$. For every ordinal a, $T \uparrow a \subseteq lfp(T)$. Moreover, there exists an ordinal b such that $c \geq b$ implies $T \uparrow c = lfp(T)$.*

If the operator T_P is not only monotonic but also continuous, then a least fixpoint of T_P is always reached not later than at the first limit ordinal (see [Llo87]). By Kleene's theorem (see [Doe94]) $lfp(P) = T^{\infty}(\emptyset)$, where T^i is inductively defined by $T^0 = \emptyset$, $T^{i+1} = T(T^i)$ for $i \geq 0$.

Theorem 1 *(**Fixpoint Characterization of the Minimal Herbrand Model**) Let P be a definite LP then $M_P^{Herb} = lfp(T_P) = T_P \uparrow \omega$.*

In summary, the semantics of LPs is now defined as follows:

Definition 48 *(**Herbrand Semantics of Logic Programs**) Let the grounding of a clause r in a language Σ be denoted as $ground(r, \Sigma)$ where $ground(r, \Sigma)$ is the set of all clauses obtained from r by all possible substitutions of elements of U_Σ for the variables in r. For any definite LP P*

$ground(P, \Sigma) = \bigcup_{r \in P} ground(r, \Sigma)$

The operator $T_P : 2^{B_P} \rightarrow 2^{B_P}$ associates with P is defined by $T_P = T_{ground(P)}$, where $ground(P)$ denotes $ground(P, \Sigma(P))$, and accordingly:

$SEM_{Herb}(P) = M^{Herb}_{ground(P)}$.

Generating $ground(P)$ is often a very complex task, since, even in case of function-free languages, it is in general exponential in the size of P (see section 7.1 for complexity results). Moreover, it is not always necessary to compute $M^{Herb}_{ground(P)}$ in order to determine whether $P \models A$ for some particular atom A. In practice, various proof-theoretic strategies of deriving atoms from a LP have been proposed. These strategies are based on variants of Robinson's famous *Resolution Principle* [Rob65]. The major variant is SLD-resolution [KK71].

3.3.2.3 SLD Resolution

In a nutshell, in SLD a goal is a conjunction of atoms. A substitution θ is a function that maps variables to terms. Asking a query $Q?$, where $Q?$ may contain variables, to a program P means asking for all possible substitutions θ of the variables in $Q?$ such that $Q\theta$ follows from P, i.e., θ is the answer to Q. In other words, SLD resolution repeatedly transforms the initial goal by applying the resolution rule to an atom Q_i from the query/goal and a rule from P, unifying Q_i with the head H of the rule, i.e., it tries to find a substitution θ such that $H\theta = Q_i\theta$. The typical selection rule is to choose always the first atom in the query. This step is repeated until all goals are resolved and the empty goal is obtained.

Example 1 *(**Linear resolution computation step**)*

$$\frac{\neg Q_1, .., \neg Q_n \qquad\qquad \neg A_1, .., \neg A_m, H}{\Big| \quad \theta = unify(Q_1, \neg H)}$$

For a more precise account see [Apt90, Llo87] and [Lei97] for resolution on normal clauses. The task to find substitutions θ such that $Q\theta$ is derivable from the program P as well as M^{Herb}_P is closely related to SLD. The following properties are equivalent:

Theorem 2 *(**Soundness and Completeness of SLD**)*

- $P \models \forall Q\theta$, *i.e* $\forall Q\theta$ *is true in all models of* P,
- $M_P^{Herb} \models \forall Q\theta$,
- *SLD computes an answer* τ *that subsumes* θ *wrt* Q, *i.e.,* $\exists\sigma : Q\tau\sigma = Q\theta$.

Since SLD resolution is a top-down approach which starts with the query, the main feature of it is, that it automatically ensures, that it only considers those rules that are relevant for the query to be answered (see also section 3.1.3 for a discussion of backward vs. forward reasoning). Rules that are not at all related are simply not considered in the course of the proof. Note that there are also several bottom up approaches for computing the least Herbrand model M_P^{Herb} from below. However, the bottom-up approach has two serious shortcomings:

1. The "goal-orientedness" from top-down approaches is lost, i.e the whole M_P^{Herb} has to be computed even for those facts that have nothing to do with the query.
2. In any step facts that are already computed before are recomputed again.

Partial solutions have been proposed, e.g., semi-naive bottom-up evaluation [Ull89, Bry90b] or Magic Sets techniques [BR91]. However, as discussed in section 3.1 top-down semantics are more appropriate in SLA representation and I mainly focus on backward-reasoning logic programming techniques in this dissertation (except for the active forward-directed processing of reactive rules - see section 4.7).

3.3.2.4 Theory of Logic Programming with Negation

As I will discuss in section 3.6 definite LPs are not expressive enough for SLA representation since they e.g., exclude negative information and (non-monotonic) default statements such as "*normally a implies c, unless something abnormal holds*". Such statements and the computation of default negation where the main motivation for alternative formulations of non-monotonic reasoning by circumscription [McC80], default reasoning [Rei80] or autoepistemic reasoning [Mar91]. Independently of these work in non-monotonic reasoning the proof-theory for *negation-as-finite-failure* (NAF), the well-known *SLDNF resolution* (SLD+NAF), originated from SLD resolution. In short, negation-as-finite-failure can be characterized as: A (default) negated literal $\sim L$ succeeds, if L finitely fails. See [Llo87, Apt90] for the formal definition of SLDNF resolution and NAF. The implementation is typically given as a cut-fail test:

```
not([P|Args]) :-
    derive([P|Args]), % derive P(Args)
    !, % cut
    fail(). % fail
not([P|Args]). % positive answer
```

The corresponding model-theoretic semantics is define by *Clark's completion* (COMP) [Cla78] who's idea was to interpret "$\leftarrow$" in rules as "$\leftrightarrow$" in the classical sense.

Definition 49 ***(Clark's Completion COMP)*** *Clark's completion semantics COMP for a program P is given by the set of all classical models $\overline{M}(comp(P))$ of the completion theory $comp(P)$.*

See Clark's Equational Theory for more details. COMP gives two rules for inferring negative information:

- Infer $\neg A$ iff $B_P \setminus \overline{M}(comp(P)) \models \neg A$
- Infer $\neg A$ iff $\overline{M}(comp(P)) \models \neg A$

But, (two-valued) COMP is incomplete and does not characterize the transitive closure correctly. In [Prz90a] various problems with loops in COMP where discussed. Therefore, Fitting [Fit85] introduce a three-valued formulation $comp_3(P)$ of the two-valued COMP. It was shown by Kunen [Kun87] that SLDNF is sound and complete wrt $COMP_3$ for propositional LPs and correct but not complete in the predicate logic case [She88].
SLDNF-resolution suffers from problems with loops and floundering and its implementation is only a simple test, i.e., no variable bindings are produced. See [She91] for a discussion of unsolvable problems related to SLDNF. Much work has been done to define restriction properties (on the dependency graph whose vertices are the predicate symbols from a program P) for which SLDNF is complete. I briefly review the important ones:

- stratified: no predicate depends negatively on itself
- strict: there are no dependencies that are both even and odd
- call-consistent: no predicate depends oddly on itself.
- hierarchical: no form of recursion is allowed
- allowedness: at least every variable occurring in a clause must occur in at least one positive literal of the body

Stratified LPs for which the rules do not have recursion through negation have been defined by [ABW88]. The predicates of stratified LPs can be placed into strata so that one can compute over the strata. The model-theoretic semantics, the *supported Herbrand model* M_P^{supp}, is defined by declaring M_P^{supp} as the intended model among all minimal Herbrand models of $comp(P)$ which could be obtained by iterating over the strata. Przymusinski [Prz88] showed that the selected model was the so-called *perfect model*. The semantics of definite and stratified LPs lead to the unique minimal model semantics which is generally accepted to be the semantics for these classes of LPs. However, this is not the case for more expressive LPs. Here are several possible ways to determine the semantics and various approaches based on extensions of the 2-valued classical

logic to three-valued logics have been proposed, e.g., Fitting [Fit85] or Kunen [Kun87] semantics which are based on Kleene's strong three-valued logics, or the *well founded semantics* (WFS) [VGRS91] which is an extension of the prefect model semantics. Another approach is based on the tradition of non-monotonic reasoning in which the definition of entailment is based on the notion of beliefs. The *stable model semantics* (STABLE) [GL88] is based on this approach. For a discussion of the relationships between non-monotonic theories and logic programming see [Min93]. In the following I will briefly review the (declarative) semantics and theory of more expressive types of LPs. A lot of different semantics have been defined in the past. Table 3.1 gives an incomplete overview. I do not want to present all semantics or discuss there merits and shortcomings in the context of SLA representation. Instead, I will mainly focus on well-founded semantics (WFS) which is the declarative semantics of choice in this dissertation. However, since the proposed ContractLog KR for representing SLAs is intended to be general and applicable to several semantics I also briefly review stable model semantics (STABLE) for normal LPs and its extension answer set semantics (ASS) for extended LPs; two other prominent semantics in declarative logic programming.

Stable Model Semantics The Gelfond-Lifschitz transformation P^M [GL88] of a normal LP P wrt to its interpretation I is obtained from the ground instance $ground(P)$ of P as follows:

Definition 50 ***(Gelfond-Lifschitz transform)*** *Let P be a program and $M \subseteq B_P$. The Gelfond-Lifschitz transform P^M of P (aka reduct of P) wrt M is defined by $P^M = r^M | r \in ground(P)$. It is obtained from $ground(P)$ by:*

1. *Replace in every ground rule $A \leftarrow B \wedge notC \in ground(P)$ the negative body by its truth value wrt M.*
2. *Deleting each rule r in P with $B^-(r) \cap M \neq \emptyset$ where B^- denotes the set of negated atoms in the body of the rule r.*

Based on P^M the concepts of stable models [GL88] and partial stable models [Prz91] have been defined:

Definition 51 ***(Stable Model)*** *An interpretation I of a normal LP P is a stable model M^{Stable} of P if I is a minimal model of P^M:*

$$SEM_{Stable}(P) = \bigcap_{M^{Stable} \in SEM_{Stable}(P)} (M^{Stable} \cup neg(B_P \setminus M^{Stable}))$$

Definition 52 ***(Partial Stable Model)*** *A partial Herbrand interpretation is called a partial stable model of P if it is a partial minimal model of P^M.*

Table 3.1: Semantics for LP Classes (adapted from [Dix95b])

Class	Semantics	Ref.
Definite LPs	Least Herbrand model: M_p	[ABW88]
Stratified LPs	Supported Herbrand model: M_p^{supp}	[ABW88]
Normal LPs	Clark's Completion: $COMP$	[Cla78]
	3-valued Completion: $COMP_3$	[Kun87, Fit85]
	Well-founded Semantics: WFS	[VGRS91]
	WFS^+ and WFS'	[Dix92]
	WFS_C	[Sch92]
	Strong Well-founded Semantics: WFS_E	[CK91]
	Stable Model Semantics: $STABLE$	[GL88]
	Generalized WFS: $GWFS$	[BLM90]
	$STABLE^+$	[Dix95a]
	$STABLE_C$	[Sch92]
	$STABLE^{rel}$	[Dix92]
	Pereira's $O-SEM$	[PAA92]
	Partial Model Semantics: $PARTIAL$	[SZ91]
	Regular Semantics: $REG-SEM$	[YY90]
	Preferred Semantics: $PREFERRED$	[Dun91]
Extended LPs	Extended Well-founded Semantics: WFS_S	[HY91]
	Answer Set Semantics: ASS	[GL90, GL91]
	Extended Well-founded Semantics: $WFSX$	[PA92]
General Disjunctive	Disjunctive WFS: $DWFS$	[BD97]
	Generalized Disjunctive WFS: $GDWFS$	[BLM92]
	Disjunctive Stable: $DSTABLE$	[Prz91]
Stratified Disjunctive	Perfect model $PERFECT$	[Prz88]
	Weakly Perfect: $WPERFECT$	[PP88]
	Generalized Closed World Assumption: $GCWA$	
Positive Disjunctive	Weak generalized closed world assumption: $WGCWA$	[RLM89]

It can be shown that stable models are always partial models and that every stratified LP P has a unique stable model where stratified and stable semantics coincide.

Answer Set Semantics Gelfond and Lifschitz [GL90, GL91] have extended the concept of stable models to extended and disjunctive LPs based on the notion of answer sets. The proposed answer-set semantics is defined as follows:

Definition 53 *(**Answer Set Semantics**) Let P be an extended (disjunctive) LP. P is transformed to a (explicit) negation-free program P' by replacing all negative literals $\neg A$ by positive literals A' over new predicate symbols. Every stable model M^{Stable} of P' defines and answer set of P, which is a set of literals:*

$$\overline{L} = A \in B_P | M^{Stable}(A) = t \cup \neg A \in \neg B_P | M^{Stable}(A') = t$$

If $\overline{L}$ does not contain complementary pairs $A, \neg A$ of literals, then the answer set is $\overline{L}$ else it is $B_P \cup \neg B_P$ is the set of all ground literals.

Associated with SEM_{Stable} are two entailment relations:

Definition 54 *(**Cautious Entailment**) An extended LP P cautiously entails a ground atomic formula a iff $a \in I$ for every answer set M^{Stable} of P.*

Definition 55 *(**Brave Entailment**) An extended program P bravely entails a ground atomic formula a iff $a \in I$ for some answer set M^{Stable} of P*

Well-founded Semantics There exists several definitions to well-founded semantics (WFS), e.g., [VGRS91, Fit90, BS91, Prz90b]. Van Gelder, Ross and Schilpf [VGRS91] were the first to extend the work of Apt et al. [ABW88] to the class of normal logic programs. The well-founded semantics (WFS) of Gelder et al. is a three-valued logic: *true*, *false* and *unknown*. WFS is an extension of the prefect model semantics, in contrast to Fitting and Jacob's semantics which is based on Kleene's strong three valued logic. For instance, WFS (as well as prefect model semantics) assigns the truth value "false" to a clause $p \leftarrow p$ while Fitting and Jacob assign "unknown". I follow the definition from [VGRS91]:

Definition 56 *(**Partial Interpretation**) Let P be a normal LP. A partial interpretation I is a set of ground literals such that for no atom A both A and not A are contained in I, i.e., $pos(I) \cap neg(I) = \emptyset$ and whose atoms are contained in B_P of P, i.e., $pos(I) \cup neg(I) \subseteq B_P$. I is a total interpretation, if I is a partial interpretation and for every atom $A \in B_P$ it contains A or not A, i.e., $pos(I) \cup neg(I) = B_P$.*

Definition 57 *(**Unfounded Set**) Let P be a normal LP. Let I be a partial interpretation. Let $\alpha \subseteq B_P$ be a set of ground atoms. α is an unfounded set of P wrt I, if for every atom $A \in \alpha$ and every ground rule instance $A \leftarrow \beta \in ground(P)$ at least one of the following conditions holds:*

1. *at least one body literal $L \in \beta$ is false in I.*
2. *at least one positive body literal $B \in \beta$ is contained in α.*

Definition 58 *(**Greatest Unfounded Set**) Let P be a normal LP. Let I be a partial interpretation. The greatest unfounded set of P wrt I is the union of all unfounded sets of P wrt I.*

Definition 59 *(**Pos. and Neg. Immediate Consequences**) For a ground normal LP P and a partial interpretation $I \subseteq B_P$ the following monotonic transformation operators are defined:*

- $T_P(I) := A \in B_P | \exists (A \leftarrow \beta) \in ground(P) : \beta \subseteq I$
- $U_P(I) :=$ *the greatest unfounded set of P wrt I*
- $W_P(I) := T_P(I) \cup \sim U_P(I)$

Lemma 1 *T_P, U_P and W_P are monotonic operators.*

Theorem 3 *Let P be a normal LP. For every countable ordinal α, $W_P \uparrow \alpha$ is a partial model of P.*

Definition 60 *(**Well-founded Model**) The least fixpoint of W_P is the well-founded (partial) model of P denoted W_P^*. The least fixpoint can be computed as follows, $lfp(W_P) = W_P^\infty(\emptyset)$[3]. If $lfp(W_P) \subseteq B_P$ is a total interpretation of P then $lfp(W_P)$ is a well-founded model. An atom $A \in B_P$ is well-founded (resp. unfounded) wrt P iff A (resp. $\neg A$) is in $lfp(W_P)$.*

WFS is defined for the grounding of an arbitrary normal LP: $ground(P)$, i.e., it defines a mapping SEM_{WFS}, which assigns to every normal LP P a set $SEM_{WFS}(P)$ of (partial) models of P such that $SEM_{WFS}(P) = SEM_{WFS}(ground(P))$ (i.e., SEM_{WFS} is instantiation invariant).

Definition 61 *(**Well-founded Semantics**) The Well-founded semantics (WFS) assigns to every normal LP P the well-founded partial model W_P^* of P:*

$SEM_{WFS}(P) := \{W_P^*\}$.

Definition 62 *(**Entailment**) A normal LP P entails a ground atom a under WFS, denoted by $P \models a$, if it is true in $SEM_{WFS}(P)$.*

WFS can be considered an approximation of stable models, i.e., if a program has stable models, then if an atom is true resp. false wrt the WFS then it is true resp. false wrt STABLE. [Prz90b] Moreover, for weakly stratified LPs [PP90] WFS coincides with STABLE. However, there are three important distinction between STABLE and WFS:

1. WFS is a three-valued semantics, whereas STABLE is two-valued.
2. every normal LP has exactly one WFS model, whereas every normal LP has zero or more stable models.
3. Irrelevant clauses (tautologies) lead to the non existence of stable models, e.g., $p \leftarrow \neg p$ has no stable model.

There have been also several proposal for extending WFS by classical negation leading to a well-founded semantics for extended LPs - see e.g., [Dun91, Dun93, BG94, Lif96, PA92, Bre96].

Existing procedural semantics for the computation of the well-founded model can be divided into two groups: (1) *bottom-up approaches* such as the alternating fixpoint approach [VG89, VG93, LT01], the magic set approach [Ros94, KSS95, Mor96, SS97] and transformation based (aka residual program) approaches [Bry90a, DK89, BD98, BDZ01] and (2) *top down approaches* such as *non-tabling based approaches* such as Global SLS resolution [Prz89a, Ros92] or or tabling-based approaches such as extensions to OLDT resolution [TS86], e.g., WELL [BL90], XOLDTNF [CW92] or the approach of Bol and Degerstedt [BD93], SLT resolution [SYY02] or the well-known SLG resolution [CSW95] (another prominent extension of OLDT). There are also some proof procedures for well-founded semantics for extended logic programs (WFSX) such as [Teu93] or SLX resolution [ADP94].

3.4 Description Logics and Semantic Web Ontology Languages

Description Logics [NBB+02] are notations that are designed to make it easier to describe conceptual definitions and properties of categories and enable reasoning with them. They are a family of KR formalisms for describing ontological knowledge, and play an important role for building the Semantic Web [W3C01, BL99, BLHL01, FWHH02]. In this section I describe the relevant notions and concepts of Description Logics and their relationships to ontologies (a formally specified vocabulary) and the Semantic Web. I first introduce Description Logics and then show how they can be used as ontology languages for the Semantic Web.

In the context of electronic contract representation, in particular contracts for IT services provided on the Semantic Web, such as Web Services, Grid Services or Internet ASP applications, ontology languages are useful to describe meta data vocabularies and semantic domain descriptions. That is, ontologies are

used as contract vocabularies to flexibly enrich the core rule syntax with domain-specific terminologies and given the language constructs a precise meaning by the description logics semantics.

3.4.1 Description Logics

Description Logics (DLs) [NBB+02] are a family of KR languages that can be used to represent the conceptual knowledge of a domain and model the relations between conceptual objects. They have been proved useful in a wide range of applications such as databases [CGL98, CLN98], configuration [MW98] and ontology engineering.

A description logic (DL) is a subset of first order logic (with equality) where expressions are built from unary predicates (referred to as concepts) and binary relations (called roles) using the concept and role constructors provided by the particular DL and the logical constructs such as conjunction ($\wedge$), negation ($\neg$), existential restriction ($\exists R.C$), value restriction ($\forall R.C$) or number restriction ($\geq nR$). The naming scheme for DLs, extends the basic AL family with symbols [NBB+02] denoting e.g., additional concept constructors, role constructor or restrictions on role interpretations, i.e., it distinguishes typographically between certain expressive extensions. ALC comprises conjunction, disjunction, negation, existential and universal quantification. Extended with transitive roles it becomes ALC_{R+} which is abbreviated with S. *SIN* extends S (i.e., ALC_{R+}) with number restrictions and inverse roles. *SHIQ* extends S with role hierarchies, inverse roles and qualified number restrictions and *SHOIN* introduces nominals and unqualified number restrictions.

3.4.1.1 Syntax of *SHIF(D)* and *SHOIN(D)*

In this subsection I recall the elements of the DLs *SHIF(D)* and *SHOIN(D)* from [HPS04] which are the DLs underlying the Semantic Web languages OWL Lite and OWL DL. I start with the DL $SHOIN(D)$.

Definition 63 *(**Datatype Theory**) In description logics, a datatype theory D is a mapping from a set of datatypes to a set of values, the domain of D, plus a mapping from data values to their denotation which must be one of the set of values.*

Given a datatype theory D, let A, R_A, R_D and I be nonempty finite and pairwise disjoint sets of atomic concepts, abstract roles, datatype roles and individuals. R_A^- denotes the set of all inverses R^- of abstract roles $R \in R_A$.

Definition 64 *(**Roles, Concepts, Axioms**) A role in SHOIN(D) is an element of $R \cup R^-$. The set of SHOIN(D) roles is $R_A \cup R_A^- \cup R_D$.*
The set of concepts in SHOIN(D) is the smallest set that can be built using the

constructors in figure 3.2 which are taken from [HPS04].
Axioms (see fig. 3.2) in SHOIN(D) allow the expression of relations between concepts, roles and individuals.

Due to different reasoning tasks (T-Box reasoning and A-Box reasoning) and their optimized implementation, axioms are partitioned into *terminological axioms* which define properties of concepts and roles and *assertional axioms* which define assertions on individuals. [GL96, NBB+02]

Definition 65 ***(Knowledge Base)*** *A knowledge base Ψ in SHOIN(D) is a finite set of SHOIN(D) axioms.*

A description logic knowledge bases (DL KB) might be separated into a terminology (T-box model) in which the set of concepts and roles are defined and assertions on individual objects (A-box model).

SHIF(D) is the restriction of *SHOIN(D)* which excludes the *oneOf* constructor and limits the *atleast* and *atmost* constructors to 0 and 1.

3.4.1.2 Semantics of *SHIF(D)* and *SHOIN(D)*

The semantics of *SHOIN(D)* is defined in analogy to the semantics of first-order logic according to [HPS04].

Definition 66 ***(Interpretation and Satisfaction)*** *An interpretation of a Description Logic such as SHOIN(D) is a pair $I = \langle \triangle, \bullet^I \rangle$ where $\triangle$ is a nonempty set, called the (abstract) domain, and $\bullet^I$ is a mapping, called the interpretation function, which assigns to each atomic concept from A a subset of $\triangle$, to each individual $o \in I$ an element of $\triangle$ and to each atomic role from R a subset of $\triangle \, x \, \triangle$. An interpretation I satisfies a SHOIN(D) axiom F, or I is a model of F, i.e., $I \models F$, under the conditions given in fig. 3.2. I satisfies a KB Ψ iff it satisfies each axiom in Ψ, i.e., I is a model of Ψ: $I \models \Psi$. Ψ is satisfiable (unsatisfiable) iff there exists (does not exist) a model of Ψ. A concept C is satisfiable wrt to Ψ iff there is a model I of Ψ with $C^I \neq \emptyset$. A concept C is subsumed by a concept D wrt Ψ iff $C^I \sqsubseteq D^I$ in every model I of Ψ. Two concepts are said to be equivalent wrt Ψ iff they subsume each other wrt Ψ. A KB Ψ_1 entails a KB Ψ_2 iff every model of Ψ_1 is also a model of Ψ_2.*

DL-reasoner provide their users various inference capabilities that deduce implicit knowledge from the explicitly represented terminology and assertions:

- *Concept Subsumption inference* determines subclass relationships such as concept C is subsumed by D iff all instances of C are necessarily instances of D wrt to a knowledge base Ψ if $C^I \subseteq D^I$ for every model I of Ψ. Concept subsumption can be reduced to concept satisfiability, i.e., a concept C is satisfiable wrt a DL KB Ψ if there exists a model I of Ψ such that C^I is nonempty.

Constructor Name	Syntax	Semantics
atomic concept **A**	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
datatype **D**	D	$D^{\mathbf{D}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
abstract role $\mathbf{R}_A$	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
datatype role $\mathbf{R}_{\mathbf{D}}$	U	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
individuals **I**	o	$o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
data values	v	$v^{\mathcal{I}} = v^{\mathbf{D}}$
inverse role	R^-	$(R^-)^{\mathcal{I}} = (R^{\mathcal{I}})^-$
top	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
bottom	$\bot$	$\bot^{\mathcal{I}} = \{\}$
conjunction	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
disjunction	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
oneOf	$\{o_1, \ldots, o_n\}$	$\{o_1, \ldots, o_n\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$
exists restriction	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y. \langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$
value restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y. \langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
atleast restriction	$\geqslant n\, R$	$(\geqslant n\, R)^{\mathcal{I}} = \{x \mid \sharp(\{y. \langle x, y\rangle \in R^{\mathcal{I}}\}) \geqslant n\}$
atmost restriction	$\leqslant n\, R$	$(\leqslant n\, R)^{\mathcal{I}} = \{x \mid \sharp(\{y. \langle x, y\rangle \in R^{\mathcal{I}}\}) \leqslant n\}$
datatype exists	$\exists U.D$	$(\exists U.D)^{\mathcal{I}} = \{x \mid \exists y. \langle x, y\rangle \in U^{\mathcal{I}} \text{ and } y \in D^{\mathbf{D}}\}$
datatype value	$\forall U.D$	$(\forall U.D)^{\mathcal{I}} = \{x \mid \forall y. \langle x, y\rangle \in U^{\mathcal{I}} \rightarrow y \in D^{\mathbf{D}}\}$
datatype atleast	$\geqslant n\, U$	$(\geqslant n\, U)^{\mathcal{I}} = \{x \mid \sharp(\{y. \langle x, y\rangle \in U^{\mathcal{I}}\}) \geqslant n\}$
datatype atmost	$\leqslant n\, U$	$(\leqslant n\, U)^{\mathcal{I}} = \{x \mid \sharp(\{y. \langle x, y\rangle \in U^{\mathcal{I}}\}) \leqslant n\}$
datatype oneOf	$\{v_1, \ldots\}$	$\{v_1, \ldots\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, \ldots\}$
Axiom Name	**Syntax**	**Semantics**
concept inclusion	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
object role inclusion	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
object role transitivity	$\mathsf{Trans}(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$
datatype role inclusion	$U_1 \sqsubseteq U_2$	$U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$
individual inclusion	$a : C$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
individual equality	$a = b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
individual inequality	$a \neq b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

Figure 3.2: Syntax and semantics of *SHOIN(D)* [HPS04]

- *Instance inference* determines instance relationships such as the individual a is an instance of concept C iff a is always interpreted as an element of C, i.e., $a^I \in C^I$ fore every model I of a KB Ψ.
- *Consistency inference* determines whether a knowledge base (T + A box) is non-contradictory. Knowledge base satisfiability, i.e., a DL KB Ψ is satisfiable if there is an interpretation of Ψ which is a model of Ψ.

As shown in [BDS93, NBB+02, HPS04] the inference problem of concept satisfiability, concept subsumption and instance checking can be reduced to a check of knowledge base (un)satisfiability.

3.4.2 Semantic Web Ontology Languages

The goal of this subsection is to sketch on a more or less informal level what the Semantic Web is and why it needs ontologies based on description logics (in combination with rules).

3.4.2.1 Semantic Web and Ontologies

The Semantic Web [W3C01, BL99, BLHL01, FWHH02] renews the idea of heterogenous information systems [PH02], hierarchical data storage and the distributed network database model and uses XML and RDF as core languages. It aims for machine-understandable Web resources, whose additional background knowledge can be shared and processed by automated tools, e.g., to answer search queries and by human users, e.g., to get additional meta information. Emerging Semantic Web standards such as the resource description framework (RDF) [Hay04, KC04] and ontology languages such as RDFS [BG04b] and OWL (web ontology language) [MH04] make creating a syntactic format specifying background knowledge / meta data for information resources possible. They are powerful KR languages to represent knowledge in a machine understandable fashion based on a simple data model using linked resources. While RDF is a simple language to express binary relationships between resources identified by URIs, OWL can be used to define vocabularies (ontologies) which are then used to describe resources. Standard vocabularies such as Dublin Core (`http://dublincore.org/`) which have been given a RDF syntax can be used to attach meta data to Web resources. The biggest advantages of using semantic web ontology mark-up languages such as RDFS or OWL to represent knowledge are their automated reasoning capabilities about ontologies due to the formal semantics and their openness: once a consensus has been reached about the vocabulary used to represent resources using URIs, new knowledge can be easily added by adding new statements about these resources, i.e., the Semantic Web as a mesh of information can take advantage of the formal semantics and open world assumption [PSHH04] of these expressive languages.

The architecture of the Semantic Web was thought by as a hierarchy of languages. Figure 3.3 illustrates the well-known "Semantic Web Stack" [BL03].

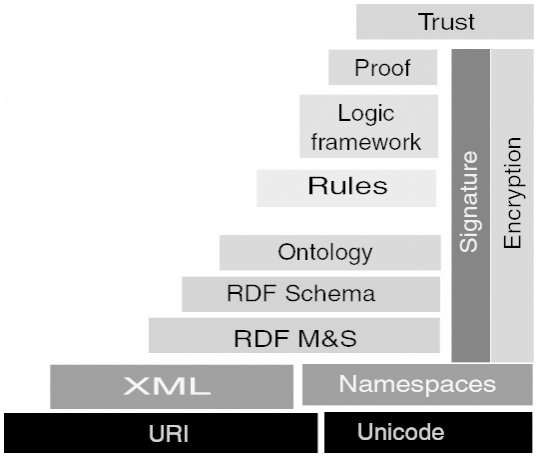

Figure 3.3: Semantic Web Stack [BL03]

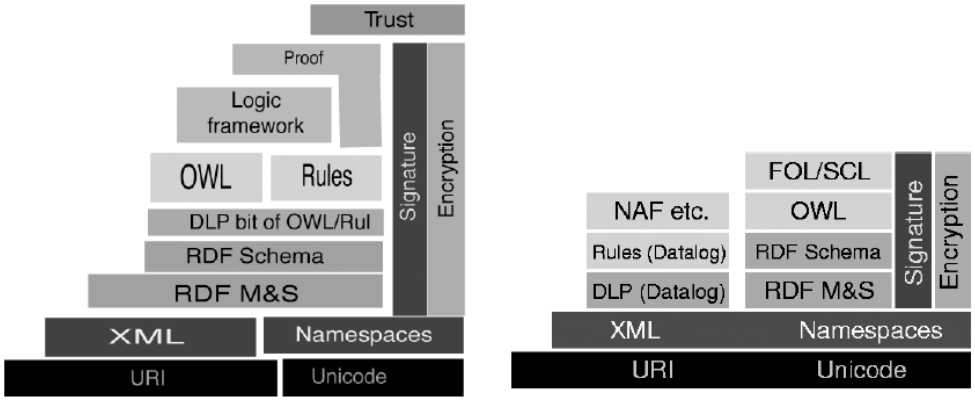

Figure 3.4: Alternative Semantic Web Architectures [BL05, HPPS+05]

Currently, the ontology layer with OWL is the highest layer of sufficient maturity in the Semantic Web stack, but increasing interest in industry and academia in more sophisticated KR technologies and automated reasoning capabilities led to much recent development towards the integration of rules and ontologies, in order to evolve the *Rules* and *Logic* layer. It has been recently argued that extending the Semantic Web with rules might lead to a stack where rules and ontologies sit side by side on top of a layer labelled as the "DLP bit of OWL/Rules" [BL05] (DLP = description logic programs) or two towers in the Semantic Web stack [HPPS+05] - see figure 3.4.

In section 4.2 I will further elaborate on the combination of ontologies and rules and introduce my approach of a hybrid DL-typed rule language in the ContractLog KR.

The term ontology originates from philosophy, where it is a theory about the nature of existence. Ontology in artificial-intelligence (AI) literature contains

many definitions, which are often contradicting. According to Gruber [Gru93] an ontology is: "*A explicit formal specification of the terms in the domain and relations among them*". For the purpose of this dissertation I state more precisely:

Definition 67 *(**Ontology**) An ontology (aka domain vocabulary) is a formal explicit description of concepts in a domain of discourse consisting of* classes *(aka concepts),* properties *(aka roles or slots) describing features and attributes of the concepts and* restrictions *on properties (aka role restrictions or facets). The classes which describe concepts in the domain are often specified as* taxonomies *with transitive parent-subclass relationships. An ontology together with a set of individual instances of classes (aka* individuals*) constitutes a (ontological)* knowledge base.

Ontologies are typically used to define common vocabularies for domains which need shared information, as for example in the domain of distributed contract specifications in open environments such as the (Semantic) Web. The major advantages are [Pas05d]:

- To share a common understanding about the domain
- To enable reuse of domain knowledge
- The make domain assumption explicit
- To separate domain knowledge from the operational knowledge
- To analyze domain knowledge

This, obviously, requires a well-designed, well-defined and Web-compatible ontology language with support for reasoning tools. Therefore, the Semantic Web generally builds on syntaxes which use URIs to represent data, usually in triples based structures based on the RDF syntax (or XML/RDF syntax), and expressive ontology languages such as RDFS or OWL which are used to define ontologies/vocabularies. In the following two subsections I will summarize relevant parts of the material from [Hay04, KC04, BG04b, MH04].

3.4.2.2 Resource Description Framework (RDF) and Schema (RDFS)

Definition 68 *(**URI References, Blank Nodes, Literals**) A URI is simply a Web identifier. Let U denote the set of URI references, B denote the set of blank nodes (i.e., existentially quantified variables) and L denote the set of literals (i.e., data values such as strings or integers), where L is the union of the set L_p of plain literals and the set of L_t typed literals. A typed literal consists of a lexical form s and a datatype URI t written as $s : t$, i.e., s has type t. The sets U, B, L_p and L_t are pairwise disjoint. A vocabulary is a subset of $U \cup L$. T denotes the set of all RDF terms, i.e., $T = U \cup B \cup L$.*

Definition 69 *(**RDF Graph, statements, instances**) An RDF graph, or simply a graph, is a set of RDF triples. A subgraph of an RDF graph is a subset of the triples in the graph. A ground RDF graph is one with no blank nodes. The elements s (subject), p (property) and o (object) of a RDF graph are called RDF statements or RDF triples of the form (spo). RDF graphs require properties to be URI references. The set of RDF terms T of a RDF graph G is $T(G) = \pi_1(G) \cup \pi_2(G) \cup \pi_3(G)$ where π_i is a projection mapping. The set of blank nodes of a RDF graph G is denoted by $bl(G) = T(G) \cap B$. The vocabulary of a RDF graph G is the set $V(G) = T(G) \cap (U \cup L)$. Two RDF graphs G and G′ are equivalent if there is a bijection f: $T(G) \rightarrow T(G')$ such that $f(bl(G)) \subseteq bl(G')$ and $f(v) = v$ for each $v \in V(G)$ and $spo \in G$ iff $f(s)f(p)f(o) \in G$. An instance i of a RDF graph G is the graph G_i obtained from G by replacing the blank nodes v in G by $M(v)$, where M is a mapping from a set of blank nodes to some set of literals, blank nodes and URI references.*

Definition 70 *(**Interpretation**) A (simple) interpretation I of a vocabulary V is a 6-tuple $I = \langle R_i, P_I, E_I, S_I, L_I, LV_I \rangle$, where R_I is a nonempty set of resources, P_I is the set of properties, LV_I is the set of literal values, $E_I := P_I \rightarrow P(R_I x R_I)$, $S_I := V \cap U \rightarrow R_I \cup P_I$ and $L_I := V \cap L_t \rightarrow R_I$, where $P(X)$ is the power set of a set X. An interpretation I denotes a function with domain V such that:*

- $I(l))l \in LV_I$ *for* $l \in L_p \cap V$.
- $I(l) = L_I(l)$ *for* $l \in L_t \cap V$.
- $I(a) = S_I(a)$ *for* $a \in U \cap V$.

Definition 71 *(**Satisfaction**) Let $E = (spo)$ a ground triples and I an interpretation for a vocabulary V then I satisfies E if $(spo) \in V$, $I(p) \in P_I$ and $(I(s), I(o)) \in E_I(I(p))$. If G is a ground RDF graph, then I satisfies G if I satisfies each triple $E \in G$. If a function $A(v)$ is defined for $v \in B$, then a function I_A is defined that extend I by using A to interpret blank nodes in the domain of A and $I_A(v) = A(v)$. I satisfies G if I_A satisfied G for some function $A : bl(G) \rightarrow R_I$, i.e., $I_A(o) \in P_I$ and $(I_A(s), I_A(o)) \in E_I(I_A(p))$ for each $(spo) \in G$. I satisfies S, where S is a set of RDF graphs, if I satisfies $M(S)$.*

Definition 72 *(**Entailment**) I satisfies E if $I(E) = true$, and a set S of RDF graphs (simply) entails a graph E if every interpretation which satisfies every member of S also satisfies E.*

Figure 3.5 lists the entailment rules for RDFS entailment defined in [Hay04] (prefixes are omitted).

Figure 3.5: RDFS entailment rules [Hay04]

If G contains	where	then add to G
$v\,p\,l$	$l \in L$	$v\,p\,b_l$
$v\,p\,b_l$	$l \in L$	$v\,p\,l$
$v\,p\,w$		p type Property
$v\,p\,l$	$l \in L_p$	b_l type Literal
p domain u		
$v\,p\,w$		v type u
p range u		
$v\,p\,w$	$w \in U \cup B$	w type u
$v\,p\,w$		v type Resource
$v\,p\,w$	$w \in U \cup B$	w type Resource
v subPropertyOf w		
w subPropertyOf u		v subPropertyOf u
v type Property		v subPropertyOf v
p subPropertyOf q		
$v\,p\,w$	$q \in U \cup B$	$v\,q\,w$
v type Class		v subClassOf Resource
v subClassOf w		
u type v		u type w
v type Class		v subClassOf v
v subClassOf w		
w subClassOf u		v subClassOf u
v type Container-MembershipProperty		v subPropertyOf member
v type Datatype		v subClassOf Literal

3.4.2.3 Web Ontology Language (OWL)

The Web Ontology Langauge (OWL) [MH04, PSHH04] is a W3C recommendation standard for the ontology layer of the Semantic Web. It consists of three sublanguages: *OWL Lite*, *OWL DL* and *OWL Full*. The logic foundation of OWL Lite and OWL DL are the expressive description logics (DL) *SHIF(D)* (for OWL Lite) and *SHOIN(D)* (for OWL DL) [HPS04] (see section 3.4.1) which have been given a RDF / XML syntax (there are also other syntaxes such as N3). As pointed out in [BHS03, NBB+02] due to the research and process of description logics and DL system in the last decades DLs now provide enough expressiveness while staying decidable and hence qualify as adequate ontology languages for ontologies for the Semantic Web. OWL is defined as an extension to RDF in the form of vocabulary entailment (see previous section), i.e., the syntax of OWL is the syntax of RDF and the semantics of OWL is an extension of the semantics of RDF. Although there are differences between OWL and description logics (OWL is based on RDF and hence on RDF graphs and RDF objects/classes), it is possible to define an abstract syntax which is close to description logics and define a direct model-theoretic semantics (in addition to the RDF-compatible semantics) [PSHH04], which is fairly standard by description logics. I have already outlined the main concepts of DL semantics and RDF semantics. For the specific details of the OWL semantics I refer to the OWL documentation [PSHH04]. As it has been shown in [HPS04] OWL DL (resp. OWL Lite) entailment can be translated into *SHOIN(D)* (resp. *SHIF(D)*) unsat-

Figure 3.6: Translation from OWL-DL to *SHOIN(D)* [HPS04]

OWL fragment F	Translation $\mathcal{V}(F)$
A, OWL class name	A
B, OWL datatype name	B
R, OWL object property name	R
T, OWL datatype property name	T
o, OWL individual name	o
v, OWL data value	v
intersectionOf($C_1 \ldots C_n$)	$\mathcal{V}(C_1) \sqcap \ldots \sqcap \mathcal{V}(C_n)$
unionOf($C_1 \ldots C_n$)	$\mathcal{V}(C_1) \sqcup \ldots \sqcup \mathcal{V}(C_n)$
complementOf(C)	$\neg\mathcal{V}(C)$
oneOf($o_1 \ldots o_n$)	$\{\mathcal{V}(o_1), \ldots, \mathcal{V}(o_n)\}$
restriction($R\ r_1\ r_2 \ldots r_n$)	$\mathcal{V}$(restriction($R\ r_1$)) $\sqcap \ldots \sqcap \mathcal{V}$(restriction($R\ r_n$))
restriction(R allValuesFrom(C))	$\forall\mathcal{V}(R).\mathcal{V}(C)$
restriction(R someValuesFrom(C))	$\exists\mathcal{V}(R).\mathcal{V}(C)$
restriction(R value(o))	$\exists\mathcal{V}(R).\{\mathcal{V}(o)\}$
restriction(R minCardinality(n))	$\geqslant n\ \mathcal{V}(R)$
restriction(R maxCardinality(n))	$\leqslant n\ \mathcal{V}(R)$
restriction(R cardinality(n))	$\geqslant n\ \mathcal{V}(R) \sqcap \leqslant n\ \mathcal{V}(R)$
restriction($T\ r_1\ r_2 \ldots r_n$)	$\mathcal{V}$(restriction($T\ r_1$)) $\sqcap \ldots \sqcap \mathcal{V}$(restriction($T\ r_n$))
restriction(T allValuesFrom(D))	$\forall\mathcal{V}(T).\mathcal{V}(D)$
restriction(T someValuesFrom(C))	$\exists\mathcal{V}(T).\mathcal{V}(D)$
restriction(T value(v))	$\exists\mathcal{V}(T).\{\mathcal{V}(v)\}$
restriction(T minCardinality(n))	$\geqslant n\ \mathcal{V}(T)$
restriction(T maxCardinality(n))	$\leqslant n\ \mathcal{V}(T)$
restriction(T cardinality(n))	$\geqslant n\ \mathcal{V}(T) \sqcap \leqslant n\ \mathcal{V}(T)$
oneOf($v_1 \ldots v_n$)	$\{\mathcal{V}(v_1), \ldots, \mathcal{V}(v_n)\}$

isfiability. The basic process works as follows: (1) An entailment between OWL DL ontologies is translated into an entailment between *SHOIN+(D)* knowledge bases, where "+" denotes concept existence axioms. (2) *SHOIN+(D)* entailment is transformed into unsatisfiability of *SHOIN(D)* KBs. Figure 3.6 taken from [HPS04] shows the translation from OWL-DL classes and names to *SHOIN(D)*.

3.5 KR Event / Action Logics and Active Database Technologies

Event-driven applications based on reactive rules and in particular ECA rules which trigger actions as a response to the detection of events have been extensively studied during the 1990s. Stemming from the early days of programming language where system events were used for interruption and exception handling, active event-driven rules have received great attention in different areas such as active databases [WC96, Pat99] which started in the late 1980s, real-time applications and system and network management tools which emerged in the early 1990s as well as publish-subscribe systems [BCV03] which appeared

in the late 1990s. Recently, there has been an increased interest in industry and academia in event-driven mechanisms, complex event processing, event streaming and high-level Event-Driven Architectures (EDA). (Pro-)active real-time or just-in-time reactions to complex events are a key factor in upcoming agile and flexible IT service infrastructures, distributed loosely coupled service oriented environments or new business models such as On-Demand or Utility computing. Industry trends such as Real-Time Enterprise (RTE), Complex Event Processing (CEP), Business Activity Management (BAM) or Business Performance Management (BPM) and closely related areas such as Service Level Management (SLM) with monitoring and enforcing Service Level Agreements (SLAs) are business drivers for this renewed interest. Another strong demand for event processing functionalities comes from the web community, in particular in the area of Semantic Web and Rule Markup Languages (e.g., RuleML, Reaction RuleML, RIF).

Active databases in their attempt to combine techniques from expert systems and databases to support automatic triggering of rules in response to events and to monitor state changes in database systems have intensively explored and developed the ECA paradigm and event algebras to compute complex events. In a nutshell, this paradigm states that an ECA rule autonomously reacts to actively or passively detected simple or complex events by evaluating a condition or a set of conditions and by executing a reaction whenever the event happens and the condition(s) is true: "*On Event if Condition do Action*".

A different approach to events and actions - the so called KR event/action logics - which has for the most part proceeded separately has the origin in the area of artificial intelligence (AI), knowledge representation (KR) and logic programming (LP). Here the focus is on the development of axioms to formalize the notions of actions/events and causality, where events/actions are characterized in terms of necessary and sufficient conditions for their occurrences. Instead of detecting the events as they occur in active databases at a single point in time, the KR approach to events focuses on the inferences that can be made from the fact that certain events are known to have occurred or are planned to happen in future.

This leads to different views and terminologies on event/actions definitions and event/action processing in these domains, including also event communication aspects such as event notifications in push, pull or publish/subscribe style. Reaction rules are a key factor in active enforcement and automated management and maintenance of service contracts and reasoning on the effects of happened events and actions on the contract states based on a precise, verifiable and traceable semantics is crucial in SLA representation. I first give an overview on the various approaches in this orthogonal domain of knowledge representation in this section and then introduce relevant terms and concepts.

3.5.1 Overview

These subsections give an overview of event / action /state processing and reaction rules approaches and systems in different domains.

3.5.1.1 Active Databases and ECA Rule Systems

Active databases are an important research topic due to the fact that they find many applications in real world systems and many commercial databases systems have been extended to allow the user expressing active rules whose execution can update the database and trigger the further execution of active rules leading to a cascading sequence of updates (often modelled in terms of execution programs). Several active database systems have been developed, e.g., ACCOOD [Eri93], Chimera [MPC96], ADL [Beh95], COMPOSE [GJS92], NAOS [CC96], HiPac [DBC96]. These systems mainly treat event detection and processing purely procedural and often focus on specific aspects. In this spirit of procedural ECA formalisms are also systems such as AMIT [AO04], RuleCore [Rul06] or JEDI [CNF98]. Several papers discuss formal aspects of active databases on a more general level - see e.g., [PCFW95] for an overview. Several event algebras have been developed, e.g., Snoop [CKAK94], SAMOS [GD93], ODE [GJS92]. The object database ODE [GJS92] implements event-detection mechanism using finite state automata. SAMOS [GD93] combines active and object-oriented features in a single framework using colored Petri nets. Associated with primitive event types are a number of parameter-value pairs by which events of that kind are detected. SAMOS does not allow simultaneous atomic events. Snoop [CKAK94] is an event specification language which defines different restriction policies that can be applied to the operators of the algebra. Complex events are strictly ordered and cannot occur simultaneously. The detection mechanism is based on trees corresponding to the event expressions, where primitive event occurrences are inserted at the leaves and propagated upwards in the trees as they cause more complex events to occur.

There has been a lot of research and development concerning knowledge updates in active rules (execution models) in the area of active databases and several techniques based on syntactic (e.g., triggering graphs [AWH94] or activation graphs [BW94]) and semantics analysis (e.g., [BCRS97], [Wid92]) of rules have been proposed to ensure termination of active rules (no cycles between rules) and confluence of update programs (always one unique minimal outcome). The combination of deductive and active rules has been also investigated in different approaches manly based on the simulation of active rules by means of deductive rules [LLM98, Zan93, DKSW03].

3.5.1.2 Production Rule Systems

The treatment of active rules in active databases is to some extend similar to the forward-chaining production rules system paradigm in artificial intelligence

(AI) research [HR85]. In fact, triggers, active rules and integrity constraints, which are common in active DBMS, are often implemented in a similar fashion to forward-chaining production rules where the changes in the conditions due to update actions such as "assert" or "retract" on the internal database are considered as implicit events leading to further update actions and to a sequence of "firing" production rules, i.e.: "if Condition then Action". There are many forward-chaining implementations in the area of deductive databases and many well-known forward-reasoning engines for production rules such as ILOG's commercial jRules system, Fair Isaac/Blaze Advisor, CA Aion, Haley, ESI Logist or popular open source solutions such as OPS5, CLIPS or Jess which are based on the RETE algorithm. In a nutshell, this algorithm keeps the derivation structure in memory and propagates changes in the fact and rule base. This algorithm can be very effective, e.g., if you just want to find out what new facts are true or when you have a small set of initial facts and when there tend to be lots of different rules which allow you to draw the same conclusion. This might be also one reason why production rules have become very popular as a widely used technique to implement large expert systems in the 1980s for diverse domains such as troubleshooting in telecommunication networks or computer configuration systems.

Classical production rule systems and most database implementations of production rules [DE88, SLR93, WF90] typically have an operational or execution semantics defined for them, but lack a precise theoretical foundation and do not have a formal semantics. Although production rules might simulate derivation rules via asserting a conclusion as consequence of the proved condition, i.e., "if Condition then assert Conclusion", the classical production rule languages such as OPS5 are less expressive since they lack a clear declarative semantics and recursion, suffer from termination and confluence problems of their execution sequences and typically do not support expressive non-monotonic features such as classical or negation-as-finite failure or preferences, which makes it sometimes hard to express certain real life problems in a natural and simple way.

However, several extensions to this core production systems paradigm have been made which introduce e.g., negations (classical and negation-as-finite failure) [DM02] and provide declarative semantics for certain subclasses of production rules systems such as stratified production rules. It has been shown that stratified production systems have a declarative semantics defined by their corresponding logic program (LP) into which they can be transformed [Ras92] and that the well-founded, stable or preferred semantics for production rule systems coincide in the class of stratified production systems [DM02]. Stratification can be implemented on top of classical production rules in from of priority assignments between rules or by means of transformations into the corresponding classical ones. The strict definition of stratification for production rule systems has been further relaxed in [RL96] which defines an execution semantics for update rule programs based upon a monotonic fixpoint operator and a declarative semantics via transformation of the update program into a normal LP with stable model semantics.

Closely related are also logical update languages such as transaction logics and in particular serial Horn programs, where the serial Horn rule body is a sequential execution of actions in combination with standard Horn pre-/post conditions. [BK95] These serial rules can be processed top-down or bottom-up and hence are closely related to the production rules style of "condition → update action". Several approaches in the active database domain also draw on transformations of active rules into LP derivation rules, in order to exploit the formal declarative semantics of LPs to overcome confluence and termination problems of active rule execution sequences [BL96, Zan95, FG98]. The combination of deductive and active rules has been also investigated in different approaches mainly based on the simulation of active rules by means of deductive rules. [LLM98, Zan93] Moreover, there are approaches which directly build reactive rules on top of LP derivation rules such as the Event-Condition-Action Logic Programming language (ECA-LP) which enables a homogeneous representation of ECA rules and derivation rules [Pas06b].

3.5.1.3 Event Notification Systems, Complex Event Processing Systems and Reaction Rule Interchange Languages

Recently, a strong demand for complex event/action processing functionalities comes from industry (enterprise architectures, descission support systems and information management systems) and the web community, in particular in the area of Semantic Web and Rule Markup and the upcoming W3C Rule Interchange Language (e.g., RuleML [BT00], Reaction RuleML [PKB+06] or RIF [RIF05]). In distributed environments such as the (Semantic) Web with independent agent / system nodes and open or closed domain boundaries complex event processing is often done using event notification and communication mechanisms based on middleware products such as an enterprise service bus. Systems either communicate events in terms of messages according to a predefined or negotiated communication/coordination protocol [Pas06e] and possibly using a particular transport language such as Java Messaging Service (JMS), FIPA Agent Communication Language (FIPA-ACL), Common Base Event (CBE) [Ogl03] or the Web Services Simple Object Access Protocol (SOAP) or they subscribe to publishing event notification servers which actively distribute events (push) to the subscribed and listening agents. Typically the interest here is in the particular and complex event sequence or event processing workflow which possibly follows a certain communication or workflow-like coordination protocol, rather than in single event occurrences which trigger immediate reactions as in the active database trigger or ECA rules. As a result reactive rules are typically local to a particular context, e.g. within a particular conversation. That is, the communicated events contribute to the detection of a complex (local) event situation which triggers a local reaction within a context (a conversation waiting for at least three sequential answers or requests). Complex event processing and event notification closely relates to the the domain of workflow and choreography as e.g. specified by the business process execution language (BPEL).

3.5.1.4 Temporal KR Event / Action / Transition and Update Logic Systems

A fourth dimension to events and actions which has for the most part proceeded separately has the origin in the area of knowledge representation (KR) and logic programming (LP) with close relations to the formalisms of process and transition logics. Here the focus is on the development of axioms to formalize the notions of actions resp. events and causality, where events are characterized in terms of necessary and sufficient conditions for their occurrences and where events/actions have an effect on the actual knowledge states, i.e., they transit states into other states and initiate / terminate changeable properties called fluents. Instead of detecting the events as they occur as in the active database domain, the KR approach to events/actions focuses on the inferences that can be made from the fact that certain events are known to have occurred or are planned to happen in future. This has led to different views and terminologies on event/action definition and event processing in the temporal event/action logics domain. Reasoning about events, actions and change is a fundamental area of research in AI since the events/actions are pervasive aspects of the world in which agents operate enabling retrospective reasoning but also perspective planning. A huge number of formalisms for deductive but also abductive reasoning about events, actions and change have been developed. The common denominator to all this formalisms and systems is the notion of states aka fluents [San89b] which are changed or transit due to occurred or planned events/actions. Among them are the event calculus [KS86] and variants such as the interval-based Event Calculus (see section 4.7.4), the situation calculus [MH69, Rei01], features and fluents [San89b], various (temporal) action languages [GL93, FN71, GL98, GL99, DGKK98], fluent calculi [HS90, Thi99] and versatile event logics [BG04a]. Most of these formalisms have been developed in relative isolation and the relationships between them have only been partially studied, e.g., between situation calculus and event calculus or temporal action logics (TAL) which has its origins in the features and fluents framework and the event calculus.

Related and also based on the notion of (complex) events, actions and states with abstract models for state transitions and parallel execution processes are various process algebras like TCC [SRG96], CSS [Mil89] or CSP [Hoa85], (labelled) transition logics (LTL) and (action) computation tree logics (ACTL) [MKB00, MKB03]. Related are also update languages [LHL95, Zan93, NK88, AV91, LAP01, ABLP02, EFST01, Lei03, Pas06b, Pas05a] and transaction logics [BK95] which address updates of logic programs where the updates can be considered as actions which transit the initial program (knowledge state/base) to a new extended or reduced state hence leading to a sequence of evolved knowledge states. Many of these update languages also try to provide meaning to such dynamic logic programs (DLPs). However, unlike ECA languages and event notification/messaging systems approaches these languages typically do not provide complex event / action processing features and exclude external calls with side effects via event notifications or procedural calls.

3.5.2 Basic Concepts in Event and Action Processing

Definition 73 *(**Atomic Events**) A raw event (aka atomic or primitive event) is defined as an instantaneous (occurs in a specific point in time), significant (relates to a context), atomic occurrence (it cannot be further dismantled and happens completely or not at all): occurs(e, t), i.e., event e occurs at time point t.*

I distinguish between event instances (simply called events) which occur and event type pattern definitions.

Definition 74 *(**Event Type Pattern Definition and Event Instance**) An event type pattern definition (or simply event definition or event type) describes the structure of an (atomic or complex) event, i.e., it describes its detection condition(s). A concrete instantiation of a type pattern is a specific event instance, which is derived (detected) from the detection conditions defined within the rules body.*

An event instance can be, e.g., a particular fact becoming true, an external event in an monitored system, a communicated event message within a conversation, a state transition from one state to another such as knowledge updates, transactions (e.g., begin, commit, abort, rollback), temporal events (e.g., at 6 p.m., every 10 minutes), etc. Typically, events occur in a context that is often relevant to the execution of the other parts of the reaction rules, i.e., event processing is done within a context.

Definition 75 *(**Event Context**) A context can have different characteristics such as:*

- *Temporal characteristic designates information with a temporal perspective, e.g., service availability within one month or within 60 minutes from X.*
- *Spatial characteristic designates information with a location perspective, e.g., message reached end-point.*
- *State characteristic designates information with a state perspective, e.g., low average response time.*
- *Semantic characteristic designates information about a specific object or entity, e.g., persons that belong to the same role or messages that belongs to same conversation.*

To capture the local context of an event, variables are used, i.e., the context information is bound to variables which can be reused in the subsequent parts of a reaction rule, e.g., in the action part.

Definition 76 *(**Complex Events and Event Algebra**) A complex event type (i.e., the detection condition of a complex event) is built from occurred atomic or other complex event instances according to the operators of an event algebra. The included events are called components while the resulting complex event is the parent event. The first event instance contributing to the detection of a complex event is called initiator, where the last is the terminator; all others are called interiors.*

Definition 77 *(**Event Processing**) Event processing describes the process of selecting and composing complex events from raw events (event derivation), situation detection (detecting transitions in the universe that requires reaction either "reactive" or "proactive") and triggering actions as a consequence of the detected situation (complex event + conditional context).*

Examples for situations are: "*If more than three outages occur then alert*" or "*If a department D is retracted from the database than retract all associated employees E*". Events can be processed in real time without persistence (short-term) or processed in retrospect as a computation of persistent earlier events (long-term), but also in aggregated from, i.e., new (raw) events are directly added to the aggregation which is persistent. According to the ECA paradigm event processing can be conditional, i.e., certain conditions must hold before an action is triggered. Event processing can be done either actively (pull-model), i.e., based on actively monitoring the environment and, upon the detection of an event, trigger a reaction, or passively (push-model), i.e., the event occurrences are detected by an external component and pushed to the event system for further processing.

3.6 Requirements for a Logic Rule Based SLA Language

After having introduced relevant basic concepts and terminologies in the various domains of knowledge representation which will be applied in this dissertation, I will now elaborate on the top-level requirements for a declarative, logic-based rule language for representing SLAs. This section continues on a more detailed level the discussion of the general requirements for SLA representation and management, as presented in section 1.2.

Different kinds of rules and facts: A logic SLA language should allow to coherently represent derivation rules, reaction rules, integrity rules, deontic rules in a homogeneous syntax.

- *Derivation rules* are sentences of knowledge that are derived from other knowledge by an inference or mathematical calculation.
- *Reaction rules* are behavioral rules which react on occurred events or changed conditions by executing actions.

- *Integrity rules* (or constraints) are assertions which express conditions that must be always satisfied.
- *Deontic rules* describe rights and obligations of roles in the context of evolving states (situations triggered by events/actions) and state transitions.
- *Facts* might describe various kinds of information such as events (event/action messages, event occurences), (object-oriented) object instances, class individuals (of ontology classes), norms, constraints, states (fluents), conditions of the various forms, actions, data (e.g., relational, XML) etc, which might be qualified, e.g., by priorities, temporally etc.

Support interoperation with (webized) descriptive specifications: A SLA language should be able to refer to external *Semantic Web ontologies* by means of URIs in order to use the taxonomical vocabularies as type systems and the individual models as external constants/objects. Domain-independent SLA rules can be given a domain-dependent meaning (with a precise semantics) and accordingly rules can be much easier interchanged and managed/maintained in a distributed environment. The core SLA rule language stays compact and can be easily extended with different domain-specific vocabularies on a "per-need-basis".

Support for practical procedural language constructs such as *constructive queries over external data sources*, *expressive procedural attachments* and *external type systems* which allow calling external functionalities and using external objects and data during rule execution. Many SLA rules refer to or describe measurement functions and SLA metrics over data stored in some kind of external database which can be anything from log files to web sources or relational databases and data warehouses. The rule language must allow the direct integration of these secondary data storages as facts into the rules in order to reduce redundancy and high memory consumption. It should also support outsourcing of expensive (pre-)processing of data to external system, e.g., constructing SQL aggregation queries (views) on the database server, and communication of data such as event messages, queries to (web) service to respective middleware products such as an ESB. Procedural attachments are procedure calls on external computational models of a standard programming language, e.g., directly on Java methods. Therefore, procedural attachments are a crucial extension of the pure logic inferences used in logic programming. They allow a combination of the benefits of declarative (rule-based) and procedural languages, e.g., to delegate computation-intensive tasks to optimized Java code or to invoke procedure calls on Java methods which can not be expressed in pure logic. In a rule-based SLA language they are also vital to integrate existing tools such as system- and network management tools, web services or data ware houses. Procedural attachments should be supplemented with a typed logic with external type systems such as Java or Semantic Web ontologies in order to support SE principles such as data abstraction or modularization in declarative SLA programming and capture the rule engineer's intended meaning of a SLA.

Provide efficient operational semantics and declarative semantics: A declarative reading of SLA logic language with a well-defined and *well-understood declarative semantics* for LPs with negations, such as well-founded semantics, is needed to ensure correctness of the evaluations of the truth-valued formulas. The operational semantics of a SLA rule language should support *automated rule chaining based on backward reasoning* in order to cope with masses of frequently changing data and enable reuse of rule definitions in different contexts. *External functionalities* such as procedural code, description logic reasoning or data queries should be integrated into the semantics in a hybrid way in order to exploit highly efficient and optimized external reasoners. *Linearity, termination properties and polynomial (data) complexity* are important to program restricted parts of the SLA logic in a more procedural and optimized way, ensure linear execution for e.g., reactive rules with a (simulated) forward-directed operational semantics and handle large temporarily populated fact bases. A declarative model-theoretic semantics in addition to the forward-directed operational semantics of reaction rules should be imposed on the *query-based processing of reaction rules* such as ECA rules which fire truth-valued event and condition queries and trigger boolean-valued procedural actions. The operational semantics for reaction rules should support *complex event and action processing* in a tight combination with the other rule types enabling situated reactions.

Support for 3-valued logic with non-monotonic default and monotonic explicit negation and open and closed world reasoning (resp. scoped reasoning with scoped negation) with reasonable structural properties: Like in natural language (i.e., SLAs defined in natural language) a logic-based SLA language needs *non-monotonic default negation* to express closed world assumptions and default (assumed) negation and *monotonic explicit negation* (explicit falsity) in order to distinguish complete (e.g., from closed database sources) from incomplete knowledge (e.g., from open possible unavailable web sources or systems), handle inconsistent theories and express default rules and strict rules. See [BG94] for examples that need explicit negation in logic programming. In the SLA domain where contracts are concluded and managed in an open distributed environment such as the Semantic Web an assumed *undefined truth value* is important since reasoning can be time-consuming and error-prone, e.g., due to long-running response times of (Internet) queries or communication failures. The inference process can proceed with an assumed undefined truth value and later be updated to true or false, when more data is available. The semantics should satisfy *reasonable structural properties* as defined by [KLM90, Dix92] such as *cumulative monotony* (cautious monotony and cut), which refers to the usability of intermediate lemmas, and *rationality*, which (in a sceptical semantics) is a stronger form of cautious monotony and refers to the ability to add the negation of non-provable conclusions. To preserve these properties for extended LPs with non-monotonic default negation in open distributed knowledge bases, a kind of *scoped reasoning*, which partially closes off parts of the open distributed knowledge base, is needed, e.g., to ensure that answers to queries in the context of default negation are not forced to be withdrawn when new knowledge is added.

Allow facts and rules to be bundled to possible distributed modules and qualified by meta-data labels and priorities: IT service contracts include rules on different contractual levels such as general conditions, agreements on service properties, agreements on service usage, operational/measurement rules on IT infrastructure level, (business) policies etc. Moreover, the agreements, policies and rule sets are typically managed in a distributed way and are scattered over domain boundaries. To support such a *distributed knowledge base* rules should be bundled to rule sets, so called *modules*, which have their own unique identifiers and which might be written and managed as stand alone script files provided on the Web using URIs and which might be imported by other modules leading to a hierarchy of nested modules (rule/fact sets). Rules (as well as modules) should be *labelled by additional meta data* such as rule names/identifiers, authoring information such as Dublin Core meta data, temporal qualifications, e.g., validity times and *priority values*, defining priorities between rules and complete modules in order to handle arbitrary (user-defined) conflicts between competing rules and define alternatives on individual and/or group level and exceptional rules and rule sets.

Support for integrity rules to define arbitrary conflicts and support for priority definitions between rules and modules to address consistency and coherence according to user preferences: In SLA rules conflicts might occur not just between simultaneous positive and negative conclusions, but also between arbitrary user-defined and domain-specific conflicting conclusions, e.g., a simultaneously concluded discount of 5% and 10% might be a conflict in certain situations. Expressive *integrity constraints* (integrity rules) are an appropriate way to represent all sorts of domain-specific conflicts. (User-defined) preferences written as priority relations between rules or rule sets (modules) should be used to address consistency and coherence wrt to the violated integrity constraints, i.e., the semantics should decide whether and how it is possible to (defeasibly) derive the expected conclusions wrt to the defined integrity constraints and preference definitions.

Support for transactional dynamic updates and evolvability of the knowledge base: The dynamic character of the SLA domain where contracts and in particular SLA rules need to be adapted to changing requirements amounts for declarative mechanisms to maintain and evolve the SLA specifications, i.e., to *dynamically update the intensional KB (rules) and extensional KB (facts)*, add and remove rules and complete modules and transit the knowledge state to an evolved knowledge state leading to a sequence of *state transitions* which might be possibly *rolled-back in a transactional style* if (integrity) constraints are missed.

Support for verification, validation and integrity testing: Measurement of the quality, anomaly freeness and well-formedness of the rule sets used to formalize SLAs is an important need for the acceptance by the contract partners, in particular when rules are subject to change and have different information sources. Basically this means using verification, validation and integrity preserving techniques and providing means to solve rule conflicts which occur

in different situations, in particular if modular revisioning/updating is allowed, i.e., new behavior can be specified by simply adding rules without the need to modify or delete previous rules. Verification and validation is also vital for collaborative engineering of larger rule sets (contracts) and interchanging rule bases in different execution environments (e.g., different inference services provided on the Semantic Web).

Support for normative reasoning on deontic rules: The main aim for concluding a contract is to arrange the normative relationships relating to permissions, obligations, prohibitions and other normative modalities between contract partners. These contract norms must be *personalized* based e.g., on role models defined in relational database schemas or Semantic Web taxonomies and must be *qualified in an extensible way*, e.g., temporal, i.e., a norm holds in a certain state which is initiated and terminated by certain events/actions. A SLA language should allow *deontic reasoning* with sophisticated temporal deontic rules.

Support for engineering and rendering: Real usage of a representation language which is usable by others than its inventors immediately makes rigorous demands on the syntax and the engineering support: comprehension, readability and usability of the language by users, compact representation, exchangeability with other formats, explanation of results, means for serialization of rules, tool support in writing and parsing rules etc. are vital for the acceptance and usability of the language. To address all these needs for different user roles (human users, automated agents and machine tools, execution environments) several syntactical layers of the language are required. In particular, a controlled natural language syntax for human consumption, a (Semantic) Web standard based language (e.g., XML based) enabling machine-interpretation, processing and translation into multiple execution syntaxes of standard rule engines. A visual layer might be useful, but is not a necessity. Using open Web based standards (e.g., RuleML) to develop a general and abstracted rule markup language facilitates these tasks.

This list of requirements describes only the top level of critical success factors and many other requirements can be derived from them following a critical success factor analysis methodology. All these requirements have to be satisfied and integrated into a single framework. It is crucial to find the right tradeoff between generality and expressiveness and strictly (classical) logical semantics and practical integration of non-classical inference features and practical rule concepts such as procedural attachments. In the following two chapters I will further expand on these requirements and introduce the ContractLog KR and the Rule Based Service Level Agreement (RBSLA) markup language as my solution to tackle these requirements in an adequate and coherent framework.

4 The ContractLog KR

ContractLog is an expressive and computational efficient KR framework consisting of selected, adequate logical knowledge representation concepts for the formalization and automated execution of electronic contracts such as SLAs or higher-level policies. It combines selected logical formalisms which are implemented on the basis of declarative logic programs and meta programming techniques. Meta-programming and meta interpreters have their roots in the original von Neumann computer architecture where program and data are treated in a uniform way. Meta interpreters are a popular technique in logic programming [BK82] for representing knowledge, in particular, knowledge in the domains containing logic programs as objects of discourse. Logic programs representing such knowledge are called meta-programs (aka meta interpreters) and their design is referred to as meta-programming. The ContractLog KR combines declarative programming of SLA rules as logic programs with technologies from the Semantic Web domain and object-oriented programming in Java. By expressive procedural attachments external Java functionalities, tools and data can be dynamically integrated into declarative rule executions by dynamically calling Java APIs and binding dynamically instantiated Java objects to logical variables at runtime. The methods and attributes of these bound objects can be used in the reasoning chains enabling, e.g., external data integration by JDBC or interface calls to existing system and network management tools or web services. A particular contribution of the ContractLog KR is its combination and implementation of adequate logics for the representation of electronic contracts. The combination of these logical formalisms qualify the ContractLog KR as an adequate, declarative tool for rule-based contract management and representation. It is worth noting that most of my implementations in the ContractLog KR are also applicable and useful in other domains than rule-based electronic contracts.

In the following section, I describe the core syntax and semantics of the ContractLog KR. I then elaborate on further advanced KR concepts and logical formalisms, which are additionally required to adequately formalize typical SLA and policy rules. These formalisms have been carefully selected wrt common adequacy criteria in KR in order to fulfil the requirements stated in section 3.6 and section 1.2. Table 4.1 gives an overview on the main logics implemented in the ContractLog KR and their contributions to SLA representation.

Table 4.1: Table (Overview ContractLog KR)

Logic	Formalism	Usage / Contribution
Extended Logic Programs (section 4.1)	Derivation Rules and extended LPs	Deductive reasoning with SLA rules with default and explicit negation.
Typed Logic (section 4.2)	Object-oriented Typed Logic and Procedural Attachments	Typed terms restrict the search space and enable object-oriented software engineering principles in declarative rule-based programming. Procedural attachments integrate object oriented programming into declarative rules.
Meta-Data Annotated Labelled Logic (section 4.3)	Meta-Data Labels such as (object) IDs or Dublin Core Annotations	Labelled rule sets (modules) can be used for scoped reasoning on explicitly closed parts of the knowledge in open environments such as the Semantic Web. The labels such as rule object ids or module ids form the basis for a efficient and distributed management of rules enabling dynamic updates and (defeasible priority reasoning with rule preferences).
Description Logics (section 4.2.2)	Semantic Web Ontology Languages such as RDFS or OWL	Semantic Web ontologies are used to define external hybrid DL-Types and semantic domain descriptions (e.g. contract vocabularies) in order to give domain-independently described rules a domain-dependent meaning. Facilitates rule interchange and enhances the expressiveness of the SLA representation language in a flexible way, i.e., different ontologies/vocabularies can be integrated.
(Re)active Logic (section 4.7)	Extended Event-Condition-Action Rules (ECA) with ID-based Updates, Rollbacks, Complex Events/Actions, Active Rules	Supports active event detection/event processing and event-triggered reactions by reaction rules.
Temporal Event/Action Logic (section 4.6)	Event Calculus	Temporal reasoning about dynamic systems, e.g., interval-based complex event definitions (event algebra) or effects of events on the contract state; contract state tracking; reasoning about events/actions and their effects
Deontic Logic (section 4.8)	Deontic Logic with norm violations and exceptions	Rights and obligations formalized as deontic contract norms with norm violations (contrary-to-duty obligations) and exceptions (conditional. defeasible obligations) stated as temporal normative deontic rules.
Integrity Preserving, Preferenced, Defeasible Logic (section 4.4)	Defeasible Logic and Integrity Constraints	Used to express default rules and priority relations of rules. Facilitates conflict detection and resolution as well as revision/updating and modularity of rules with preferences or rules and modules in terms of defeasible ordered LPs (OLPs with priorities between rules and rule sets (modules)).
Test Logic (section 4.9)	Declarative Test-driven Verification and Validation of Rule Bases	Validation and Verification of SLA specifications against predefined SLA requirements in terms of rule-based test cases and test suits. Helps to safeguard the engineering, dynamic update and interchange process of SLAs.

4.1 Core Syntax and Semantic of the ContractLog KR

The expressive power of definite Horn programs is too limited for adequately representing complex service contracts in open distributed environments (see section 3.6 for a discussion of the requirements). Accordingly, a ContractLog LP is an extended LP (ELP) [LW92], i.e., a LP with monotonic explicit negation and non-monotonic default negation. An ELP is a set of clauses of the from $H \leftarrow B$, where H is a literal over L called the head of the rule, and B is a set of literals over L called the body of the rule. A literal $B_i \in B$, which might be default negated ($\sim B_i$), is either an atom or the negation $\neg$ of an atom, where $\sim$ denotes default negation written as $not(...)$ and $\neg$ is denoted as explicit negation written as $neg(...)$. Roughly, default negation means, everything that can not be proven as true is assumed to be false. A rule is called a fact if it only consists of the rule head H. An atom is a n-ary formula containing terms $p(a, X, f(...))$, where p is the predicate name. A term is either a constant a, a variable X or a n-ary complex term/function $f(...)$. A goal/query G is a headless clause defining a conjunction of literals (positive or negative atoms) $L_1 \wedge .. \wedge L_i$ where each L_i is called a subgoal. A *query* is embedded in the built-in function $:-solve(...)$ or $:-eval(...)$.

4.1.1 Syntax of ContractLog

Syntactically, the ContractLog KR uses an extended ISO Prolog related scripting syntax (ISO Prolog ISO/IEC 13211-1:1995) called Prova (`http://www.prova.ws/` [KPS06]) to write ContractLog programs as stand-alone scripts. I give the syntax in EBNF notation, i.e., alternatives are separated by vertical bars (|); zero to one occurrences are written in square brackets ([]) and zero to many occurrences in braces ({}). The EBNF syntax already comprises meta data labels, defeasible rules, typed terms, java-based procedural attachments, arithmetic relations and negations. I will describe these expressive extensions provided by the ContractLog KR in the subsequent sections of this chapter:

```
prova ::= {statements}, end of file;
meta data ::= "properties", "(", {property value pair}, ")";
property value pair ::= lowercase word, "(", (constant | individual), ")";
statements ::= ([meta data, "::"], statement), {statements};
statement ::= (fact | rule | query), end of statement;
fact ::= (literal | defeasible literal);
rule ::= (literal | defeasible literal), ":-", body literals;
query ::= ":-", ("eval" | "solve"), "(", (literal | naf literal | defeasible literal), ")";
literal ::= atom | neg atom;
naf literal ::= "not", "(", (literal | defeasible literal), ")";
defeasible literal ::= "defeasible", "(", literal, ")";
body literals ::= (literal | defeasible literal | naf literal | arithmetic relation |
                  java call | cut), {",", body literals};
atom ::= relation;
neg atom ::= "neg", "(", relation, ")";
relation ::= predicate symbol, "(", terms, {"|", argument tail}, ")";
argument tail ::= variable;
predicate symbol ::= lowercase word | uppercase word;
java call ::= functional java call | predicate java call | constructor java call;
```

```
functional java call ::= left term, "=", predicate java call;
predicate java call ::= static java call | instance java call;
static java call ::= qualified java class,".",method name,"(", terms,")";
instance java call ::= variable, ".", method name, "(", terms, ")";
constructor java call ::= left term,"=",qualified java class,"(",terms,")";
terms ::= {term, {",", terms}};
term := left term | (func, "(", terms, ")");
left term ::= variable | constant | individual | prova list;
func ::= variable | constant;
variable ::= uppercase word | java typed variable | dl typed variable;
constant ::= lowercase word | ('"', string, '"') ;
individual ::= namespace prefix, "_", dl class, ":", dl instance;
java typed variable ::= qualified java class, ".", uppercase word;
dl typed variable ::= uppercase word, ":", dl type;
dl type ::= namespace prefix, "_", dl class;
namespace prefix ::= lowercase word;
dl class ::= word;
dl instance ::= word;
prova list ::= "[]" | ("[", head, {"|", tail}, "]");
arithmetic relation ::= left term, binary operator, term;
binary operator ::= "=" | "<>" | ">" | "<" | ">=" | "<=";
head ::= term;
tail ::= variable;
uppercase word ::= ["A"-"Z","_"], {lowercase word};
lowercase word ::= ["a"-"z","_"], {word};
word ::= ["a"-"Z",0-9]+;
cut ::= "!";
end of statement ::= "." newline;
```

4.1.2 Declarative Semantics of ContractLog

ContractLog is intended to be a general KR framework which is applicable to various rule languages and declarative LP semantics such as well-founded semantics (WFS), stable model semantics (STABLE) or completion semantics (COMP) (with its procedural counterpart SLDNF resolution which suffers from the well-known drawbacks of possible loops, floundering and allowendness restriction for variables in negation-as-finite-failure tests). The ContractLog KR to an large extend avoids a strong commitment to one particular LP semantics used to interpret the formalisms of the KR and allows for direct implementations respectively transformations on top of existing rule engines. However, in order to fully exploit the expressiveness of the ContractLog KR the model-theoretic semantics $SEM(P)$ of a ContractLog LP P should be a subset of the 3-valued Herbrand-models with an unknown truth value and two forms of negation (see requirements in section 3.6). I adopt a sceptical viewpoint [GLV91] where the set of all atoms or default atoms of a program P are true in all canonical models $SEM^{scept}(P)$, in contrast to a credulous entailment relation, where the set of all atoms or default atoms are true in at least one model of $SEM^{cred}(P)$. Since negation is used the existence of a least model as in definite LPs is not longer guaranteed and it might be that the LP has several minimal models. Several semantics for assigning models to LPs with negation have been proposed (see section 3.3.2.4). The two broadly accepted semantics for normal LPs are the well-founded semantics and stable model semantics respectively extended WFS and answer set semantics (ASS) for extended LPs. The semantics of choice for ContractLog programs is WFS for the following reasons:

- WFS is generally accepted as a robust and natural semantics for logic programs
- it assigns a unique (partial) model to every program (with unrestricted negation-as-failure)
- it has polynomial worst-case data complexity for computing the well-founded model of a given intensional knowledge base, i.e., a set of rules is polynomial in the size of the extensional knowledge base (the facts); in contrast to STABLE which is NP-complete even for propositional normal LPs [Mar91]
- it has an undefined truth-value
- it satisfies reasonable structural properties such as rationality and cumulative monotony (cautious monotony and cut) [Dix95b]
- it coincides with two-valued STABLE for definite, acyclic and stratified LPs and coincides with the least three-valued STABLE for normal LPs [Prz90b]

By definition SEM_{WFS} only contains a single model for a particular knowledge state of a program P: $SEM_{WFS}(P) = M$. The well-founded semantics (WFS) of Van Gelder et al. [VGRS91] is a 3-valued semantics with three possible truth values: true (t), false (f) and unknown (u). In short, WFS assigns value "unknown" to an atom A, if it is defined by unrestricted negation. I refer to section 3.3.2.4 and [VGRS91, Prz89a, Prz90b] for a definition of WFS for normal LPs. Several authors extended WFS for normal LPs to extended WFS for LPs with two negations and a sceptical entailment relation that refrains from drawing conclusions, whenever there is a potential conflict [BG94, Lif96, PA92, Bre96]. However, a consensus in logic programming research which one is the most adequate extension of WFS for extended LPs is still missing. I now define a defeasible well-founded semantics for extended logic programs, called $WFSX_{DefL}$, as semantics of choice for ContractLog LPs.

The extended defeasible well-founded semantics $WFSX_{DefL}$ of ContractLog extends the well-founded semantics for extended logic programs (WFSX) [PA92] with integrity constraints and defeasible reasoning to handle arbitrary conflicts in programs, e.g., inconsistency between a simultaneously concluded literal and its negation. Basically, WFSX follows from WFS for normal LPs as defined by Van Gelder et al. [VGRS91] plus a coherence principle which implements the intuition that explicit negation implies default negation: $\neg L \Rightarrow \sim L$. Note that coherence in WFSX has a different meaning than in non-monotonic defeasible logic (see section 4.4.3), where it states that no literal should be simultaneously provable and unprovable. I first recall the definition of WFSX from [ADP95].

Definition 78 ***(Extended Herbrand Base)*** *The extended Herbrand base $\overline{B}(P)$ of an extended LP P is the set of all objective literals of P. An objective literal L is either an atom A or its explicit negation $\neg A$. $\neg L$ is the complement of the*

literal L such that $\neg\neg A = A$. $\sim L$ is called a default literal, where $\sim$ denotes default negation.

Definition 79 ***((Pseudo-)Interpretation)*** *A pseudo-interpretation of a program P is a set $\overline{T} \cup \sim \overline{F}$ where $\overline{T}$ and $\overline{F}$ are subsets of $B(P)$. An interpretation I is a pseudo-interpretation where the sets $\overline{T}$ and $\overline{F}$ are disjoint. Objective literals in $\overline{T}$ are said to be true in I, objective literals in $\overline{F}$ are said to be false by default in I, and in $B(P) - I$ undefined in I. If $\overline{T} \cup \overline{F} = B(P)$ the interpretation is two-valued otherwise it is three-valued (as in WFS).*

Definition 80 ***(Coherence Operator)*** *Let $I = \overline{T} \cup \sim \overline{F}$ be a set of literals such that $\overline{T}$ does not contain any pair $A, \neg A$. The coherence operator coh is defined as:*

$$coh(I) = I \cup \sim \{\neg L | L \in \overline{T}\}$$

Definition 81 ***(Seminormal version of a program)*** *The seminormal version of a program P is the program P, obtained from P by replacing every rule $H \leftarrow B$ in P by the rule $H \leftarrow B, \sim \neg H$, where H is the head literal and B is the body (set of literals) of the rule.*

In the following $\Gamma(\overline{E})$, where $\overline{E}$ is a set of expressions, is used to denote $\Gamma_P(\overline{E})$, and $\Gamma_{\overline{E}}(\overline{E})$ to denote $T_{P_{\overline{E}}}(\overline{E})$. See [GL90] for a definition of the operator Γ.

Theorem 4 ***(Well-founded Model)*** *Let P be a program whose least fixpoint of $\Gamma\Gamma_{\overline{E}}$ is $\overline{T}$. The well-founded model W_P^* of P is the pseudo-interpretation $I = \overline{T} \cup \sim (B(P) - \Gamma_{\overline{E}}\overline{T})$. If W_P^* is an interpretation, then P is called non-contradictively, and W_P^* is the well-founded model of P. W_P^* is iteratively defined by the following sequence I_α, where δ is a limit ordinal:*
$I_0 = \emptyset$
$I_{\alpha+1} = \Gamma\Gamma_{\overline{E}} I_\alpha$ *for successor ordinal* $\alpha + 1$
$I_\delta = \bigcup_{\alpha<\delta} I_\alpha$ *for limit ordinal* δ
There exists a smallest ordinal δ_0 such that I_{δ_0} is the least fixpoint of $\Gamma\Gamma_{\overline{E}}$, and $W_P^* = I_{\delta_0} \cup \sim (B(P) - \Gamma_{\overline{E}} I_{\delta_0})$.

Definition 82 ***(WFSX semantics)*** *The Well-founded semantics for extended LPs (WFSX) assigns to every non-contradicting extended LP P the well-founded model W_P^* of P:*

$$SEM_{WFSX}(P) := \{W_P^*\}.$$

The following theorem which has been proven in [Alf93] states that WFSX is sound wrt to the answer-set semantics (ASS) of [GL90].

Theorem 5 *(**WFSX is sound wrt to ASS**) Let P be an non-contradicting extended LP with at least one answer-set, i.e., it has partial stable models. Then for any objective literal L, if $L \in W_P^*$ then L belongs to all answer-sets of P; if $\sim L \in W_P^*$ then L does not belong to any answer-set of P.*

Since in ContractLog explicit negation is allowed in the head of rules conflicts between simultaneous positive and negative conclusions might occur. Hence, an important question is whether and how it should be possible to derive conflicts, i.e., how consistency and coherence (here in terms of provable/unprovable literals) are addressed by the semantics. WFSX [Alf93] defines a program transformation to the seminormal version. Using a monotone operator the seminormal version of a program in WFSX guarantees that a literal L is not considered a potential conclusion whenever the complementary (conflicting) literal is already known to be true. However, in the contract or legal reasoning domain conflicts might occur not just between positive and negative conclusions, but also between arbitrary literals and derived answers (see section 3.6 and section 4.4). For instance, a conflict might occur if two different penalty levels or discount level can be concluded at the same time. Hence, more expressive mechanisms to represent and solve arbitrary conflicts are needed than simply building the seminormal version of a program and proving the default falsity of the complementary head literal of each rule. To provide these levels of expressiveness in ContractLog I introduce a defeasible logic variant in terms of a meta programming approach for defeasible reasoning on top of extended LPs. The defeasible logic extension enables user-defined integrity rules to describe arbitrary conflicts and defeasibly solves conflicts wrt to defined integrity constraints by superiority definitions between rules and complete rule sets (aka modules), i.e., it preserves coherence and consistency for extended LPs wrt integrity constraints by defeasible reasoning. The main advantages of using defeasible logic, as opposed to other non-monotonic approaches such as default logic or circumscription are:

- defeasible logic is computationally efficient, i.e., the computation of defeasible conclusions is polynomial, and highly efficient implementations exist [ABMR00]
- defeasible logic is directly sceptical and has built-in preference handling facilities
- it can be implemented as a meta program in logic programming providing more flexibility to decompose defeasible theories into different rule languages, use different rule engines and implement optimized variants with e.g., dynamic (rule-based) preference adaption and explicit negation-as-failure of defeasible literals in rule bodies
- defeasible logic has a proof-theory and a model-theoretic semantics [Mah02]

The detailed description and discussion of the integrity based defeasible logic approach of ContractLog will follow in section 4.4. Here I will introduce the basic concepts and the core defeasible semantics $WFSX_{DefL}$ of ContractLog's

extended well-founded defeasible logic which follows from the well-founded semantics of Van Gelder, Ross and Schlipf [VGRS91], its extension in WFSX [PA92] and the WFS semantics decomposition of defeasible logic of Maher et al. [MG99].

Definition 83 *(**Defeasible Theory with Integrity Constraints**) A defeasible theory Φ consists of a set of facts $\overline{Fa}$, a set of rules $\overline{R}$, a set of integrity constraints $\overline{IC}$ and a superiority relation $>$ on $\overline{R}$.*

Definition 84 *(**Mutual Exclusive Complements**) Let Φ be a defeasible theory. Let $\overline{L}$ be a set of mutual exclusive literals defined in an xor integrity constraint IC_{Xor}, i.e., the literals are not allowed to be simultaneously concluded from Φ. Then $-L$ denotes a complement of the literal $L \in \overline{L}$ if $-L \in \overline{L} \setminus \{L\}$.*

For instance, if $\overline{L} = \{A, \neg A\}$, i.e., the atoms A and $\neg A$ are defined as mutual exclusive then A is a complement of $\neg A$ and vice versa. The defeasible meta program of ContractLog defines a general consistency which states that positive and negative conclusions are mutual exclusive, i.e., $IC_{Xor} = \{L, \neg L\}$:

```
% positive and negative conclusions are mutual exclusive
integrity(xor([P|Args],neg([P|Args]))).
```

That is, a literal L and its negation $\neg L$ are mutual exclusive.

Definition 85 *(**Conclusions**) A conclusion of a defeasible theory Φ is a tagged literal and can have one of the following forms:*

- *$+\triangle L$, i.e., it is proved that L is strictly provable in Φ, i.e., using only facts and strict rules.*
- *$-\triangle L$, i.e., it is proved that L is not strictly provable in Φ.*
- *$+\partial L$, i.e., it is proved that L is defeasibly provable in Φ.*
- *$-\partial L$, i.e., it is proved that L is not defeasibly provable in Φ.*

Defeaters are omitted in ContractLog. The defeasible logic implemented in ContractLog uses expressive integrity constraints to define (arbitrary) conflicts and uses superiority relations between rules and rule sets to solve conflicts. Strictly provability involves only strict rules and facts, whereas defeasible provability involves defeasible and possibly strict knowledge.

```
% strict knowledge is also defeasible derivable
defeasible([P|Args]) :- derive([P|Args]).
```

The tagged conclusions are meta-theoretical statements about provability. They do not appear directly in the defeasible theory, i.e., the tagged literals can not be used in rules. Hence, there is no way to directly express in defeasible logic that a literal should fail to be proved. But, since in ContractLog the inference rules of the defeasible logic are represented as a defeasible meta program and hence exploit the logic programming semantics for LPs with negation, it is possible to state explicitly in terms of default negation that the prove of a literal should fail. I introduce the operator $\sim$ into the defeasible logic language to state that it should be proved that a (body) literal can not be proved. That is, a goal $\sim L$ in an extended defeasible theory Φ is strictly resp. defeasibly provable, if it is proved that the literal L can not be proved strictly resp. defeasibly.

Definition 86 ***(Extended Defeasible Logic)***

- $\leftarrow$ *L: it should be proved that L can be strictly proved, i.e.,* $\Phi \vdash +\triangle L$
- $\leftarrow\sim$ *L: it should be proved that L can not be strictly proved, i.e.,* $\Phi \vdash -\triangle L$
- $\Leftarrow$ *L: it should be proved that L can be defeasibly proved, i.e.,* $\Phi \vdash +\partial L$
- $\Leftarrow\sim$ *L: it should be proved that L can not be defeasibly proved, i.e.,* $\Phi \vdash -\partial L$

Note, the difference between $\sim$ and $\neg$. For instance, a goal $\leftarrow \neg A$ means that it should be proved that the negated atom $\neg A$ can be strictly proved, i.e., $\Phi \vdash +\triangle \neg A$ and a goal $\Leftarrow \neg A$, means that it should be proved that $\neg A$ can be defeasibly proved, i.e., $\Phi \vdash +\partial \neg A$. In contrast, a goal $\leftarrow\sim A$ means that it should be proved that A can be strictly proved, i.e., $\Phi \vdash +\triangle A$ and a goal $\Leftarrow\sim A$ means that it should be proved that A can not be defeasibly proved, i.e., $\Phi \vdash -\partial A$. The implementation is given by the following mapping from the goals of an extended defeasible theory Φ into the goals of the meta program representation $P(\Phi)$:

- $\leftarrow A$ maps to a goal A
- $\leftarrow \neg A$ maps to a goal $neg(A)$
- $\leftarrow\sim A$ maps to a goal $not(A)$
- $\Leftarrow A$ maps to a goal $defeasible(A)$
- $\Leftarrow\sim A$ maps to a goal $not(defeasible(A)$
- $\Leftarrow \neg A$ maps to a goal $defeasible(neg(A))$

The following definition of the well-founded defeasible logic goes back to the semantics decomposition of defeasible logic of Maher et al. [MG99].

Definition 87 ***(Extension)*** *An extension Ex is a 4-tuples of sets of tagged literals:* $\langle +\triangle, -\triangle, +\partial, -\partial \rangle$.

Definition 88 *(**Unfounded Set** A set $\overline{L}$ of literals is strictly unfounded ($\triangle$-unfounded) wrt an extension Ex and strict inference ($\triangle$) if for every literal L in $\overline{L}$, and for every strict rule $B \rightarrow L$ either holds:*

- $B \cap -\triangle_{Ex} \neq \emptyset$, *or*
- $B \cap \overline{L} \neq \emptyset$

A set $\overline{L}$ of literals is defeasibly unfounded (∂-unfounded) wrt an extension Ex and defeasible inference (∂) if for every literal $L \in \overline{L}$, and for every rule $r1 : B(r1) \hookrightarrow L$ in Φ, where $r1$ is the rule name, $B(r1)$ the body of the rule $r1$ and $\hookrightarrow$ is unspecified (i.e., might be strict ($\leftarrow$) or defeasible ($\Leftarrow$)) it either holds that:

- $B(r1) \cap -\partial_{Ex} \neq \emptyset$, *or*
- $B(r1) \cap \overline{L} \neq \emptyset$, *or*
- *there is a rule $r2 : B(r2) \hookrightarrow \neg L$ in Φ such that $B(r2) \subseteq +\partial Ex$ and for every rule $r3 : B(r3) \hookrightarrow L$ in Φ either $B(r3) \cap -\partial_{Ex} \neq \emptyset$ or $r3 \not> r2$.*

The union of the $\triangle$-unfounded and ∂-unfounded sets is closed.

Definition 89 *(**Greatest Unfounded Set**) Let Φ be a defeasible theory. The greatest $\triangle$-unfounded set of Φ wrt an extension Ex, denoted by $U^{\triangle}_{\Phi}(Ex)$, is the union of all $\triangle$-unfounded sets of Φ wrt Ex and the greatest ∂-unfounded set of Φ wrt Ex, denoted by $U^{\partial}_{\Phi}(Ex)$, is the union of all ∂-unfounded sets of Φ.*

Definition 90 *(**Consequence Operators**) Let $\overline{R}$ be the set of rules in Φ (which is mapped to $P(\Phi)$), $\overline{R}_s$ be the set of all strict rules in $\overline{R}$, $\overline{R}_d$ be the set of defeasible rules in $\overline{R}$ and $\overline{R}_{sd}$ be the set of strict and defeasible rules in $\overline{R}$. $\overline{R}[L]$ denotes the set of rules in $\overline{R}$ with head L. The following monotonic transformation operators are defined wrt I:*

- $T_{\Phi}(I) = (+\triangle', -\triangle', +\partial', -\partial')$ *where*
  $+\triangle' = \overline{Fa} \cup \{L | \exists r \in \overline{R}_s[L] B(r) \subseteq +\triangle\}$
  $-\triangle' = -\triangle \cup (\{L | \forall r \in \overline{R}_s[L] B(r) \cap -\triangle \neq \emptyset\} - \overline{Fa})$
  $+\partial' = +\triangle \cup \{L | \exists r \in \overline{R}_{sd}[L] B(r) \subseteq +\partial, -L \in -\triangle$ *and* $\forall s \in \overline{R}[-L]$ *either* $B(s) \cap -\partial \neq \emptyset$, *or* $\exists t \in \overline{R}[L]$ *such that* $B(t) \subseteq +\partial$ *and* $t > s\}$
  $-\partial' = \{L \in -\triangle | \forall r \in \overline{R}_{sd}[L] B(r) \cap -\partial \neq \emptyset$ *or* $-L \in +\triangle$ *or* $\exists s \in \overline{R}[-L]$ *such that* $B(s) \subseteq +\partial$ *and* $\forall t \in \overline{R}[L]$ *either* $B(t) \cap -\partial \neq \emptyset$ *or* $t \not> s\}$
- $U_{\Phi}(I) = (\emptyset, U^{\triangle}_{\Phi}(I), \emptyset, U^{\partial}_{\Phi}(I)))$
- $W_{\Phi}(I) = T_{\Phi}(I) \cup \neg U_{\Phi}(I)$ *where* $\neg U_{\Phi}(I)$ *is obtained from* $U_{\Phi}(I)$ *by taking the complement of each atom in* $U_{\Phi}(I)$

The bottom-up definition of $T_{\Phi}(I)$ allows to transfer the declarative computational model of defeasible logic to the bottom-up characterizations of the WFS.

For more details on the bottom-up definition see [AM02]. Note that, due to the general consistency constraint in ContractLog, as defined above, the complement $-L$ by default consists of its negation $\neg L$. However, further arbitrary and mutual exclusive literals might be defined in terms of integrity rules as complements of L. The approach, due to the meta program representation, is not restricted to propositional theories, but allows defining complements in a wider sense, e.g., $IC_{Xor} = discount(Customer, 5), discount(Customer, 10)$, i.e., both atoms are mutual exclusive and complements to each other. A more detailed discussion of the preferenced defeasible integrity logic will follow in section 4.4.3.

Lemma 2 *$T_\Phi(Ex)$, $U_\Phi(Ex)$ and $W_\Phi(Ex)$ are monotonic.*

Theorem 6 *Let I_α be the elements of the increasing Kleene sequence starting from $\perp = (\emptyset, \emptyset, \emptyset, \emptyset)$ which has a limit $WFSX_{DefL} = (+\triangle_{WF}, -\triangle_{WF}, +\partial_{WF}, -\partial_{WF})$. Then $WFSX_{DefL}$ defines the conclusions of the Well-Founded Defeasible Logic theory Φ as follows:*

- $\Phi \vdash_{WFSX_{DefL}} +\triangle L$ *iff* $L \in +\triangle_{WFSX_{DefL}}$
- $\Phi \vdash_{WFSX_{DefL}} -\triangle L$ *iff* $L \in -\triangle_{WFSX_{DefL}}$
- $\Phi \vdash_{WFSX_{DefL}} +\partial L$ *iff* $L \in +\partial_{WFSX_{DefL}}$
- $\Phi \vdash_{WFSX_{DefL}} -\partial L$ *iff* $L \in -\partial_{WFSX_{DefL}}$

The proof can be given by induction on the length of derivation in the one direction and the number of iterations of the consequence operator in the other. A more detailed definition of the proof-theory and the implementation of the inference rules as a meta program is given in section 4.4.

I adopt the coherence principle of WFSX into the $WFSX_{DefL}$ semantics for well-founded defeasible theories.

Definition 91 *(Coherence principle)*

- $\triangle \neg L \rightarrow -\triangle L$
- $\partial \neg L \Rightarrow -\partial L$

The coherence principle for strict reasoning is inherited from the WFSX semantics. The coherence principle for defeasible reasoning is implemented by the following inference rule in the meta program $P(\Phi)$:

```
% defeasible(neg(...)) implies not(defeasible(...))
not(defeasible([P|Args])) :- defeasible(neg([P|Args])).
```

Note, the difference between the coherence principle adopted from WFSX and the coherency definition of a defeasible logic [Bil90], which states that no literal is simultaneously provable and unprovable, i.e., an extension $\langle +\triangle, -\triangle, +\partial, -\partial \rangle$ is coherent if $+\triangle \cap -\triangle = \emptyset$ and $+\partial \cap -\partial = \emptyset$. In order to ensure coherency for the extended well-founded defeasible logic its extension must be consistent, i.e., $L \in +\partial$ then $-L \in -\partial$. As defined above (in terms of an integrity constraint in the meta program) the default complement of a positive literal is its negation. Hence, to guarantee consistency I implement a general priority relation between positive/negative conflicting defeasible rules, such that the rule with the negative conclusion is preferred (overrides the "positive" rule if both are concluded simultaneously).

```
% negative information by default overrides positive information
overrides(neg([P|Args]), [P|Args]).
```

Proposition 1 *Extended Well-founded Defeasible Logic is coherent and consistent in the defeasible case.*

Proof 1 *If $+\partial L$ and $+\partial -L$ then $+\partial -L$ overrides $+\partial L$ (consistency). If $-L = \neg L$ then $+\partial \neg L$ implies $-\partial$ (coherence principle) and $-\partial \cap +\partial L = \emptyset$ (coherence).*

Note, that further priority rules are needed to preserve consistency wrt to other, arbitrary integrity constraints.

The following theorem shows completeness of the well-founded defeasible logic wrt to the $WFSX_{DefL}$ interpretation of its meta program $P(\Phi)$.

Theorem 7

- $\Phi \vdash_{WFSX_{DefL}} +\triangle A$ iff $P(\Phi) \models_{WFSX} A$
- $\Phi \vdash_{WFSX_{DefL}} +\triangle \neg A$ iff $P(\Phi) \models_{WFSX} neg(A)$
- $\Phi \vdash_{WFSX_{DefL}} -\triangle A$ iff $P(\Phi) \models_{WFSX} not(A)$
- $\Phi \vdash_{WFSX_{DefL}} +\partial A$ iff $P(\Phi) \models_{WFSX} defeasible(A)$
- $\Phi \vdash_{WFSX_{DefL}} +\partial \neg A$ iff $P(\Phi) \models_{WFSX} defeasible(neg(A))$
- $\Phi \vdash_{WFSX_{DefL}} -\partial A$ iff $P(\Phi) \models_{WFSX} not(defeasible(A))$

The proof can be given by induction on the length of derivations P in Φ. A defeasible theory Φ without defeasible rules coincides with an extended LP, i.e., the meta program $P(\Phi)$ conforms to the conventional extended LP P since I do not introduce any new meta predicates for strict knowledge, i.e., a predicate is defined in the predicate symbols of the signature without the use of any meta functions. The coherence principle is adopted from WFSX. Accordingly,

well-founded defeasible logic in the strict case is complete wrt the WFSX interpretation of its meta program. In the defeasible case the extended well-founded defeasible logic in ContractLog solves conflicts between positive and negative information (as well as other user defined integrity-based conflicts) by the use of the defeasible inference rules and the priority relations defined between knowledge.

To illustrate the correspondence and difference between WFSX semantics and $WFSX_{DefL}$ consider the following propositional example from [BG94]:

Example 2

$a \leftarrow\sim b$
$b \leftarrow\sim a$
$\neg a$

In WFSX $\neg a$ is obtained from the seminormal version of the program and then by coherence principle b is true. In $WFSX_{DefL}$ the conflict between a and $\neg a$ is solved by translating the rules into defeasible rules: $a \Leftarrow\sim b$ and $\neg a \Leftarrow$. Then $\neg a$ is derived due to prioritization by defeasible reasoning as described above and by coherence principle b is concluded. However, WFSX is weaker than $WFSX_{DefL}$ since it derives less literals and is less expressive since it does not support the definition of arbitrary conflicts and priorities between rules and rule sets. For instance, consider the following program from [Ant02].

Example 3

$p \Leftarrow$
$\neg p \Leftarrow$
$q \Leftarrow$
$\neg q \Leftarrow p$

The corresponding seminormal version in WFSX is:

$p \leftarrow\sim \neg p$
$\neg p \leftarrow\sim p$
$q \leftarrow\sim \neg q$
$\neg q \Leftarrow p, \sim q$

In $WFSX_{DefL}$ q is provable because the rule with head $\neg q$ is not applicable since $\neg p$ is provable according to the priority rule in $WFSX_{DefL}$ which defines that negative rules override complementary positive rules. In WFSX q is not entailed as can be concluded from the seminormal form.

Note that it has been argued in [Bre01], that defeasible logic is in general different from well-founded semantics, the latter being able to draw more conclusions due to:

1. the lack of loop checking in defeasible logic,
2. the ambivalent role of strict rules which might become defeasible rules
3. the preference handling which only considers conflicts between positive and negative conflicts.

However, the author compares conventional defeasible logic (as defined by Nute [Nut94]) with a direct translation into the seminormal form of an extended logic program. As it was illustrated in the example above such a direct translation must fail, whereas in the case of a translation of a defeasible theory Φ into the well-founded defeasible meta program $P(\Phi)$, the sceptical conclusions of $P(\Phi)$ under WFSX correspond to the conclusions of the defeasible theory Φ. The well-founded defeasible logic handles loops due to the reconstruction in the extended well-founded semantics. Moreover, the extend well-founded defeasible logic allows expressive integrity constraints and preferences between rules and rules sets (modules), akin to ordered logic programing (OLP) [BLR96]) where components (sets of rules) can be prioritized against each other. In contrast to OLP, which handles only negation as failure and predefined preferences of components by inheritance hierarchies, ContractLog supports two kinds of negation and explicit preferences represented in terms of superiority clauses which might be conditional or be adapted dynamically by knowledge updates. If a program does not contain explicit negation and defeasible rules the semantics coincides with the well-founded semantics for normal LPs.

In summary, the declarative semantics $WFSX_{DefL}$ defines the meaning of a ContractLog LP by specifying the intended model among all models of the program. The coherence principle from WFSX establishes a relation between default and explicit negation. The defeasible logic extension of the extended well-founded semantics automatically solves (user-defined) integrity conflicts, such as inconsistency between positive and negative conclusions. Since the defeasible inference rules in ContractLog are given in terms of a meta program a strong commitment to a particular LP semantics can be avoided. Hence, interpretations of ContractLog LPs with other semantics, e.g., answer set semantics, are possible, although well-founded semantics due to its computational, structural and expressive properties, as discussed above, qualifies to be the semantics of choice for ContractLog. In the next section I will introduce a linear top-down procedural semantics, called SLE resolution (**L**inear resolution for **E**xtended LPs with **S**election rule) to compute the well-founded model for extended LPs. I have implemented the approach in the RBSLA/ContractLog inference engine, which is integrated into the Prova project since version 2.0 [KPS06]. I will focus on top-down approaches - see section 3.3.2.4 for references on bottom-up semantics.

4.1.3 Procedural Semantic of ContractLog

The well-known 2-valued top-down SLDNF (Prolog) resolution [Cla78], a resolution based method derived from SLD resolution [KK71, AE82], as a procedural

semantics for LPs has many advantages. Due to its linear derivations it can be implemented using efficient stack-based memory structures, it supports very useful sequential operators such as cut, denoted by !, or *assert/retract* and the negation-as-finite failure test is computationally quite efficient. Nevertheless, it is a too weak procedural semantics for unrestricted LPs with negations. It does not support goal memoization and suffers from well-known problems such as redundant computations of identical calls, non-terminating loops or floundering. It is not complete for LPs with negation or infinite functions. Moreover, it can not answer free variables in negative subgoals since the negation as finite failure rules is only a simple test. For more information on SLDNF-resolution I refer to [Llo87, AB94]. For typical unsolvable problems related to SLDNF see e.g. [She91].

SLG resolution [CW93, CSW95] is the most prominent tabling based top-down method for computing the well-founded semantics for normal LPs. SLG resolution overcomes infinite loops and redundant computations by tabling. The basic idea of tabling, as implemented e.g., in ODLT resolution [TS86], is to answer calls (goals) with the memorized answers from earlier identical goals which are stored in a table. However, SLG resolution is a non-linear approach. SLG is based on program transformations using six basic transformation rules, instead of the tree-based approach of SLDNF. It distinguishes between solution nodes, which derive child nodes using the clauses from the program and look-up nodes, which produce child nodes using the memorized answers in the tables. Since all variant subgoals derive answers from the same solution node, SLG resolution essentially generates a search graph instead of a search tree and jumps back and forth between lookup and solution nodes, i.e., it is non-linear. Special delaying literals are used for temporarily undefined negative literals and a dependency graph is maintained to identify negative loops. Calls to look-up nodes will be suspended until all answers are collected in the table, in contrast to the linear SLD style where a new goal is always generated by linearly extending the latest goal. It is up to this non-linearity of SLG that tabled calls are not allowed to occur in the scope of sequential operators such as cut.

Global-SLS resolution [Prz89a, Prz89b, Ros92] for WFS is a procedural semantics which directly extends SLDNF-resolution and hence preserves the linearity property of SLDNF. In contrast to SLDNF-trees, SLS-trees treat infinite derivations as failed and recursions through negation as undefined. However, it assumes a positivistic computation rule that selects all positive literals before negative ones and inherits the problem of redundant computations from SLDNF. Moreover, a query fails if the SLS-trees for the goal either end at a failure leave or are infinite, which makes Global-SLS computationally ineffective [Ros92]. To avoid redundant computations in SLS a tabling approach called tabulated SLS resolution [Bol98] was proposed. But the approach, like SLG, is based on non-linear tabling.

SLX resolution [ADP94] is a procedural semantics for extended LPs which is sound and theoretically complete wrt WFSX semantics. As in SLS resolution it uses a failure rule to solve the problems of infinite positive recursions and dis-

tinguishes two kinds of derivations for proving verity (SLX-T tree) and proving non-falsity (SLX-TU tree) in the well-founded model in order to fail or succeed literals involved in recursion through negation. Thus, SLX does not consider a temporal undefined status as the other top-down approaches for WFS do, but implements the following derivations: if a goal L is to be undefined wrt WFS it must be failed, if it occurs in a SLX-T derivation and refuted if it occurs in a SLX-TU derivation. To fulfill the coherence requirement of WFSX a default negated literal $\not L$ is removed from a goal if there is no SLX-TU refutation for L or if there is one SLX-T refutation for $\neg L$. In short, SLX is very close to SLDNF resolution. As already pointed out by the authors [ADP94, ADP95] it is only theoretically complete, does not guarantee termination since it lacks loop detection mechanisms, is in general not effective and makes redundant computations since tabling is not supported. Its implementation is given as a meta program in Prolog.

I will now introduce SLE resolution (Linear resolution with Selection function for Extended WFS), which I implemented in the RBSLA/ContractLog inference engine to compute extended WFS. It extends linear SLDNF with goal memoization based on linear tabling and loop cutting. In short, it resolves infinite loops and redundant computations by tabling without violating the linearity property of SLD style resolutions. The main advantages are:

- linear tabling to resolve infinite loops and redundant computations while preserving the linearity of derivations, enabling tabled predicates within the scope of sequential operators such as cuts and procedural attachments to external methods.
- efficient memory structures in trampoline style in order to overcome stack-overflows for very large derivation trees (large knowledge bases)
- extended key indexing to effectively query complex literals, e.g., defeasible meta literals or explicitly negated literals (faster access to KB and narrower search space)
- provides an undefined truth value as in WFS and a temporarily undefined truth value, which enables long running, distributed derivations in (open) distributed knowledge bases (such as in the Semantic Web) to be temporarily undefined
- terminating and sound and complete wrt to the extended well-founded defeasible logic semantics $WFSX_{DefL}$
- its time complexity is comparable with SLG resolution, which has polynomial time data complexity for well-founded negation
- downward compatible with SLDNF resolution, i.e., the tabling cache is highly flexible and supports automated or user-defined management of its size and validity of tables in local and global derivations; it can be completely turned of which reduces SLE resolution to SLDNF

SLE resolution is based on four truth values: t (true), f (false), u (undefined) and u' (temporarily undefined) with $t = \neg f$, $\neg f = t$, $\neg u = u$ and $\neg u' = u$ and a truth ordering $\neg f > t > u > u'$. u' will be used if the truth value of a subgoal is temporarily undecided. SLE resolution follows SLDNF, where derivation trees are constructed by resolution. For more information on the notion of trees for describing the search space of top-down proof procedures see e.g. [Llo87]. In SLE a node in a tree is defined by $N_i : G_i$, where N_i is the node name and G_i is the first goal labelling the node. Tables are used to store intermediate results. In contrast to SLG resolution, there is no distinction between lookup and solution nodes in SLE. The algorithm, always, first tries to answer the call (goal) with the memorized answers in the tables. If there are no answers available in a table the call is resolved against program clauses which are selected in the same top-down order as in SLDNF. This avoids redundant computations. To preserve the order the answers stored in a table are used in a FIFO (first-in-first-out) style, i.e., the first memorized answer is first used to answer the call. In case of loops the two main issues in top-down procedural semantics for WFS are solutions to infinite positive recursions (positive loops) and infinite recursion through negation by default (negative loops).

SLE resolution instantiates the program clauses by goals (calls) in a similar way as in SLD resolution leading to specific instances of these clauses, i.e., a goal G is unified with the heads of appropriate program clauses $H \leftarrow B$, where B is the set of body literals, leading to instances of the clauses $(H \leftarrow B)'$ if there exists a substitution $\theta = \{X_1/t_1, .., X_n/t_n\}$ which assigns terms t_i to variables X_i such that $(H \leftarrow B)' = (H \leftarrow B)\theta$. The instance body $B\theta$ is the goal reduction (sub goal) for further derivation leading to more specific instances. Repeating this process leads to a SLD tree. During this process SLE resolution memorizes intermediate answers for a (tabled) goal (tabled goal memoization) and reuses them to answer variants of calls. This avoids redundant computations.

If the selected call (subgoal) is a variant (child call) of some former call (parent call), i.e., the resolution process is in a positive loop, SLE first uses all answers in the table to resolve the child call. After all tabled answers are used it proceeds with the remaining not yet resolved clauses of the latest parent call to resolve the child call. That is, the child call takes over the remaining task of the parent call and completes the set of answers in the tables, so that no answers are missed from clauses which apply after the looping clause. When all answers are computed the table is marked with a completion flag to indicate the completeness of the table wrt a call (goal).

In case of a negative loop the variant call (subgoal) will be memorized as temporarily undefined in a table. The resolution process proceeds with the remaining clauses which need to be resolved and the temporarily undefined value will be replaced later by the derived true, false or undefined value and the computed answer substitutions. This guarantees correctness of the answer. Again a completion flag will be set, when all outstanding clauses have been resolved. Due to the coherence principle default negated calls are also variant calls of explicit negated calls.

As a result, SLE resolution cuts positive and negative loops by the use of goal memoization and allows to postpone results of temporarily unknown calls in negative loops until all answers are collected. The linearity property of SLD-style resolutions is preserved in SLE since first all tabled answers are used in a FIFO style and then the remaining clauses are applied in their occurrence order. This linearity is crucial to support strictly sequential operators such as cuts in order to impose a more procedural reading on certain parts of the program. This allows implementing "if-then-else"-style rule pairs which are often needed in contracts to define alternatives. And, it allows to situationally prune the search space leading to significant computational savings. Moreover, it is also necessary for serial updates and procedural attachments in rules which must apply in a predefined order. Due to the linearity, the underlying algorithm of SLE resolution can be implemented by an extension of standard abstract machines such as Prolog's ATOAM or WAM, enabling efficient stack-based memory structures.

Remarkably, in my reference implementation of the ContractLog/RBSLA inference engine I use some form of trampoline for the implementation of the SLE resolution algorithm instead of Prologs' standard stack-based recursion. A trampoline is a piece of code that repeatedly calls inner functions by an outer loop. All functions are entered by the trampoline, i.e., by the outer loop. When a function has to call another function, instead of calling it recursively it returns the continuation of the function to be called, the arguments to be used, and so on, to the outer loop. This ensures that the stack does not grow and iteration can continue indefinitely. The sub-goals derived by unification in the proof function are passed by an efficient stack (not the system stack) to the outer trampoline. As a result, the resolution process will never run into the typical stack-overflows for large problem sizes which lead to deeply nested and recursively expanded SLD derivation trees.

In comparison to existing top-down tabling methods for goal memoization the ContractLog approach has the following properties:

- Tabled resolutions supporting goal memoization such as SLG are non-linear approaches which cannot be implemented using an efficient stack-based memory structure.
- Unlike SLG-resolution and other existing tabling methods, the ContractLog memoization does not distinguish between look-up nodes and goal answers. All nodes will be expanded by applying first the existing memorized answers; if no answer can be found in the tables the goal will be expanded in standard linear SLD-style using clauses from the knowledge base.
- Because of the linearity for query evaluation efficient sequential Prolog-style operators such as cuts, procedural attachments with external side effects and serial update rules can be still supported and used, e.g., in order to implement efficient "if-then-else" rule representations or program parts with a more procedural meaning.

In the following sections of this chapter I further elaborate on the ContractLog KR and subsequently extend the core syntax and semantics with further logical formalisms which are needed to adequately represent and execute rule-based SLAs or policies.

4.2 Typed Logic

Traditional logic programming languages such as most Prolog derivatives are typically purely based on the untyped theory of predicate calculus with untyped logical objects (untyped terms). That is, the logical reasoning algorithms apply pure syntactical reasoning and use flat untyped logical objects (terms) in their rule descriptions. From the engineering perspective of a rule-based SLA specification this is a serious restriction which lacks major SE principles such as *data abstraction* or *modularization*. Such principles become important when rule applications grow larger and more complex, are engineered and maintained by different people and are managed and interchanged in a distributed environment over domain-boundaries. To support such SE principles in declarative SLA programming and capture the rule engineer's intended meaning of a contract, types and typed objects play an important role.

SLA rules are typically defined over external business objects and business/-contract vocabularies. Types are an appropriate way to integrate such domain specific ontologies into dynamic domain-independent SLA rules. As a result, such typed rules are much easier to interchange and semantically interpret between domain-boundaries in distributed environments such as the (Semantic) Web, where the domain vocabularies are expressed as ontologies written in expressive ontology languages such as RDF(S) or OWL. From a descriptive point of view types attach additional information, which can be used as *type constraints* for selecting specific goals, i.e., they constrain the level of generality in queries, lead to much smaller search spaces and improve the execution efficiency of query answering.

Moreover, SLM tools and SLA rules do not simply operate on a static internal fact base, but access a great variety of external systems such as system and network management tools, data warehouses / data bases, EJBs, Web Services. It is desirable for any practical rule-based SLA system to enable dynamic access to these external data sources and programming interfaces in order to reuse the existing procedural and highly optimized functionalities for certain tasks in the inference process of deriving answers from rules. For instance, although it is possible to use a LP inference engine for reasoning with numbers and mathematical computations e.g., based on a formalization of the Peano axioms of arithmetic [Lan51], this will never be computationally efficient (since here reasoning is done in an infinite domain which needs to be recursively expanded). Hence, such computations, e.g., adding two numbers and using the resulting sum in the further derivation process, should be dynamically shifted to efficient external procedural code during run-time, e.g., a Java method which

implements an *add* functionality. List computations, aggregations and other complex selection functions are other examples. If the data (facts) are stored in an external relational database highly-optimized SQL queries can be used to temporarily select the needed data, populate the fact base of the rule system, answer the query, and discard the facts from memory afterwards (possibly with some intermediate caching in order to speed up query answering for recurring queries). In fact, as discussed in section 3.1.3 this temporary use of facts during query time, is one of the major advantages of backward-reasoning rule systems.

In the following I will describe the typed logic with procedural attachments provided by ContractLog. I first review the history of types in logic programming languages and describe basic concepts. Then I introduce two external type systems which are supported in the ContractLog KR, namely *Java class hierarchies* with support for highly expressive *procedural attachments* and *Description Logic (DL) Semantic Web ontologies*. Based on the heterogeneous integration, which exploits the optimized reasoning algorithms of external DL reasoners for ontology reasoning, I will elaborate on a *hybrid DL-typed polymorphic unification* as a semantics for a DL-typed logic programming language. Finally, I will conclude this section with a discussion of the hybrid typed logic.

4.2.1 Types in Logic Programming

Definition 92 ***(Type System)*** *A type system [Car97] is responsible for assigning types to variables and expressions. It uses the type expressions for static type checking at compile time and/or dynamic type checking at runtime.*

Type systems typically define a type relation of the form $t : T$, denoting that the term t is of type T. The primary purpose of a type system is to prevent typing errors which violate the intended semantics of a rule or function in a LP. That is, a type system is used to ensure robustness of a LP. By ensuring that the structure is well-defined types enable a more disciplined and regular engineering process and facilitate modularity and partial specification of the intended use of the logical functions and their arguments in a logic program. It has been demonstrated that types play an important role to capture the programmer's intended meaning of a logic program, see e.g. [Nai92] and that they can be used to dramatically reduce the rule search space - see e.g., Schubert's Steamroller Problem which illustrates this advantage [Sti86].

The theory of types in LP has been studied in several works and many different approaches for the introduction of types into logic programming have been proposed reaching from many-sorted or order-sorted systems with subtyping to ad-hoc and parametric polymorphic typing and combinations such as parametric polymorphic order-sorted type systems. Most of the works on type systems and their properties are based on the theory of λ-calculus [Bar88] which gives some fundamentals for reasoning about types in functional languages and which has been generalized to different programming languages such as object-oriented or declarative logic programming languages. Based on this

theory one of the most well-known type systems is the Hindley-Milner type system [Mil78]. Different forms of type declarations have been proposed such as declarations which use a rather general constraint language [HS88], logical formulas [XW88, Nai92], regular sets [Mis84, MR85, HJ92, DZ92a, AE93], equational specifications [Han92] or typed terms over a order-sorted structure [MO84, LR91, SNGM89, HT92, Han91] (see e.g. [Pfe92] for an overview).

In general, the works can be classified into two different views on types in logic programming, namely *descriptive types* and *prescriptive* types [Red88] (aka explicit and implicit types or syntactic and semantic typing). Early work on types in logic programming mainly concentrate on descriptive type systems, e.g. [DZ92b, Mis84, HCC94, Zob87] which attach type information with programs without changing the language used and without affecting the meaning of logic programs. These descriptive approaches are seeking to approximate the structure of a program for use by an optimizing compiler at compile time. For instance, Mycroft and O'Keefe [MO84] demonstrated that the polymorphic type discipline of Milner [Mil78] can be represented in pure Prolog, where the type declarations for variables occur outside of the clauses and do not change the semantics of the pure Prolog program. Dietrich and Hagl's [DH88] extend this approach with input/output mode declarations and subsorts. In contrast, prescriptive type systems, e.g., Gödel [HL94], Typed Prolog [LR91], λ-Prolog [MN86, MNFS91], consider types as properties of the formulas one wants to give a meaning to, i.e., they use a *typed logic* for programming leading to languages with higher expressiveness. The purpose is to identify ill-typed programs, so that the actual semantics of a program satisfies the intended semantics of the rule engineer. Lakshman and Reddy have redefined the Mycroft-O'Keefe type discipline as Typed Prolog [LR91] which adopts the prescriptive view and gives a semantics to the typed logic programs. Several other works follow this approach of defining semantics for prescriptive typed LPs, e.g. [HJ92, HT92, LR91].

These semantics approaches, to which also my ContractLog work contributes, base logic programming with types on typed logics that use sorts (type definitions). Following the terminology of abstract data types [GTW78, EM85] in *many-sorted type systems* sorts are defined by their constructors, i.e., the constituent elements of a sort. The sorts are used to define the number of arguments and their types for every predicate in a logic program. In the many-sorted case sorts are not allowed to have subsort relations between them and accordingly type checking can be done statically at compile time, e.g., realized by a preprocessor without any extensions to the underlying unsorted predicate logic.

In the *order-sorted approach* subsort hierarchies are supported typically by the use of a *order-sorted unification* (typed unification) in order to incorporate some form of *subtyping polymorphism* for untyped variables which assume the type of a unified typed term at runtime. A first order-sorted logic was given by Oberschelp [Obe62] and an order-sorted algebra was developed by Goguen et al. [GM87, Smo89] which forms the basis for the language Eqlog [GM86]. An extended order-sorted algebra with error-handling was proposed by Gogolla [Gog86]. Several other order-sorted approaches have been described using order-

sorted unification [Wal87, HV87].

Different forms of polymorphism such as *generic polymorphism* (see e.g., ML programming language [Mil78]), *ad-hoc polymorphism* or *parametric polymorphism* haven been introduced into logic programming. For a discussion of the differences between parametric and ad-hoc polymorphism see e.g. [Str00]. In the context of polymorphism terms (variables) are authorized to change their types dynamically at runtime, which makes static compile-time analysis insufficient in general. If the type system permits ad-hoc polymorphism, the unifiers for terms are determined by their types and the procedures (logical functions) being invoked are dependent on this information, i.e., the types affect the question of unifiability in an intrinsic way and the computation process must use some form of typed unification procedure to ensure type correctness also during runtime. The types are needed to determine the existence of unifiers and hence also the applicability of clauses. An interesting aspect of this typed semantics is that it enables overloading of clauses leading to different functional variants which are selected dynamically at runtime according to the types of the queries. As a result, the specific inferences that are performed and the correct answers to queries are directly related to the types of the terms in the program clauses and the answers to queries not only display the bindings of variables, but also their types. Typed unification has been studied for order-sorted and polymorphic type systems, see e.g., Typed Prolog [LR91], Protos-L [BB89], λ-Prolog extensions [KNW93].

Order-sorted unification extends the usual term unification with additional dynamic type checking. In a nutshell, the basic idea of sorted unification of two typed variables is to find the greatest lower bound (glb) of their types based on the type hierarchy with subtype relationships, failing if it does not exist. In other words, the unification algorithm tries to find the glb of two sort restrictions yielding a variable whose sort restriction is the greatest common subsort of the two sorts of the unified terms in the given sort hierarchy. This typing approach provides higher levels of abstractions and allows *ad-hoc polymorphism* wrt *coercion*, i.e., automatic type conversion between subtypes, and *overloading*, i.e., defining multiple functions (rules with the same head but different types) taking different types, where the unification algorithm automatically does the type conversion and calls the right function (unifies a (sub-)goal with the right rule head).

Parametric polymorphic types allow to parameterize a structured sort over some other sort, i.e., types and predicates can be written generically so that they can deal equally with any object without depending on their types, in contrast to the many-sorted case, where for each predicate variant having a different type a sort definition must be given explicitly. Typically, parametric polymorphism still maintains full static type-safety on a syntactic level. But, there are approaches with a semantic notion of polymorphic types, e.g., order-sorted parametric polymorphic typed logics which have to take the type information into account at runtime and hence require an extended unification algorithm with type inferencing, as in the order sorted case. These polymorphic type

systems being very complicated artifacts both theoretically and computational wise and have been primarily designed for use in the context of higher-order logic programming. The emphasis in these higher-order type languages has been on describing conditions under which the computationally expensive type analysis can be avoided at runtime which often amounts to banishing ad hoc polymorphism and applying several restrictions to function symbols which must be type preserving. Recent works on types for LPs have concentrated on implementation techniques for efficiently checking or inferring types at runtime, in particular polymorphic types, e.g., by means of abstract interpretations [Lu98] or constraint solving techniques [DGdlBS99].

4.2.2 Description Logic Type System

Until recently, the use of Semantic Web ontology languages such as OWL [MH04] or RDFS [BG04b] has been limited primarily to define meta data vocabularies and add semantic machine-readable meta data to Web pages enabling automated processing of Web contents. But, Semantic Web ontologies also qualify to represent agreement related vocabularies as plug-able webized type systems. They can be used to define domain-specific vocabularies, e.g., WSMO [WSM05b], WS-Policy OWL ontology [PKH05, VAG05], OWL-S [OS03] or KAoS [JCJ+03] or other ontologies such as OWL time [PH04], which can be dynamically integrated into domain-independent contract rules giving the rule terms a domain specific meaning. Both, description logics [NBB+02] which form the logical basis of Semantic Web serialization languages such as OWL (see section 3.4.2) and (Horn) LPs (if restricted to function-free Datalog LPs or normal/extended LPs with finite functions) are decidable fragments of first order logic, however for the most part with orthogonal expressive power. Whereas OWL is basically restricted to unary resp. binary axioms, but e.g., provides classical/strong negation under open world assumption (OWA) and existentially as well as universally quantified variables, Datalog LPs allow n-ary axioms and non-monotonic negations, but are restricted to universal quantifications and are therefore not able to reason about unknown individuals. Clearly, both approaches can benefit from a combination and several integration approaches have been proposed recently.

The works on combining rules and ontologies can be basically classified into two basic approaches: *homogeneous and heterogeneous integrations* [Pas05g]. Starting from the early Krypthon language [BGL85] among the heterogeneous approaches, which hybridly use DL reasoning techniques and tools in combination with rule languages and rule engines are e.g., CARIN [LR96], Life [AKP91], Al-log [DLNS91], non-monotonic dl-programs [ELST04] and r-hybrid KBs [Ric05]. Among the homogeneous approaches which combine the rule component and the DL component in one homogeneous framework sharing the combined language symbols are e.g., DLP [GHVD03], KAON2 [MSS05] or SWRL [HPSB+04]. Both integration approaches have pros and cons and different integration strategies such as reductions or fixpoint iterations are applied with different restrictions to ensure decidability. These restrictions reach from the

intersection of DLs and Horn rules [GHVD03] to leaving full syntactic freedom for the DL component, but restricting the rules to *DL-safe rules* [MSS05], where DL variables must also occur in a non DL-atom in the rule body, or *role-safe rules* [LR96], where at least one variable in a binary DL-query in the body of a hybrid rule must also appear in a non-DL atom in the body of the rule which never appears in the consequent of any rule in the program or to tree-shaped rules [HVNV05]. Furthermore, they can be distinguished according to their information flow which might be *uni-directional* or *bi-directional*. For instance, in homogeneous approaches bi-directional information flows between the rules and the ontology part are naturally supported and new DL constructs introduced in the rule heads can be directly used in the integrated ontology inferences, e.g., with the restriction that the variables also appear in the rule body (safeness condition). However, in these approaches the DL reasoning is typically solved completely by the rule engine and benefits of existing tableau based algorithms in DL reasoners are lost. On the other hand, heterogenous approaches, benefit from the hybrid use of both reasoning concepts exploiting the advantages of both (using LP reasoning and tableaux based DL reasoning), but bi-directional information flow and fresh DL constructs in rule heads are much more difficult to implement. In summary, the question whether the Semantic Web should adopt an homogeneous or heterogenous view is still very much at the beginning and needs more investigation.

In ContractLog I have implemented support for both homogeneous and heterogeneous integration approaches, in the so called *OWL2Prova API* of the ContractLog KR. In the context of term typing using ontologies or object class hierarchies as order-sorted type systems, I apply a hybrid prescriptive typing approach, implemented on top of the OWL2Prova API, which exploits highly optimized external DL reasoner for type checking by means of subsumption checking and instance inference permitting also equivalence reasoning or inner anonymous existentials. In contrast to homogenous approaches such as SWRL [HPSB$^+$04], which is based on the union of OWL-DL and Datalog, or DLP [GHVD03], which is based on the intersection of restricted OWL and Datalog, my heterogeneous approach has the advantage that is does not need any non-standard homogenous reasoning services. Instead, it may reuse existing implementations exploiting e.g., highly optimized external DL reasoner for DL reasoning tasks and a rule engine for the rule reasoning tasks.

In most hybrid approaches, which combine ontologies and rules, the interaction between the subsystems is solved by additional *constraints* in the body of (Datalog) rules, which restrict the values of variables and constants in the *constraint clauses* to range over the instances of the externally specified DL concepts. Although this constraint solution to some extent ensures expressiveness in the sense that it allows to reuse the binary relations (classes and object properties) defined in the external ontology, it has some serious drawbacks in the context of term typing, since this approach leaves the usual operational semantics of resolution and unification unchanged. Hence, the constraints apply only in the body of a rule according to the selection function of the LP inference algorithm which typically selects the "left-most" unified literal as next subgoal.

As a result, according to the standard depth-first strategy the type constraints are used relatively late in the derivation process. In particular, they do not apply directly during the unification between (sub)goals and rule heads, which leads to large and needless search spaces (failed search trees) and might lead to unintended results, since there is no way to directly verify and exclude the unification of typed free (sub)goals and typed rule heads which do not match according to their type definitions. Moreover, fresh typed constant terms, i.e., new DL instances of a particular DL type (class), can not be introduced directly in rule heads since this would require special semantics and formalizations, e.g., with conjunctive rule heads consisting of a literal which defines the individual and another literal which defines its type (or Lloyd-Topor transformations). For instance, a rule with a variable X of type C in the rule head, would need an extra constraint, e.g., $type(X, C)$ in the body: $p(X) \leftarrow q(X), type(X, C)$. Obviously, such a constraint rule is dependent on the order of the body atoms: if $type(X, C)$ is before $q(X)$ the variable X might not be bound to a ground individual and hence can not be verified; on the other hand, if it is after $q(X)$ it does not directly constrain the subgoal $q(X)$, which might be possibly very deep including many variable bindings which are not of type C.

In contrast to these constraint approaches or approaches which apply explicit additional DL-queries/atoms in the rules' body for querying the type system, I have implemented a hybrid typed logic in ContractLog where dynamic type checking for prescriptively typed term is directly integrated into the unification process, i.e., directly implemented within the operational semantics, rather than indirectly through the resolution-based inference mechanism solving the additional constraint/query terms in the body of rules. The interaction during typed unification between the rule component and the DL type system is based on entailment. As a result, the hybrid typed method provides a built-in technical separation between the inferences in the rule component and the type checks in the DL type system (resp. Java type system) which results in:

- in more efficient computations, since the type restrictions directly apply during the unification of typed terms
- higher flexibility with minimal interfaces (based on entailment) between rules component and the type system
- robustly decidable combinations, even in case where the rule language is far more expressive than Datalog and no safeness restrictions apply
- enhanced language expressiveness, e.g., ad-hoc polymorphism with overloading is possible, type casting during order-sorted unification, data abstraction and modularization
- existing highly optimized implementations can be reused, e.g., external DL reasoner for type checking
- rules can be more intuitively modelled and easily combined and interchanged

In the following two subsections I will describe the syntax and semantics of the typed logic of the ContractLog KR. In particular I will elaborate on hybrid description logic programs (*hybrid DLPs*), namely description logic Semantic Web type systems (DL type systems with DL-types) which are used for *term typing of LP rules based on a polymorphic, order-sorted, hybrid DL-typed unification as operational semantics for hybrid DLPs.*

4.2.3 Syntax of Typed ContractLog

ContractLog assumes not just a single universe of discourse, but several domains, so called sorts (types). I first describe the basic extension of the language for extended LPs towards a multi-sorted logic, i.e., the extension of the signature and the variables of the alphabet with sorts (aka types).

Definition 93 *(**Multi-sorted Signature**) A multi-sorted signature S is defined as a tuple $\langle \overline{T}, \overline{P}, \overline{F}, arity, \overline{c}, sort \rangle$ where $\overline{T} = \{T_1, .., T_n\}$ is a set of sort/-type symbols called sorts. The function sort associates with each predicate, function or constant its sorts:*

- *if c is a constant, then $sort(c)$ returns the type T of c.*
- *if p is a predicate of arity k, then $sort(p)$ is a k-tuple of sorts $sort(p) = (T_1, .., T_k)$ where each term t_i of p is of some type T_j, i.e., $t_i : T_j$.*
- *if f is a function of arity k, then $sort(f)$ is a $k+1$-tuple of sorts $sort(f) = (T_1, .., T_k, T_{k+1})$ where $(T_1, .., T_k)$ defines the sorts of the domain of f and T_{k+1} defines the sorts of the range of f*

I define the following three basic types of sorts

1. *primitive sorts are given as a fixed set of primitive data types such as integer, string, etc.*
2. *function sorts are complex sorts constructed from primitive sorts (or other complex sorts) $T_1 \times ... \times T_n \rightarrow T_{n+1}$*
3. *Boolean sorts are (predicate) statement of the form $T_1 \times ... \times T_n$*

Additionally, each variable X_j in the language with a multi-sorted signature is associated with a specific sort: $sort(X_j) = T_i$, which I abbreviate as $X_j : T_i$ (note: sorts must not be distinct for variables). The intuitive meaning is that a predicate or function holds only if each of its terms is of the respective sort given by *sort*. A binary equality predicate $=$ exists in the language. I write $t_1 = t2$ instead of $= (t1, t2)$ to denote that two terms are equal, e.g., $1 = 1$.

Definition 94 *(**Multi-sorted Logic**) A multi-sorted logic associates which each term, predicate and function a particular sort:*

1. *Any constant or variable t is a term and its sort T is given by $sort(t)$*
2. *Let $f(t_1,..,t_n)$ be a function then it is a term of sort T_{n+1} if $sort(f) = \langle T_1,..,T_n,T_{n+1}\rangle$, i.e., f takes argument of sort $T_1,..,T_n$ and returns arguments in sort T_{n+1}.*

I now extend the language to consider external sort/type alphabets. The combined signatures of the rule language and the external type languages form the basis for combined hybrid knowledge bases and the integration of external type systems into the rule system.

Definition 95 ***(Type alphabet)*** *A type alphabet $\overline{T}$ is a finite set of monomorphic sort/type symbols built over the distinct set of terminological class concepts of a (external type) language.*

Informally, a typed ContractLog LP consists of an extended LP with typed terms and a set of external (order-sorted) type systems in which the types (sorts) are defined over their type alphabets. An external type system might possibly define a complete knowledge base with types/sorts (classes) and individuals associated with these types (instances of the classes). Restricted built-in and procedural attachment predicates or functions which construct or return individuals of a certain type (boolean or object-valued) are also considered to be part of the external type system(s), i.e., part of the external signature. The combined signature is then the union of the two (or more) signatures, i.e., the combination of the signature of the rule component and the signatures of the external type systems / knowledge bases combining their type alphabets, their functions and predicates and their individuals.

Definition 96 ***(Combined Signature)*** *A combined signature $\overline{S}$ is the union of all its constituent finite signatures: $\overline{S} = \langle S_1 \cup .. \cup S_n\rangle$*

Based on this definition of a combined signature I now describe the concrete syntax of typed ContractLog. Using equalities ContractLog implements a notion of default inequality for the combined set of individuals/constants which leads to a less restrictive unique name assumption:

Definition 97 ***(Default Unique Name Assumption)*** *Two ground terms are assumed to be unequal, unless equality between the terms can be derived.*

The type systems considered in ContractLog are order-sorted (i.e., with subtype relations):

Definition 98 ***(Order-sorted Type System)*** *A finite order-sorted type system TS comes with a partial order $\leq$, i.e., TS under $\leq$ has a greatest lower bound $glb(T_1,T_2)$ for any two types T_1 and T_2 having a lower bound at all. Since TS is finite also a least upper bound $lub(T_1,T_2)$ exists for any two types T_1 and T_2 having an upper bound at all.*

Currently, ContractLog provides support for two external order-sorted type systems, one is *Java* (implemented in Prova [KS04]) and the other one are *Semantic Web ontologies* (defined in OWL or RDFS) respectively Description Logic KBs. In the following two subsections I define the syntax of typed ContractLog for both external type systems. I have chosen a **prescriptive typing approach**, where types are direct properties of the logical formulas, i.e., are directly attached to terms in a type relation $t : T$, denoting that a term t has a type T. This has several advantages for the semantics as I will discuss later. The EBNF syntax for the typed ContractLog language has been already given in section 4.1.1.

4.2.3.1 Java-typed ContractLog Syntax

The combined multi-sorted signature $\overline{S}_{Java}$ uses the fully qualified order-sorted Java class hierarchy as type symbols. In order to type a variable with a Java type the fully qualified name of the Java class to which the variable should belong must be specified as a prefix separated from the variable by a dot ".".

Example 4

```
java.lang.Integer.X          variable X is of type Integer
java.util.Calendar.T         variable T is of type Calendar
java.sql.Types.STRUCT.S      variable S is of SQL type Struct
```

Java objects, as instances of Java classes, can be dynamically constructed by calling their constructors or static methods using highly-expressive procedural attachments. The returned objects, might then be used as individuals/constants that are bound by an equality relation (denoting typed unification equality) to appropriate variables, i.e., the variables must be of the same type or of a super type of the Java object (see type unification).

Procedural Attachments A procedural attachment is a function that is implemented by an external procedure (i.e., a Java method). They are used to dynamically call external procedural methods during runtime, i.e., they enable the (re)use of procedural code and allow dynamic access to external data sources and tools using their programming interfaces (APIs). Hence, in particular in the SLA domain, they are a crucial extension to traditional logic programming, combining the benefits of object-oriented languages (Java) with declarative rule based programming, e.g., in order to externalize mathematical computations such as aggregations to highly optimized procedural code in Java or use query languages such as SQL by JDBC to select and aggregate facts from external data sources.

Definition 99 ***(Procedural Attachments)*** *A procedural attachment is a function or predicate whose implementation is given by an external procedure. Two types of procedural attachments are distinguished:*

- ***Boolean-valued attachments*** *(or* ***predicate attachments****) which call methods which return a Boolean value, i.e., which are of Boolean sort (type).*
- ***Object-valued attachments*** *(or* ***functional attachments****) which are treated as functions that take arguments and return one or more objects, i.e., which are of a function sort.*

Functional Java attachments have a left-hand side with which the results (the returned object(s)) of the call are unified by a unification equality relation $=$, e.g., $C = java.util.Calendar.getInstance()$. If the left-hand side is a free (unassigned) variable the latter stores the result of the invocation. If the left-hand side is a bound variable or a list pattern the unification can succeed or fail according to the typed unification (see semantics) and consequently the call itself can succeed or fail. List structures are used on the left-hand side to allow matching of sets of constructed/returned objects to specified list patterns. A predicate attachment is assumed to be a test in such a way that the call succeeds only if a true Boolean variable is returned. Static, instance and constructor calls are supported in both predicate and functional attachments depending on their return type. Constructor calls follow the Java syntax with the fully qualified name of the class and the constructor arguments, e.g., $X = java.lang.Long(123)$. Static method calls require fully qualified class names to appear before the name of the static method followed by arguments, e.g., $Z = java.lang.Math.min(X, Y)$. Instance methods are mapped to concrete classes dynamically based on the type of the variable, i.e., the method of a previously bound Java object is called. They require a variable before the name of an instance method followed by the arguments, e.g., $S = X.toString()$.

Example 5

```
add(java.lang.Integer.In1,java.lang.Integer.In2,Result):-
   Result = java.lang.Integer.In1 + java.lang.Integer.In2.
add(In1, In2,Result):-
   I1 = java.lang.Integer(In1),
   I2 = java.lang.Integer(In2),
   X = I1+I2,
   Result = X.toString().
```

The first rule takes two Integer variables $In1$ and $In2$ as input and returns the result which is bound to the untyped variable *Result*. Accordingly, a query $add(1, 1, Result)$? succeeds with an Integer object 2 bound to the *Result* variable, while a query $add(\text{"}abc\text{"}, \text{"}def\text{"}, Result)$? will fail. The second rule takes the untyped variables $In1$ and $In2$ as input arguments and first tries to construct Integer objects from the inputs by a constructor call *java.lang.Integer*, then adds the two constructed Integer objects and returns the results which is transformed back to a String value. Accordingly, a query $add(\text{"1"}, \text{"1"}, Result)$?

with String number representations as inputs will succeed. Primitive datatypes such as Strings and Integers are directly supported, e.g., a constant 1 is automatically interpreted of the primitive sort/type *Integer* and a constant *abc* or "*abc*" is of type *String*.

It is important to note, that Java objects can be bound to variables and their methods can be dynamically used as procedural attachment functions anywhere during the reasoning process, i.e., in other rules. This enables a tight and highly expressive integration of external object oriented functions into declarative rules' execution.

Built-Ins

Definition 100 ***(Built-in Predicates or Functions)*** *Built-in predicates or functions are special restricted predicate resp. function symbols in the language for concrete domains, e.g., integers or strings, that may occur in the body of a rules.*

Examples are $+$, $=$, *assert*, *bound* etc. For a complete overview of all built-ins see the ContractLog/Prova documentation [KP06].

Input/Output Mode Declarations Procedural attachments and built-ins as well as unrestricted functions may compromise the safety of rules, since they may yield to infinite relations. To ensure safety I require that each variable occurring as an argument of a procedural attachment, built-in predicate/function or recursive function must be previously bound. That is, variables are only allowed to occur in the universal closure of a closed functional or procedural formula. To represent this restriction for functions and attachments, I define modes in addition to types in ContractLog. That is, I extend the combined signature $\overline{S}$ with mode declarations (mode functions) of the form $+, -, ?$ and a function *mode* that associates with each k-ary predicate or function the mode declaration m_i of its argument terms $mode(p) = (m_1, .., m_k)$ where each mode definition m_i is of some declaration $+, -, ?$.

Definition 101 ***(Modes)*** *Modes are states of instantiation of a predicate / function described by mode declarations +,-,?, i.e., declarations of the intended input-output constellations of the predicate / function terms with the following syntax and semantics:*

- *"+" The term is intended to be input*
- *"−" The term is intended to be output*
- *"?" The term is undefined/arbitrary (input or output)*

To syntactically represent modes in ContractLog I use the special built-in predicates $bound(Variable)$ to state that the variable must be bound to a ground value, i.e., is intended to be an input variable, or $free(Variable)$ to state that the variable must be free, i.e., is intended to be an output variable. Most standard Prolog interpreters provide such built-in predicates.

Example 6

```
add(In1,In2,Result):-
   bound(In1),
   bound(In2),
   free(Result),
   Result = In1 + In2.
```

In the example $In1$ and $In2$ are input variables and *Result* is the output variable, i.e., the mode declaration is $add(+,+,-)$. Accordingly, a query $add(1,1,Result)$? will succeed with 2 as its output whereas a query $add(In, 2, Result)$? will fail if the variable In is a free variable, which is desirable for the predicate function *add*, because for such a query the number of potential answers would be infinite.

4.2.3.2 DL-typed ContractLog Syntax

The second sort of type systems supported by ContractLog are webized Semantic Web ontologies defined in RDFS or OWL (OWL Lite or OWL DL). That is, the combined signature $\overline{S}_{DL}$ consisting of the finite signature S of the rule component and the finite signature(s) S_i of the ontology language(s). Note that the ContractLog approach is general and can be extended to different DL languages defining DL KBs as type systems. Hence, in the following I use the term DL also for Semantic Web ontology languages and DL knowledge bases. For the reason of understandability, I assume only one external type system in the following.

The type alphabet TS is a finite set of monomorphic type symbols built over the distinct set of terminological atomic concepts $\overline{T}$ in a Semantic Web ontology language Σ^{DL}, i.e., defined by the atomic classes in the T-Box model. Note, that restricting types to atomic concepts is not a real restriction, because for any complex concept such as $(T_1 \sqcap T_2)$ or $(T_1 \sqcup T_2)$ one may introduce an atomic concept T_3 in the T-Box and use T_3 as atomic type instead of the complex concept. This approach is also reasonable from a practical point of view since dynamic type checking must be computationally efficient in order to be usable in an order-sorted typed logic with possible very large rule derivation trees and many typed unification steps, i.e., fast type checks are crucial during typed term unification. I assume that the type alphabet is fixed (but arbitrary), i.e., no new terminological concepts can be introduced in the T-Box by the rules at runtime. This ensure completeness of the domain and enables static type checking on

the used DL-types in ContractLog LPs at compile time (during parsing the LP script).

The set of constants/individuals $\overline{c}$ is built over the set of individual names in Σ^{DL}, but I do not fix the constant names and allow arbitrary fresh constants (individuals) (under default UNA) to be introduced in the head of rules and facts of the rule base. The precise syntax of DL-typed terms is defined as follows:

Definition 102 ***(DL-typed Terms)*** *A DL-type is a terminological concept/-class defined in the DL-type system (T-Box model) possibly prefixed by a URI namespace abbreviation separated by _ , i.e., "namespace_Type". A typed term is denoted by the relation $T : t$ for typed variable terms and $T : t$ for typed constant terms, i.e., $ns\_a : ns\_C$ denoting that individual ns_a is of type ns_C or $X : ns\_C$, i.e., variable X is of type ns_C.*

The type ontologies are typically provided on the Web under an unique URL and types and individuals are represented as resources having an webized URI. Namespaces can be used to avoid name conflicts and namespace abbreviations facilitate are more readable language. Note, that fresh individuals which are introduced in rules or facts apply locally within the scope of the predicate/function and rules in which they are defined, i.e., within a local reasoning chain; in contrast to the individuals defined in the A-box model of the type system which apply globally as individuals of a class.

Example 7

```
% Import external type systems

import("http//../dl_typing/businessVocabulary1.owl").
import("http//../dl_typing/businessVocabulary2.owl").
import("http//../dl_typing/mathVocabulary.owl").
import("http//../dl_typing/currencyVocabulary.owl").

reasoner("dl"). % configure reasoner (OWL-DL=Pellet)


% Rule-based Discount Policy

discount(X:businessVoc1_Customer, math_Percentage:10) :-
    gold(X: businessVoc1_Customer).
discount(X: businessVoc1_Customer, math_Percentage:5) :-
    silver(X: businessVoc1_Customer).
discount(X: businessVoc1_Customer, math_Percentage:2) :-
  bronze(X: businessVoc1_Customer).

% Note that these rules use a different vocabulary
```

```
% if the types "Client" and "Customer" are equal
% both typed rule sets unify
% Class equivalence between both "types"
% is defined in the second OWL-DL ontology

gold(X:businessVoc2_Client) :-
   spending(X:businessVoc2_Client, S:currency_Dollar),
   S:currency_Dollar > currency_Dollar:1000.
silver(X:businessVoc2_Client) :-
   spending(X:businessVoc2_Client, S:currency_Dollar),
   S:currency_Dollar > currency_Dollar:500.
bronze(X:businessVoc2_Client) :-
   spending(X:businessVoc2_Client, S:currency_Dollar),
   S:currency_Dollar > currency_Dollar:100.


% Facts

spending(businessVoc1_Customer:Adrian, currency_Dollar:1000).
spending(businessVoc1_Customer:Aira, currency_Dollar:200).

% Query

:-solve(discount(X:businessVoc_Client, Y:math_Percentage)).
```

The example shows term typing with "plug-able" (imports of) Semantic Web type vocabularies. Remarkably, $businessVocabulary1$ defines a type $Customer$ which is defined to be equal to the type $Client$ in $businessVocabulary2$ (not shown here). Hence, both types unify and the first three rules unify with the second three rules of the discount policy and the defined facts. The user can use both types interchangeable to define queries on the hybrid KB, i.e., the domain-specific vocabulary of choice can be used. This leads to a very flexible and compact language design where rules can be given a domain-specific meaning with a precise semantics by using external domain-specific vocabularies as type systems for terms.

Free DL-typed variables are allowed in facts. They act as free instance queries on the ontology layer, i.e., they query all individuals of the given type and bind them to the typed variable. In addition ContractLog provides a special query predicate which can be used in the body of rules to interact with the ontology component and explicitly expressed queries, such as concept membership, role membership or concept inclusion on the DL knowledge base. The special query predicate *rdf* (implemented in owl.prova library) is used to query external ontologies written in RDF(S) or OWL (OWL Lite or OWL DL).

Example 8

```
% Bind all individuals of type "Wine" to the variable "Subject"
% using the owl ontology WineProjectOWL.owl and the "rdfs" reasoner

rdf("./examples/function_tests/owl/testdata/WineProjectOWL.owl",
"rdfs",Subject,"rdf_type","http://www.ontologies.com/wine.owl#Wine")

% Use the transitive reasoner and namespace abbreviations

rdf("./rules/function_tests/owl/testdata/WineProjectOWL.owl",
"transitive",Subject,"rdfs_subClassOf","default_Wine")
```

The first argument specifies the URL of the external ontology. The second argument specifies the external reasoner which is used to infer the ontology model and answer the query. Note, that this hybrid method using an external reasoner to answer queries provides a technical separation between the inferences in the Description Logic part which is solved by an optimized external DL reasoner and the Logic Programming component which is solved by the rule engine. As a result, the combined approach is robustly decidable, even in case where the rule language is far more expressive than Datalog. Moreover, the triple-based query language also supports queries to plain RDF data sources, e.g., Dublin Core meta data. The following predefined reasoner are supported:

- "" | "empty" | null = no reasoner
- default = OWL reasoner
- transitive = transitive reasoner
- rdfs = RDFS rule reasoner
- owl = OWL reasoner
- daml = DAML reasoner
- dl = OWL-DL reasoner
- swrl = SWRL reasoner
- rdfs_full = rdfs full reasoner
- rdfs_simple = rdfs simple reasoner
- owl_mini = owl mini reasoner
- owl_micro = owl micro reasoner

User-defined reasoners can be easily configured and used. By default, the selected reasoner is used to query the external models on the fly, i.e., to dynamically answer the queries using the external reasoner. But, ContractLog also supports a pre-processing mode. Here the reasoners are used to pre-infer the ontology model, i.e., build an inferred RDF triple model where the logical DL entailments such as transitive subclasses are already resolved at compilation time. Queries then operate on the inferred model which is much faster,

but with the drawback that updates of the ontology model require a complete recompilation of the inferred model.

Example 9

```
:-eval(owl2prova(
            transitive, % reasoner
            [predicate,subject,object], % converter pattern
            "wine.owl", % input file
            "wine.prova")). % inferred output file
```

Special converters can be used to translate the ontology model into a *homogeneous LP representation*, i.e., in LP facts and homogeneous DLP inference rules. In this homogenous integration mode queries to the DLP component are directly answered by the rule engine using the DL meta inference rules and the translated DL facts. The following predefined converters are supported by the OWL2Prova API of the ContractLog KR:

- simple = simple converter
- dlp = homogeneous DLP (Description Logic Programs) with instance equivalence
- defeasible = defeasible converter

The *simple converter* allows arbitrary translation patterns to be specified by the user, e.g., ["*predicate*","*subject*","*object*"] or ["*rdf*","*subject*","*predicate*", "*object*"]. The terms *predicate*,*subject* and *object* are restricted key words in the user-defined translation patterns which give the position of the subject, object and property of the input RDF triple in the LP output fact. The first term in the pattern always becomes the predicate name of the output fact.

Example 10
Translation pattern is ["*predicate*","*subject*","*object*"]
RDF input triple is: $rbsla:User, rdfs:subClassOf, rbsla:Customer$
LP output fact is: $rdfs\_subClassOf(rbsla\_User, rbsla\_Customer)$

Translation Pattern is: ["*rdf*","*subject*","*predicate*","*object*"]
RDF input triple is: $rbsla:User, rdfs:subClassOf, rbsla:Customer$
LP output fact is: $rdf(rbsla\_User, rdfs\_subClassOf, rbsla\_Customer)$

In the example, according to the first translation pattern, the property of the RDF triple becomes the predicate name of the output fact and *subject* and *object* become the terms within the binary fact. In the second translation pattern the predicate name is *rdf* and subject, property and object of the RDF triple become the terms of the RDF fact.

The *dlp converter* follows the homogeneous integration approach as described in [GHVD03]. It translates RDFS and OWL ontologies into homogeneous DLPs. For instance:

Example 11
RDFS examples and translation into DLP:

$C \sqsubseteq D$, *i.e., class* C *is subclass of class* D: $C(X) \leftarrow D(X)$.
$Q \sqsubseteq P$, *i.e.,* Q *is a subproperty of* P: $P(X,Y) \leftarrow Q(X,Y)$.
$T \sqsubseteq \forall P.C$, *i.e., the range of property* P *is class* C: $C(Y) \leftarrow P(X,Y)$.
$T \sqsubseteq \forall P^{-}.C$, *i.e., the domain of property* P *is class* C: $C(X) \leftarrow P(X,Y)$.
$a : C$, *i.e., the individual* a *is an instance of class* C: $C(a)$.
$< a,b >: P$, *i.e., the individual* a *is related to the individual* b *by the property* P: $P(a,b)$

OWL examples:

$C \equiv D$, *i.e., class* C *is equivalent to class* D: $C(X) : -D(X).D(X) : -C(X)$.
$P \equiv Q$, *i.e., the property* P *has the same extension as the property* Q: $P(X,Y) : -Q(X,Y)$.
$P^{+} \sqsubseteq P$, *i.e., the property* P *is transitive:* $P(X,Z) : -P(X,Y), P(Y,Z)$

To overcome restrictions of the homogeneous DLP approach I have implemented several extensions in the DLP converter. For example, I define an instance mapping to represent equivalent (owl:sameAs) individuals with an unique ID (also for undefined individuals). This allows representing individual equivalence and undefined individuals which are missing in the homogeneous DLP fragment as defined by [GHVD03].

4.2.4 Semantics of Typed ContractLog

4.2.4.1 Declarative Semantics: Multi-Sorted Logic

As discussed in section 3.3.2.2 the semantics of a LP P is typically defined wrt to the closed Herbrand universe $U(P)$. I now extend the domain of discourse towards a combined knowledge base defined over a combined signature where individuals and types (sorts) from one or more type systems outside of the fixed domain of the rule component are taken into account. The semantics of the combined KB based on an extended Herbrand base is then defined wrt to the combined signature.

Definition 103 ***(Combined Knowledge Base)*** *The combined knowledge base of a typed ContractLog LP* $\overline{KB} = \langle \Phi, \Psi \rangle$ *consists of a finite set of (order-sorted) type systems / type knowledge bases* $\Psi = \{\Psi_1 \cap .. \cap \Psi_n\}$ *and a typed ContractLog KB* Φ.

The combined signature is the union of all constituent signatures, i.e., each interpretation of a ContractLog LP has the set of ground terms of the combined signature as its fixed universe.

Definition 104 *(**Extended Herbrand Base**) Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a typed combined ContractLog LP P. The extended Herbrand base of P, denoted $\overline{B}(P)$, is the set of all ground literals which can be formed by using the predicate symbols in the combined signature with the ground typed terms in the combined universe $\overline{U}(P)$, which is the set of all ground typed terms which can be formed out of the constants, type and function symbols of the combined signature.*

The grounding of the combined KB is computed wrt the composite signature.

Definition 105 *(**Grounding**) Let P be a typed (combined) ContractLog LP and $\overline{c}$ its set of constant symbols in the combined signature. The grounding $ground(P)$ consists of all ground instances of all rules in P w.r.t to the combined multi-sorted signature which can be obtained as follows:*

- *The ground instantiation of a rule r is the collection of all formulae $r[X_1 : T_1/t_1, .., X_n : T_n/t_n]$ with $X_1, .., X_n$ denoting the variables and $T_1, .., T_n$ the types of the variables (which must not necessarily be disjoint) which occur in r and $t_1, .., t_n$ ranging over all constants in $\overline{c}$ wrt to their types.*
- *For every explicit query/goal $Q[X_1 : T_1, .., X_m : T_m]$ to the type system, being either a fact with one or more free typed variables $X_1 : T_1, .., X_m : T_m$ or a special query atom $rdf(...)$ with variables as arguments in the triple-like query, the grounding $ground(Q)$ is an instantiation of all variables with constants (individuals) in $\overline{c}$ according to their types.*

The interpretation I of a typed program P then is a subset of the extended Herbrand base $\overline{B}(P)$.

Definition 106 *(**Multi-sorted Interpretation**) Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a combined KB and $\overline{c}$ its set of constant symbols. An interpretation I for a multi-sorted combined signature $\overline{S}$ consists of*

1. *a universe $|M| = T_1^I \cup T_2^I \cup .. \cup T_n^I$, which is the union of the types (sorts), and*
2. *the predicates, function symbols and constansts/individuals $\overline{c}$ in the combined signature, which are interpreted in accordance with their types.*

The assignment function σ from the set of variable $\overline{X}$ of P into the combined universe $\overline{U}(P)$ must respect the sorts/types of the variables (in order-sorted type systems also subtypes). That is, if X_i is a variable of type T, then $\sigma(X) \in T^I$. In general, if ϕ is a typed predicate or function in Φ and σ an assignment to the interpretation I, then $I \models \phi[\sigma]$, i.e., ϕ is true in I when each variable X of ϕ is

substituted by the values $\sigma(X)$ wrt to its type. Since the assignment to constant and function symbols is fixed and the domain of discourse corresponds one-to-one with the constants $\overline{c}$ in the combined signature $\overline{U}(P)$, it is possible to identify an interpretation I with a subset of the extended Herbrand base: $I \subseteq \overline{B}(P)$.

The assignment function is then given as a query from the rule component to the type system, so that there is a separation between the inferences in a type system and the rule component. Moreover, explicit queries to a type system (Java or Semantic Web) defined in the body of a rule, e.g., procedural attachments, built-ins or ontology queries (special *rdf* query or free DL-typed facts) are based on this hybrid query mechanism. The query interaction between the rules and the type system is based on entailment. I now define the notion of model for a typed ContractLog LP

Definition 107 ***(Model)*** *Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a combined KB of a typed ContractLog LP P.*
An interpretation I is a model of an untyped ground atom $A \in \overline{KB}$ or I satisfies A, denoted $I \models A$ iff $A \in I$.
I is a model for a ground typed atom $A : T \in \overline{KB}$, or I satisfies $A : T$, denoted $I \models A : T$, iff $A : T \in I$ and for every typed term $t_i : T_j$ in A the type query $T_j = sort(t_i)$, denoting the type check "is t_i of type T_i", is entailed in $\overline{KB}$, i.e., $\overline{KB} \models T_i = sort(t_i)$ (in an order sorted type system subtypes are considered, i.e., t_i is of the same or a subtype of T_j).
I is an interpretation of an ground explicit query/goal Q to the type system Ψ if $\Psi \models Q$.
I is a model of a ground rule $r : H \leftarrow B$ iff $I \models H(r)$ whenever $I \models B(r)$. I is a model a typed program P (resp. a combined knowledge base $\overline{KB}$), denoted by $I \models P$, if $I \models r$ for all $r \in ground(P)$.

Note, that the definition considers the combined KB $\overline{KB}$, i.e., fresh typed individuals/constants are allowed to be introduced in the head of rules. To support WFS as default semantics for negated queries to external type systems in ContractLog, i.e., the negated type query $\sim Q$ succeeds, if the query fails, i.e., $\Psi \not\models Q$, I extend the definition of unfounded sets with additional query atoms to the external type systems (knowledge bases).

Definition 108 ***(Unfounded Set)*** *Let P be a typed LP (combined KB). Let I be a partial interpretation. Let $\overline{A} \subseteq \overline{B}_P$ be a set of ground atoms. α is an unfounded set of P wrt I, if for every atom $A \in \alpha$ and every ground rule $r \in ground(P)$ with a finite sequence of ground standard literals and type query atoms (querying the type system) in $B(r)$ at least one of the following conditions holds:*

1. *at least one standard body literal $L \in B(r)$ is false in I.*
2. *at least one standard positive body literal $L \in B(r)$ is contained in α.*

3. *at least one type query atom* $Q \in B(r)$ *is false in* $I \cup \neg\alpha$.
4. *at least one negative type query atom* $\sim Q \in B(r)$ *is true in* I

Remark 1 *Note, that the structures in Java type systems are usually not considered as interpretations in the strict model-theoretic definition, but are composite structures involving several different structures whose elements have a certain inner composition. However, transformations of composite structures into their flat model theoretic presentations is in the majority of cases possible. From a practical point of view, it is convenient to neglect the inner composition of the elements of the universe of a structure. These elements are just considered as "abstract" points devoid of any inherent meaning. Note that this does not hold for procedural functions and predicates (boolean-valued attachments / built-ins) defined on them.*

This structural mapping between objects from their interpretations in the Java universe to their interpretation in the rule system ignoring finer-grained differences that might arise from the respective definitions is given by the following isomorphism.

Definition 109 *(Isomorphism) Let* I_1, I_2 *be two interpretations of the combined signature* $\overline{S} = \{T_1, .., T_n\}$, *then* $f_{\cong} : |M_1| \rightarrow |M_2|$ *is an isomorphism of* I_1 *and* I_2 *if* $f_{\cong}$ *is a one-to-one mapping from the universe* $|M_1|$ *of* I_1 *onto the universe* $|M_2|$ *of* I_2 *such that:*

1. *For every type* T_i, $t \in T_i^{I_1}$, *iff* $f_{\cong}(t) \in T_i^{I_2}$
2. *For every constant* c, $f_{\cong}(c^{I_1}) \cong c^{I_2}$
3. *For every n-ary predicate symbol* p *with n-tuple* $t_1, .., t_n \in |M_1|$, $\langle t_1, .., t_n \rangle \in p^{I_1}$ *iff* $\langle f_{\cong}(t_1), .., f_{\cong}(t_n) \rangle \in p^{I_2}$
4. *For every n-ary function symbol* f *with n-tuple* $t_1, .., t_n, \in |M_1|$, $f_{\cong}(f^{I_1}(t_1, .., t_n)) \cong f^{I_2}(f_{\cong}(t_1), .., f_{\cong}(t_n))$

For instance, in ContractLog an isomorphism between Boolean Java objects and their model-theoretic truth value is defined, which makes it possible to treat boolean-valued procedural attachments as body atoms in rules and establish an entailment relation as defined above between the Java type system and the model-theoretic semantics of the typed logic of the rule component. Other examples are String objects which are treated as standard constants in rules, i.e., they map with the untyped theory of logic programming. Primitive datatype values, from the ontology resp. XML domain (XSD datatypes) can be mapped similarly.

4.2.4.2 Operational Semantics: Hybrid Polymorphic Order-Sorted Unification

In the following I define the operational semantics of typed ContractLog LPs. In contrast to other hybrid approaches which apply additional constraint atoms as type guards in the rule body and leave the usual machinery of resolution and unification unchanged, the operational semantics for prescriptive types in ContractLog's typed logic is implemented by an order-sorted unification. Here the specific computations that are performed in this typed language are intimately related to the types attached to the atomic term symbols. The order-sorted unification yields the term of the two sorts (types) in the given sort hierarchy. This ensures that type checks apply directly during typed unification of terms at runtime enabling ad-hoc polymorphism of variables leading e.g., to different optimized rule variants and early constrained search trees. Thus, the order-sorted mechanism provides higher level of abstraction, providing more compact and complete solutions and avoiding possibly expensive backtracking.

The standard untyped unification algorithm in logic programming serves as a tool for the resolution principle (see section 3.3.2.1. For a survey on unification theory see, e.g. [BS01, Sie89]. In the following I first define the rules for untyped unification in terms of equation-solving transformations [MM82] for elimination (El), decomposition (De), variable binding (Bi) and orientation (Or). The judgements beneath the horizontal line is the conclusion of the rule and the judgements below the line are the premises of the rule. The computation starts with a set of equations $Eq = \{(t_1 = t'_1), .., (t_n = t'_n)\}$ representing the terms to be unified, and to transform Eq into the solved set of equations Eq' using the following four transformation rules El, De, Bi and Or:

1. $(El)\ \frac{Eq \vdash (X=X)}{Eq}$, where X is a variable
2. $(De)\ \frac{Eq \vdash (f(t_1,..,t_n)=f(t'_1,..,t'_n))}{Eq \vdash \{(t_1=t'_1),..,(t_n=t'_n)\}}$, where f is a predicate or function
3. $(Bi)\ \frac{Eq \vdash (X=t)}{\theta(Eq) \vdash (X=t)}$, where X is a variable, t is a constant or variable term, and X occurs in Eq but not in t, and where $\theta = \{X/t\}$
4. $(Or)\ \frac{Eq \vdash (t=X)}{Eq \vdash (X=t)}$, where X is a variable and t is not a variable

The computation starts with a set of equations $Eq = \{t_1 = t'_1, ..., t_n = t'_n\}$ where $\{t_i/t'_i\}$ describes the pairs of unifiable terms. Using the four rules Eq is transformed into a set of equations $Eq' = \{X_i = t_i | i \in \{1, .., n\}\}$ where X_i are distinct variables which do not occur elsewhere in Eq'. Unification fails, if there is an equation $(f(t_1, .., t_n) = f'(t'_1, .., t'_m)) \in Eq$ with $f \neq f'$ or if there is an equation $(X = t) \in Eq$ such that $X \in t$ and $X = t$. Eq' is solved if $Eq' = \{(X_i/t_i) | (X_i = t_i) \in Eq'$ and $X_i \in Eq\}$, then $\theta = \{X_1/t_1, .., X_n/t_n\}$ is the mgu of Eq.

I now extend this basic set of unification rules to a hybrid polymorphic order-sorted DL-typed unification. I restrict type checking to finding the lower bound

of two types (T_1, T_2) under the partial order $\leq$ of the order-sorted type model with an upper bound $\top$ and a lower bound $\bot \equiv$ *empty* and replace the type of a term with the more specific type concept. Therefore, I define a *lower* operation, which is applied during typed unification, as follows:

$lower(T_1, T_2) = (T_2/T_1) \rightarrow T_1$, if $T_1 \leq T_2$
$lower(T_1, T_2) = (T_1/T_2) \rightarrow T_2$, if $T_1 > T_2$
$lower(T_1, \top) = (\top/T_1) \rightarrow T_1$
$lower(\top, T_2) = (\top/T_2) \rightarrow T_2$, where $\top = untyped$
$lower(T_1, T_2) = \bot$, otherwise, where $\bot =$ empty type.

Note that, the operation *lower* requires at most two queries to the external type system to compute the lower bound of two types having a lower bound at all.

To enable polymorphic typing of variables during typed unification, where a variable may change its type dynamically, I add the sort function $sort(\bar{t}) = \overline{T}$ to the set of equations Eq, which gives the set $SR = \{t_1 : T_1, .., t_n : T_n\}$ of type restrictions, denoting that the term t_i (currently) has type T_i. That is, for every equation $t_i = t_i' \in Eq$ it must also be that $sort(t_i) \leq sort(t_i')$, i.e., $T_i \leq T_i'$ under the sorted order of types $\leq$. The modified and extended type rules for order-sorted unification are as follows:

1. (El) $\frac{SR, Eq \vdash (X=X)}{SR, Eq}$
2. (De) $\frac{SR, Eq \vdash (f(t_1,..,t_n)=f(t_1',..,t_n'))}{SR, Eq \vdash \{(t_1=t_1'),..,(t_n=t_n')\}}$
3. (Bi') $\frac{SR, (X:T), Eq \vdash (X=t)}{SR', \theta(Eq) \vdash (X=t)}$, where X is a variable, t is a variable or non-variable term, and where $\theta = \{X/t\}$ and $SR, (X : T)$ reduces to SR' using the auxiliary type rules El^{SR}, $El^{SR'}$ and Bi^{SR}
4. (Or) $\frac{SR, Eq \vdash (t=X)}{SR, Eq \vdash (X=t)}$, where X is a variable and t is not a variable

The auxiliary rules for polymorphic unification of types are:

1. (El^{SR}) $\frac{SR \vdash f(t_1,..,t_n):T_2}{SR}$, if $sort(f) = T_1$ and $T_1 \leq T_2$
2. $(El^{SR'})$ $\frac{SR \vdash f(t_1,..,t_n):\top}{SR}$
3. (Bi^{SR}) $\frac{SR \vdash X:T_1 = X:T_2}{SR \vdash X:lower(T_1,T_2)}$

The typed unification fails if:

- there is an equation $f(t_1, .., t_n) = g(t_1', .., t_m')$ in Eq with $f \neq g$ or
- there is an equation $X = t$ in Eq such that $X = t$ and $X \in t$ (occurs) or
- there is an equation $X = t$ in Eq such that $X : T_1$ and $t : T_2$, where t is a constant term and $T_2 > T_1$ or
- there is an equation $X = Y$ in Eq such that $X : T_1$ and $Y : T_2$ and $lower(T_1, T_2) = \bot$, where X and Y are variable terms.

Otherwise, $Eq^{'}$ is solved if $Eq^{'} = \{(X_i/t_i)|(X_i = t_i) \in Eq^{'}$ and $sort(X_i) \leq sort(t_i)$ and $X_i \in Eq\}$, then $\theta = \{X_1/t_1, .., X_n/t_n\}$ is the mgu of Eq.

In contrast to the unsorted unification, Bi' now involves ad-hoc polymorphic unification of order-sorted types with a subtype resp. equivalence test $T_1 \leq T_2$ and a computation of the lower bound of two types $lower(T_1, T_2)$ in the auxiliary rules, possibly assigning the more specific type (i.e., the lower type) to a variable. The variables may change their type during unification according to the rule Bi^{SR} and the *lower* operation. $El^{SR'}$ is introduced to reduce unification to special cases of the binding rule Bi in the untyped case without type checking, i.e., to efficiently process untyped variables $X : \top$. That is, the order-sorted unification coincides with the untyped unification, if the underlying program does not contain typed terms. To ensure decidability I require that there are no infinite function definitions such as $f(f(f(...)))$ and hence introduce the following restriction for typed unification: $\theta = \{X/f(t_1, .., t_n)\}$ if $X \ni f$, i.e., the variable X is not allowed to occur in the function f with which it is unified. Furthermore, I restrict the unification algorithm to only well-typed terms, i.e., the type of the argument t_i in $f(t_1, .., t_n)$ must be a subtype of the type T_i for $f : T_1 \times .. \times T_n \rightarrow T$, where T is the target type of the function. I define the type of predicates, terms and lists to be untyped by default denoted by $\top$. As a result untyped complex terms or lists can be unified only with other untyped variables. Informally the polymorphic order-sorted unification rules state:

- Untyped Unification: Ordinary untyped unification without type checking
- Untyped-Typed Unification:The untyped query variable assumes the type of the typed target
- Variable-Variable Unification:
  (1) If the query variable is of the same type as the target variable or belongs to a subtype of the target variable, the query variable retains its type (according to *lower*), i.e., the target variable is replaced by the query variable.
  (2) If the query variable belongs to a super-type of the target variable, the query variable assumes the type of the target variable (according to *lower*), i.e., the query variable is replaced by the target variable.
  (3) If the query and the target variable are not assignable ($lower = \bot$) the unification fails
- Variable-Constant Term Unification:
  (1) If a variable is unified with a constant of its super-type, the unification fails.
  (2) If the type of the constant is the same or a sub-type of the variable, it succeeds and the variable becomes instantiated.
- Constant-Constant Term Unification: Both constants are equal and the type of the query constant is equal to the type of the target constant.

  Complex terms such as lists are untyped by default and hence are only allowed to be unified with untyped variables.

After having defined the general semantics for order-sorted typed unification, I will now discuss its implementation for the two type systems in ContractLog: Java and Description Logics.

Typed Unification for Java Type Systems The object-oriented type system of Java is essentially a first-order static type system in the sense that is does not allow dynamic type parameterizations. As an object-oriented language it supports subtyping polymorphism using inheritance (aka subclassing) and ad-hoc polymorphism with overloading and coercion (casting) as a kind of automatic type conversion. Although, since Java 1.5 generics are supported [Bra04], the primary approach to polymorphism in Java is still, declaring generic data structures and functions of type *Object* (the class from which all other classes inherit) and explicitly downcast from *Object* to a more specific type. Type safety will be then checked at runtime. Following this approach untyped variables in ContractLog are implicitly declared to be of type *Object* and can assume a specific type when bound to a ground (Java) object at runtime. Although, this adds some ambiguity in the sense that such untyped variables can not be statically type checked at compile time (instead they must be dynamically verified by the typed unification), I argue that in the context of a declarative SLA scripting language the benefits in expressiveness in terms of dynamic typing prevail and the ambiguity can be efficiently handled by the implicit backtracking mechanism of the rule interpreter. The rule interpreter uses the declared types, assuming the root type *Object* if no information is given ($\cong \top$) and tries to unify the terms. To explicitly test subclass and instance relations, as needed for the unification of typed terms, Java provides the operators $isInstance$ and $isAssignableFrom$. This operators are used in the *lower* operation of the typed unification algorithm, i.e., the unification algorithm externalizes the computation of the lower bound of two Java types to the Java runtime.

Typing errors due to not-defined or ill-defined qualified class names (types) are checked during compile time, i.e., during parsing the ContractLog script. complementary to the traditional descriptive usage of the Java type system for static type checking at compile-time in a compiled language, its usage in an interpreted and typed logical language such as ContractLog with a typed unification enables dynamic ad-hoc polymorphic typing of variables. Instances (objects) of particular types are dynamically constructed by procedural attachments and bound to appropriate variables as described in section 4.2.3.1. The operational semantics for such procedural calls is provided by the core Mandarax inference system [Die04] and has been further extended in Prova [KS04, KPS06] with dynamic instantiation based on Java Reflection.

Typed Unification for DL Type Systems A DL resp. Semantic Web type system consists of a T-Box defining the order-sorted types and their relations and a possible empty A-Box defining global individuals of the defined types. The T-Box typically has a partial order $\leq$. ContractLog assumes $owl : Resource$ as common maximum class under the partial order of any DL type system. I define

$owl : Resource$ to be $\top$ (i.e., $\cong$ untyped) and $owl : Nothing$ to be the minimum lower bound $\bot$ (i.e., empty). Note that both type systems, Java and DL, coincide in the untyped framework which can be downcasted from *java.lang.Object* or $owl : Resource$. The DL type checks, applied as hybrid queries to the DL ontology type system(s) during the unification process, are primarily concerned with *Instantiation*, i.e., querying whether an individual is an instance of a class or deriving all individuals for a particular class, and *Subsumption*, i.e., deciding whether a class is subsumed by another class (is a subclass of). *Equivalence* inferences which check if two classes or individuals are equal (or disjoint) and hence can be unified is another important task which is provided by expressive DL ontology languages, e.g., OWL-DL (*SHOIN(D)*). The typed unification rules take into account, that the type system is defined by one or more DL ontologies and that subtype tests by subsumption are constrained by the expressiveness of the DL query language. For instance, in OWL there is no way to express the concept of a most general superclass or a most specific type. Although, it is possible to compute such statements by applying iterative subsumption queries, such an approach will impose greater computational burdens for dynamic type checking. Therefore, in ContractLog I restrict type checking to finding the lower bound of two types (T_1, T_2) under $\leq$ in the *lower* operation of the typed unification and replace the type of a term with the more specific type concept in the unification rules. The operation *lower* requires at most two subsumption queries to the external DL reasoner to compute the lower bound under the partial order $\leq$. If the type system consists of more than one DL ontology, the ontologies are merged into a combined ontology. The common super class under which all ontologies are subsumed is the concept "Resource". Hence, the partial order $\leq$ also holds for the combined ontology with $owlResource \cong \top$ under the assumption that no cycles are introduced. Cross links between the component ontologies might be defined, e.g., by relating classes with $owl : equivalentClass$ or $owl : disjointWith$. Note that this may introduce conflicts between terminological definitions which need conflict resolution strategies, e.g., based on defeasible reasoning.

Remark 2 *Note the difference between my order-sorted typing approach and hybrid integration approaches which apply additional DL atoms resp. DL constraints to query the DL ontology component such as e.g., dl-programs [ELST04] or Carin [LR96]. In my prescriptive approach the type checks in terms of DL-queries to the DL component apply during the typed unification process and constrain the unification of terms. The operational semantics provides a "built-in" technical separation between the rule inferences and the DL inferences which directly applies during typed term unification and results in flexible formalisms that are robustly decidable, even in the case where the rule language is far more expressive than Datalog. This particular combination cannot be seen neither as a super- or subset of homogeneous approaches such as SWRL nor as related to existing hybrid approaches which apply DL constraints resp. DL query atoms in the body of rules, since the semantics is completely different. In ContractLog it is based on a prescriptive typing approach with a order-sorted typed unification as operational semantics.*

4.2.5 Summary

The main motivation for introducing Semantic Web or Java based types into declarative logic programs comes from Software Engineering, where principles such as data abstraction, modularization and consistency checks are vital for the development and maintenance of large rule bases. Distributed system engineering and collaboration, where domain-independent rules need to be interchanged and given a domain-dependent meaning based upon one or more common vocabularies in their target environments is an other matter. The possibility to use arbitrary Semantic Web ontologies and object-oriented domain models in declarative logic programming gives a highly flexible, extensible and syntactically rich language design with a precise semantics. Semantic Web counterparts of common syntactic specification languages for SLAs and policies such as WS-Policy, WSLA or WS-Agreement and other domain-specific ontologies, e.g., OWL Time, can be easily used as type systems giving the rule terms a domain-specific meaning. This syntactic and semantics combination which allows efficient declarative programming, in contrast to simple syntactic specification of agreements, is vital for the automated discovery, management and monitoring/enforcement of electronic contracts/agreements/policies for (Semantic Web) services. From a computational point of view, the use of order-sorted types can drastically reduce the search space, hence increasing the runtime efficiency and the expressive power, e.g., enabling overloading of rule variants. The tight combination of declarative and object-oriented programming by rich Java-based procedural attachments facilitates integration of existing procedural functionalities, tools and external data sources into rule executions at runtime. In this section I have presented a hybrid approach which provides a technical separation with minimal interfaces between the rule and type components leading to a robust, flexible and expressive typed logic with support for external Java and DL type systems, Java-based procedural attachments, modes and built-ins. The implementation follows a prescriptive typing approach and incorporates type information directly into the names of symbols in the rule language. The interaction between the rules and the type system is based on entailment in a multi-sorted logic. As an operational semantics a typed unification is applied which permits dynamic type checking, ad-hoc polymorphism, i.e., variables might change their types (coercion), and overloading, i.e., overloading of rules by using different types in their rule heads, leading to variants of the same rule.

In summary, the main contributions of the typed logic extension are:

- a compact, flexible prescriptive typed rule language with typed terms
- flexible, hybrid integration of external vocabularies as order-sorted type systems
- integration of external systems (by Java API) and object-oriented functionalities by highly expressive procedural attachments and built-ins
- integration of external reasoners and procedural functionalities for dynamic type checking by an ad-hoc polymorphic order-sorted typed unifi-

cation

4.3 Meta-data Annotated Labelled Logic

As discussed in section 2.3 typical electronic contracts consist of a hierarchy of possibly distributed, interlinked and collaboratively engineered and maintained subcontracts reaching from general terms and conditions to customer specific SLAs and internal Operational Level Agreements (OLAs) and Underpinning Contracts (UCs). Moreover, information from various independent data sources, e.g., data bases, data ware houses, Semantic Web ontologies, system management and network management tools, are dynamically integrated as extensional knowledge (fact base) into the contract rule executions at runtime.

To capture this distributed, Web-based, open structure, enable *scoped queries* on explicitly closed parts of the formalized open and distributed contract knowledge, and support principles of information hiding and modularization I have extended the ContractLog KR to a general *meta-data annotated labelled logic* with *scoped reasoning*. Meta-data such as rule labels, module labels or Dublin Core annotations (e.g., author, date etc.) can be attached to rules and facts. These additional meta-data annotations become in particular interesting when the knowledge base consists of several (possibly distributed) rule sets, so called modules, which might be collaboratively engineered, interchanged and dynamically imported from different external sources accessible by their Web-based URIs. The meta-data might be used to infer additional knowledge, e.g., for automated discovery and matchmaking, the dynamic management of the distributed KB with support for dynamic updates/imports by ID-based labels (see section 4.5) or *scoped reasoning* on explicitly closed parts of the KB by scoped queries, e.g., "all rules from a particular author" or "all facts with time stamps after a certain date/time". In an open contract environment provided on the (Semantic) Web where the knowledge is inherently incomplete the meta data annotations can be used to restrict problematic non-monotonic features such as negation-as-failure, which amounts for a CWA, to an explicitly closed scope of the open KB. As a result scoped goals are only evaluated wrt an explicitly closed scope which ensures complete information and hence provides sound answers for the scoped queries. Another important application domain is rule interchange. Additional meta data about the semantics of a program, module or single rule can be given, which can be used in the target environment to decide whether the execution will be correct. Different optimized variants might be implemented for well-known semantics and the meta data might be used to select the correct variant wrt the semantics of the target execution environment. Moreover, "scoping" leads to much smaller search spaces and allows an explicit management of the level of generality of queries/goals. In the following I introduce an extension of the ContractLog/Prova rule language with meta-data annotation labels and discuss the semantics of this new meta-data annotated, labelled logic for distributed logic programs.

4.3.1 Syntax of Meta-Data Annotated ContractLog LPs

In analogy to the multi-sorted extension of the last section, the meta-data extension of the ContractLog language is defined over a combined signature $\overline{S}$ which is the union of the signature of the rule component and the signatures of the used meta data vocabularies (e.g. Dublin Core).

Definition 110 ***(Combined Signature with Meta-data Annotations)*** *The combined meta data annotated signature $\overline{S}$ is defined as a tuple $\langle \overline{T}, \overline{P}, \overline{F}, arity, \overline{c}, sort \rangle$ where $\overline{P}$ is the union of the predicate symbols define in the signature of the core rule language and the meta data predicate symbols (names) defined in the signature(s) of the meta data vocabularie(s) and $\overline{c}$ is the union of constant symbols defined in the rule signature and in the meta data signature(s) (values).*

In order to keep the complexity of meta data reasoning at a minimum, the arity of meta data predicates is restricted to unary predicates, i.e., to typical meta data property-value pairs $property(value)$ such as $dc\_creator(\text{"}AdrianPaschke\text{"})$ (although the underlying mechanisms could handle more expressive meta data annotations with n-ary relations). Moreover, variables are not allowed in the meta data labels, i.e., the meta data atoms must be ground. To explicitly annotate clauses in a labelled logic program (LLP) P with an additional set of meta-data labels I introduce a general n-ary *metadata* function into the ContractLog language. The function *metadata* is a partial injective labelling function that assigns a set of meta data annotations m (property-value pairs) to a clause cl in P, i.e., $m : cl$. It is syntactically defined separated from a clause (rule/fact/query) by "::":

$$metadata(L_1, .., L_n) :: H \leftarrow B$$

where L_i are a finite set of unary positive literals (positive meta data literals) which denote an arbitrary meta data $property(value)$ pair, e.g., $label(rule1)$. That is, I write in implicit form $metadata(L_1, .., L_n) :: H \leftarrow B$ to express that $metadata(H \leftarrow B) = L_1, .., L_n$. The explicit $metadata()$ annotation is optional, i.e., a program P without meta data annotated clauses coincides with a standard unlabelled LP. Akin to object oriented programming (OOP), clauses in ContractLog are treated as objects having an unique *object id* (*oid*) which might be user-defined, i.e., explicitly defined by a label in the meta data annotations $metadata(label(< oid >), ...) :: H \leftarrow B$ or system-defined i.e., all rules are automatically "labelled" with an auto-incremented *oid* (an increasing natural number) provided by the system at compile time. Rules and facts might be bundled to clause sets, so called *modules*, which also have an object id, the module oid. By default the module oid is the URI of the ContractLog script which defines the module. But the module oid might also be user-defined (see update functions in section 4.5). The oids are used to manage the knowledge in the (distributed) knowledge base, e.g., to import a rule set from an URI which is then used as the module oid or remove a module from the KB by its oid. Beside oids arbitrary other semantic annotations such as Dublin Core data might be specified in the *metadata* label.

Example 12

```
metadata(label(r1), dc:author("provider"), dc:date(2006-11-12))::
    p(X):-q(X).
metadata(label(f1))::q(1).
```

The example shows a rule with rule label $r1$ and two additional Dublin Core annotations $dc : author(\text{"}provider1\text{"})$ and $dc : date(2006 - 11 - 12)$ and a fact with fact label $f1$. Since there is no explicitly user-defined module oid in the meta-data labels, the default module oid for both clauses is the URI of the script in which they are defined.

The meta annotation of rules and rule sets (modules) enables (meta) reasoning with the semantic annotations. It forms the basis for many expressive functionalities of the ContractLog KR such as superiority relations between rules and modules or oid based knowledge updates. Moreover, the meta data can act as an explicit scope for constructive queries (creating a view) on the knowledge base. For instance, the meta data annotations might be used to constrain the level of generality of a scoped goal to a particular module, i.e., to consider only the set of rules and facts which belong to the specified module.

Definition 111 ***(Scoped Literal)*** *A scoped literal is of the form $L : \overline{C}$ where L is a positive or negative atom and $\overline{C}$ is the scope definition which is a set of one or more meta data constraints.*

Scoped literals are only allowed in the body of a rule. Scoped literals might be default negated $\sim L : \overline{C}$. Syntactically, the following built-in predicates are used in ContractLog to query the meta data annotations and define the scope of literals for metadata-based *scoped reasoning* on explicitly specified parts of the KB:

```
% scoped literal
partial(<literal>,<meta data value>)
% query meta data value
metadata(<literal>,<Variable>,<meta data property>)
% constrain scoped goal literal
metadata(<literal>,<meta data value>,<meta data property>)
```

Example 13

```
metadata(label(rule1), src("http://rbsla.de/module1"))::
        p1(X):-partial(q(X),http://rbsla.de/module1).
metadata(label(rule2), src(http://rbsla.de/module2))::
        p2(X):-partial(q(X),http://rbsla.de/module2).
metadata(label(fact1), src(http://rbsla.de/module1))::
```

```
        q(1).
metadata(label(fact2), src(http://rbsla.de/module2))::
        q(2).

:-solve(partial(p1(X),http://rbsla.de/module1).
:-solve(partial(p2(X),http://rbsla.de/module2)).
:-solve(metadata(p1(X),RuleOID,label).
:-solve(metadata(p2(X),RuleOID,label).
```

The example shows *scoped reasoning* on explicitly closed parts of the KB. The first query has the user-defined scope $http://rbsla.de/module1$. The URI is the object id (oid) which is used as module label. Accordingly, the answer here is $X = 1$. The second query has the scope $http://rbsla.de/module2$ and accordingly the answer is $X = 2$. The third and fourth queries return the rule oid of both rules which is bound to the variable $RuleOID$, i.e., $RuleOID = rule1$ and $RuleOID = rule2$, i.e., they define queries on the set of meta data annotations.

Example 14

```
metadata(label(loop),semantics(WFS))::a():-b(),a().
```

The example shows the annotation of a rule with additional meta data about its intended semantics. This additional information can be used to avoid unintended and incorrect executions. For instance, standard SLDNF resolution would run into a loop in the above example.

4.3.2 Semantics of Meta-Data Annotated Labelled Logic Programs with Scoped Reasoning

In the context of an open environment such as the Semantic Web with distributed data sources and distributed programs (LP scripts) which are dynamically imported as modules, the KB is typically not closed. As a result, queries and intermediate goals inherently depend on incomplete information. For instance, new knowledge is discovered, programs/modules change or are no longer accessible due to lost Internet connection. Hence, it is impossible to find sound answers to default negated queries due to the inherent non-monotonicity of default negation where soundness depends on the assumption of a closed world and complete information. To overcome these problems scoped literals are used in ContractLog which apply only in their defined scope. Accordingly, if a default negated query/goal depends only on scoped literals the KB is explicitly closed for this query and newly added knowledge does not affect already drawn answers. In the following I define the semantics of *meta-data annotated labelled logic programs (LLPs)* in ContractLog and discuss how the concept of scoped

reasoning and explicit (partial) closure of an open knowledge base helps to preserve well-known abstract properties of general non-monotonic inference such as *rationality*, *cut* and *cautious monotonicity* [KLM90].

Informally, scoped literals allow to explicitly close the domain of discourse to certain parts of the KB.

Definition 112 *(**Metadata-based Scope**) Let $\overline{KB}$ be a combined KB consisting of a set of submodules $\overline{KB} = \{KB_1 \cup .. \cup KB_k\}$. The scope KB' of a scoped literal $L : \overline{C}$ is the set of clauses $KB' = \{m'_1 : cl_1, .., m'_n : cl_n\} \in \overline{KB}$, i.e., $KB' = \{cl(m'_1), .., cl(m'_n)\}$ where for all clauses $cl_i(m'_i) \in \overline{KB'}$ its set of meta data annotations m'_i satisfy the meta data constraints $\overline{C}$ of the scoped literal L, i.e., $m'_i \models \overline{C}$.*

Accordingly, a scope (aka view) is constructed by one or more meta data constraints, e.g., a module oid $src(< URI >)$ or Dublin Core values $dc\_author(...)$, $dc\_date(...)$.

Definition 113 *(**Closure**) Let $\overline{KB}$ be a combined KB. The closure of $\overline{KB}$, denoted $Cl(\overline{KB})$, is defined by $\overline{KB}$ plus all KBs which are in the scope of any literal in $\overline{KB}$.*

A scoped literal $L : \overline{C}$ is closed if each rule in $\overline{KB}$ which unifies with the literal L is also closed, i.e., its body literals are closed in $Cl(\overline{KB})$.

Intuitively, this means that the closure of a program depends on the scopes of the literals in the bodies of its rules. Obviously, if one of the subsequently used goal literals in a proof attempt is open, i.e., without a scope, the closure expands to the open KB.

Without being specific to a particular semantics, since ContractLog is basically intended to be interpretable with different semantics, the semantics for LLPs is defined as follows:

Definition 114 *(**Scoped Semantics**) Given any (sceptical) semantics SEM^{scept} and a scoped KB $\overline{KB}$, where all default negated literals are scoped and hence the closure is $Cl(\overline{KB})$, the truth value of a scoped literal $L : \overline{C}$ wrt $SEM^{scept}(\overline{KB})$, i.e., $SEM^{scept}(\overline{KB}) \models L$, depends on the partial interpretation formed from the clauses of $\overline{KB}$ wrt the scope definition $\overline{C}$, i.e., $SEM(partial(\overline{KB}, \overline{C})) \models L$.*

A simple transformation of all default negated atoms into scoped atoms using the parent module oid as scope maps a program into a scoped program.

I now show that a scope program with meta data-based scopes, in particular scopes defined by modules, can be used to preserve typical structural properties of general non-monotonic inference such as *rationality*, *cut* and *cautious monotonicity* [KLM90]. Kraus et al. [KLM90] proposed several structural properties to compare non-monotonic theories. The following structural properties are from [KLM90]:

- *Right Weak*: $\alpha \rightarrow \beta$ and $\gamma \vdash \alpha \Rightarrow \gamma \vdash \alpha$, where α,β,γ are sets of atoms.
- *Reflexivity*: $\alpha \vdash \alpha$
- *And*: $\alpha \vdash \beta$ and $\alpha \vdash \gamma \Rightarrow \alpha \vdash \beta \wedge \gamma$
- *Or*: $\alpha \vdash \gamma$ and $\beta \vdash \gamma \Rightarrow \alpha \vee \beta \vdash \gamma$
- *Left Log. Equiv.*: $\models \alpha \leftrightarrow \beta$ and $\alpha \vdash \gamma \Rightarrow \beta \vdash \gamma$
- *Cautious Monotony*: $\alpha \vdash \beta$ and $\alpha \vdash \gamma \Rightarrow \alpha \wedge \beta \vdash \gamma$
- *Cut*: $\alpha \vdash \beta$ and $\alpha \wedge \beta \vdash \gamma \Rightarrow \alpha \vdash \gamma$
- *Rationality*: *not* $\alpha \vdash \neg\beta$ and $\alpha \vdash \gamma \Rightarrow \alpha \wedge \beta \vdash \gamma$
- *Negation Rat.*: $\alpha \vdash \beta \Rightarrow \alpha \wedge \gamma \vdash \beta$ *or* $\alpha \wedge \neg\gamma \vdash \beta$
- *Disj. Rat.*: $\alpha \vee \beta \vdash \gamma \Rightarrow \alpha \vdash \gamma$ *or* $\beta \vdash \gamma$

In contrast, to the general entailment relation of Kraus et al. [KLM90] the sceptical entailment relation $\vdash$ used in ContractLog is defined between possible infinite sets of atoms and literals $\vdash \subseteq 2^{Atoms} \times 2^{Literals}$ and is not defined on the whole set of formulae as in the general setting of non-monotonic reasoning. Hence, *Right Weak*, *Reflexivity*, *And* and *Left Logical Equivalence* are always satisfied in any sceptical semantics such as WFS and the interesting properties in the context of distributed LPs with scoped non-monotonic reasoning in terms of default negation are *Cautious Monotony* and *Cut* which are combined to *Cumulative Monotony*: "If $\alpha \vdash \beta$ then: $\alpha \vdash \gamma$ iff $\alpha \wedge \beta \vdash \gamma$" and *Rationality*: "*not* $\alpha \vdash \neg\beta$ and $\alpha \vdash \gamma \Rightarrow \alpha \wedge \beta \vdash \gamma$". Cumulative Monotony states that a formula β derived from α can be used as a lemma and does not affect the set of formulae derivable from α alone. This condition in the context of distributed KBs and knowledge updates is important since intermediate lemmas are of no use if it does not hold, i.e., answers to queries and computed subgoals possibly need to be retracted if new information is added in the form of additional modules (rule/fact sets). *Rationality* is in any sceptical semantics a stronger form of *Cautious Monotony* since $\alpha \vdash \beta \Rightarrow$ *not* $\alpha \vdash \neg\beta$.

- *Cumulative Monotony*: If $\alpha \subseteq \beta \subseteq SEM^{scept}_{\overline{KB}}(\alpha)$, then $SEM^{scept}_{\overline{KB}}(\alpha) = SEM^{scept}_{\overline{KB}}(\beta)$, where α and β are sets of atoms and $SEM^{scept}_{\overline{KB}}$ is an arbitrary sceptical semantics for the program $\overline{KB}$ which is a set of program modules $\overline{KB} = KB_1 \cup .. \cup KB_k$.
- *Rationality*: If $\alpha \subseteq \beta, \beta \cap \{A : SEM^{scept}_{P}(\alpha) \models \neg A\} = \emptyset$, then $SEM^{scept}_{\overline{KB}}(\alpha) \subseteq SEM^{scept}_{\overline{KB}}(\beta)$

The following example shows that in the un-scoped case of modular knowledge bases consisting of imported distributed LPs or dynamically added modules (see section 4.5 for dynamic updates) cumulative monotony and rationality for typical LP semantics do not hold. For the reason of simplicity the example is given as a propositional LP, but it turns out that similar results are achieved for other logic classes such as normal or extended LPs.

Example 15

```
http://rbsla.de/LP1:    p
                        b <- not a, p

http://rbsla.de/LP2:    p <- a
                        a <- not b
```

The example defines two programs provided at the URLs $http://rbsla.de/LP1$ and $http://rbsla.de/LP2$ (the URLs are used by default as the oids of the programs), which I abbreviate as $LP1$ and $LP2$ in the following. The first program derives the answers p and b and the second program derives p and a under typical LP semantics such as $COMP$ (with SLDNF resolution and negation-as-finite-failure rule), $STABLE$ or WFS. However, if both programs are combined in a (distributed) knowledge base $\overline{KB} = LP1 \cup LP2$, e.g., by importing $LP1$ to $LP2$ or vice versa, these conclusions for the combined KB are no longer valid. Since the answers of the two-valued $COMP$ agrees with the mechanisms of SLDNF "$a :- not\ b.\ b :- not\ a, p.$" presents a loop (see e.g. [She91]) and the completion of the combined LP is incomplete. $STABLE$, due to the fact "p" and the rule "$b :- not\ a, p.$", does no longer derive a and WFS neither derives b nor a for the combined KB. Accordingly, as can be seen from this example, although all three semantics fulfil cut, cumulative monotony and rationality are not fulfilled. I now refine the example and explicitly close the negated atoms with a scope which applies on the module in which they occur. For the reason of readability, I use $L : \overline{C}$ instead of the built-in $partial(L, \overline{C})$ to denote that the scoped literal L has scope $\overline{C}$.

```
http://rbsla.de/LP1:    p
                        b <- not a:LP1, p

http://rbsla.de/LP2:    p <- a
                        a <- not b:LP2
```

Now, due to the scoped negation as failure ($not\ a : LP1$ and $not\ b : LP2$), SLDNF does not run into a loop for the combined KB and $STABLE$ implies b, p and a as well as WFS, i.e., cumulative monotony is preserved in all semantics.

The operational semantics for scopes in the ContractLog reference implementation is based on the treatment of clauses and clause sets as objects, i.e., clauses (facts, rules) and clause sets (modules), as in object-oriented programming, are managed as objects in the KB, each having an unique object id and a set of meta data properties. The meta data are used to preselect the objects (clauses) from the knowledge base , akin to a constructive view as in relational database systems where a SQL "where" clause can be used to constrain the set of selected data.

4.3.3 Summary and Related Work

The usual close world assumption [Rei82] of logic programming, stating that an atom that does not appear in the KB is assumed to be false, is only applicable when the KB is closed and contains the complete knowledge of the domain of discourse. This assumption is inherently inappropriate in the SLA domain since here the rule bases and data sources are typically scattered in an open domain such as the Semantic Web, where new knowledge might become available or unavailable.

To overcome this closed world problem in open domains an open world assumption (OWA) has been proposed in some works, where a KB is interpreted as the conjunction of all literals in the KB, e.g., in [MLF02]. In [AADW05] a combination of open-worlds and closed-world reasoning based on partial logic [HJW99] has been proposed.

Local closed world assumption (LCWA) has been proposed as another possible solution, e.g., in [EGW97, DLS00, CCDA$^+$05, HA02]. The approach of LCWA is to syntactically state that a goal G is locally closed wrt to a knowledge base, i.e., state that every ground clause that unifies with G either follows from a partial KB which is closed wrt G or does not follow from KB: $LCWA(G) = (KB \models G\theta) \vee (KB \models \neg G\theta)$ for all ground substitutions θ. Circumscription [McC80] has been propose as a possible selection function [DLS00] which selects those models which satisfy the locally closed KB and that are minimal wrt the goal G for which KB is complete.

A third way to distinguish different levels of knowledge which has been used by several approaches is to label the predicates with additional information about its class, e.g "sound", "complete", "exact" as proposed e.g., in [Gra02, CDGL02] or "objective", "open", "closed" as proposed by [DAAW06].

Recently, scoped negation-as-failure [KBBF05] has been termed in the Semantic Web for non-monotonic negation which applies in the context of a module, where the module identifier, which is typically an URI, is used to define the closed scope of a (negated) query. Flora-2 [YKZ03], which exploits F-Logic [KLW95], a frame-based representation of FOL used for ontology reasoning, and TRIPLE [SD02], a LP engine for RDF reasoning, are two systems which support the concept of a module and allow a form of module-base scoped negation as failure.

My approach in ContractLog generalizes the idea of scoped reasoning towards meta-data annotated Labelled Logic Programs where program clause might be labelled by arbitrary meta data sets over which arbitrary scopes (views) can be constructed. That is, the meta data annotations of rules and facts are used to explicitly define scopes in which the (negated) knowledge is complete and cumulative monotony and rationality is preserved (under WFS). Accordingly, already given answers from a closed scope are never forced to be retracted if new knowledge becomes available in the open environment. To some extend this approach is comparable to constructive views in relational databases where a

close world assumption is made for the set difference operation of the relational algebra.

From a computational point of view my constructive, meta-data based scoping approach significantly decreases complexity of computing semantics, since it allows to reduce the open distributed KB to explicitly closed subparts with a smaller Herbrand base. For instance, a scope constructed by a fact set which is provided at a certain URL $src(http://...)$ or facts which are not older than a certain date $dc\_date(...)$.

In distributed SLA management the additionally provided meta data annotations are used for the dynamic management of contract structures and contract specifications enabling flexible oid-based updates (see section 4.5). In particular, modules which bundle rule and/or fact sets, facilitates large rule bases to be put together from components which can be developed, compiled, tested and managed separately. This modularization enforces typical SE principles such as the principle of information hiding and data abstraction. It is vital in rule-based contract representation, where Service Level Agreements (SLAs) are typically defined as distributed, hierarchical structures with e.g., general terms and conditions, master agreements, several SLAs and underpinning contracts or operational agreements (see section 2.3).

In the context of rule interchange additional meta data about the intended semantics of the target execution environment can be given, possibly permitting semantics variants of rules and rule sets. Moreover, in the negotiation and discovery phase of agreements the meta data can be used to give additional information enabling more exact searches with higher precision of the answers.

In summary, the meta data labelled logic extension of ContractLog

- is intuitive to use, flexible and extensible in the way of using meta data vocabularies
- can be implemented with the existing technologies of standard LP inference engines and Semantic Web technologies
- addresses the need of distributed contract management in open environments such as the Semantic Web

The utilization is many-fold reaching from modularized engineering, meta-data enriched discovery, efficient scoped querying with scoped closed world assumption, rule interchange to dynamic management and maintenance with oid-based knowledge updates.

4.4 Integrity Preserving, Preferenced, Defeasible Logic

Rules in SLAs might overlap and contradict each other, in particular if contracts grow larger and more complex and are authored, maintained and updated by different people. A typical conflict which might arise in the context of extended

LPs, as used in ContractLog, are contradictions between simultaneously concluded positive and negative information, i.e., $KB \vdash A$ and $KB \vdash \neg A$, an atom and its negation (its complement) follow from the KB. However, as discussed in section 4.1.2, from an application point of view conflicts might also occur between arbitrary conclusion, not just between positive and negative information.

Example 16

```
r1 :: discount(Customer, 10) :-
        spending(Customer, Value, last year), Value > 1000.
r2 :: discount(Customer, 5) :-
        spending(Customer, Value, last year), Value > 500.
```

In the example, a customer might apply for a discount of 10% as well as a discount of 5%, if the spending of the last year is more than 1000. From an applications point of view only the higher discount should be drawn and hence both conclusions are in a conflict, if they can be concluded simultaneously. Moreover, in many rule-based SLA specifications certain parts of the rule based decision logic should be static and not subjected to changes at run time or it must be assured that updates do not change the intended behavior and do not lead to anomalies or conflicts. A common way to represent such constraints and define the conditions of conflicts in dynamic systems are integrity constraints (IC). ICs are a way to formulate consistency (or inconsistency) criteria and they are used to verify the consistency of a dynamic KB which is updated during runtime. In case these integrity constraints are violated, default reasoning formalisms and preferences between rules can be used to resolve the conflicts. In this section I will further elaborate on the defeasible logic based approach for conflict resolution in the ContractLog KR (see also section 4.1.2). I first review the history and basic concepts of defeasible reasoning. I then introduce ICs into ContractLog. Based on this extension and the labelled logic introduced in the previous section, I refine the syntax and semantics of prioritized defeasible reasoning in ContractLog.

4.4.1 Basic Concepts and History of Defeasible Logics

The roots of defeasible logic (DefL) lie in inheritance networks. [HTT87] Based on these concept, Nute's defeasible logic [Nut94] has been designed as a nonmonotonic reasoning formalism for logic programming using defaults, called defeasible rules which allow defeasible reasoning, where the conclusion of a rule might be overturned by the effect of another rule with higher priority. That is, it seeks to resolve conflicts by "defeating" and by explicitly expressed superiority relations between rules. Defeasible logic differs between strict rules and defeasible rules (since defeaters, which prevent conclusion, are not used in ContractLog I omit them here):

- Facts are strict ("standard" facts), i.e., indisputable

- Strict rules: "Standard" rules (derivation rules) with rule label: $r :: H \leftarrow B$
- Defeasible rules are rules that can be defeated by contrary rules: $r :: H \Leftarrow B$
- Priority relations are used to define priorities among rules to represent that one rule may override the conclusion of another (defeasible) rule: $r1 > r2$

DefLs have been studied in terms of proof theory [ABGM01], denotational semantics [Mah00], model-theoretic semantics [Mah02] and computational complexity [Mah01]. In contrast to other non-monotonic approaches such as Reiters default logic [Rei80] which is based on a possible worlds semantics, defeasible logic adopts a sceptical reasoning approach to default reasoning in the sense that conflicts are resolved by explicit superiority relations and conflicting rules do not fire. The main advantage of DefL, compared to other nonmontonic reasoning approaches such as circumscription [McC80], default logic [Rei80] or autoepistemic logic [Moo85], are its low computational complexity [Mah01] and the enhanced expressiveness allowing reasoning with incomplete and contradictionary information in standard LP interpreters. Different DefL variants have been proposed, reaching from simple defeasible implementations that deal with conflicts between positive and negative conclusions [Nut94, ABGM01] with ambiguity blocking or ambiguity propagation to generalized courteous logic programs (GCLP) [Gro99] which use an additional "mutex" (from mutual exclusive) to define and handle arbitrary mutual exclusive literals. Several meta programming approaches have been proposed to execute a defeasible theory in a logic program without [ABGM00, AM02] and with negation-as-failure [AMB00, ABGM01, MG99]. In ContractLog I follow the meta programming approach and generalize it with the concept of (scoped) default negation, integrity constraints and oid-based labelled logic in order to define arbitrary conflicts in terms of integrity rules and priorities between labelled rules and complete rule sets for conflict resolution.

4.4.2 Integrity Constraints

ICs are a way to formulate consistency (or inconsistency) criteria of a rule based system and they can be used to verify the consistency of a dynamic KB which is updated during runtime (see section 4.5).

Definition 115 ***(Integrity Constraints)***

- *An integrity constraint on a LP is defined as a set of conditions that the constrained KB must always satisfy, in order to be considered as a consistent model of the intended model.*
- Satisfaction *of an integrity constraint is the fulfillment to the conditions imposed by the constraint*
- Violation *of an integrity constraint is the fact of not giving strict fulfillment to the conditions imposed by the constraint*

- *Satisfaction resp. violation of a program P wrt the set of integrity constraints $\overline{IC} := \{IC_1, .., IC_n\}$ defined in P is the satisfaction of each $IC_i \in \overline{IC}$ wrt to the actual knowledge state of P.*

Note, that integrity constraints are closely related to the notion of test cases for LPs in the ContractLog KR which will be discussed in section 4.9. In fact, test cases can be seen as more expressive integrity constraints.

In the context of knowledge updates, which I will describe in the section 4.5 conflicts might naturally arise, in particular when the rule program grows larger and updates are done by different people/systems. Let me first define what I mean by a conflict:

Definition 116 ***(Conflict)***

- *direct logical conflict: a rule $r1$ and a rule $r2$ are conflicting iff the head of the head of the rule with label $r1$ is the complement of the head of the rule $r2$ and both can be derived simultaneously*
- *indirect logical conflict: a rule $r1$ and a rules $r2$ are conflicting with respect to a set of rules $\overline{R}$ iff the application of one rule defeats the other or vice versa*
- *domain-specific conflict: a rule $r1$ and a rule $r2$ are conflicting with respect to a particular application domain, if both apply in the same situation/-context*

While the first two conflicts denote a conflict from a logical point of view, the last type of conflict is application specific. For example, two rules defining either a discount of 5% or are discount of 10% might be conflicting if both discounts can be derived at the same time. The first type of conflict arises if a positive and a negative atom (explicitly negated) can be derived at the same time. To describe the second type of conflict consider the following program:

Example 17

```
a <- not b
b <- not a
```

Here the heads of both rules are not complementary, but the both rules defeat each other (using e.g., WFS) due to the negation. Such conflicts are much harder to detect and typically occur indirectly, based on the existence of other rules. Integrity constraints can be used to define conditions which denote a logic or application specific conflict.

4.4.2.1 Syntax of Integrity Constraints

Integrity constraints in ContractLog are defined by a restricted n-ary function symbol *integrity* in the rule language: $integrity(< operator >, < conditions >$). In ContractLog I distinguish four types of integrity constraints:

- *Not-constraints* which express that none (akin to strong negation) of the stated conclusions should be drawn: $integrity(not(L_1, .., L_n)$.
- *Xor-constraints* which express that the stated conclusions are mutual exclusive, i.e., should not be drawn at the same time: $integrity(xor(L_1, .., L_n))$.
- *Or-constraints* which express that at least one of the stated conclusions must be drawn: $integrity(or(L_1, .., L_n))$.
- *And-constraints* which express that all of the stated conclusion must drawn: $integrity(and(L_1, .., L_n))$.

An IC might be also expressed conditionally as a *integrity rule*, e.g.,

$integrity(xor(p(), q())) : -a(), b()$.

That is, the integrity rule states that $p()$ and $q()$ are mutual exclusive, denoting a conflict if both can be derived at the same time. The integrity constraint applies if the body literals (conditions) $a()$ and $b()$ hold.

Example 18

```
integrity(xor(discount(Customer,5),discount(Customer,10))).
integrity(not(discount(Customer,100))).
integrity(and(discount(Customer,Discount),price(Customer,Price))).
```

The example defines three integrity constraints. The first constraint states that a customer might be given a discount of 5 percent or a discount of 10 percent, but not both at the same time. The second constraints states that a customer never gets a discount of 100 percent and the last constraint states that all customers who get a discount must pay a certain base price (on which the discount applies).

4.4.2.2 Semantics of Integrity Constraints

The semantics for integrity constraints is directly inherited from the semantics of logic programs. Integrity constraints are meta level statements which are defined as constraints on the set of possible models and describe the model(s) which should be considered as conflicting. Roughly speaking, the set of constraints defined in an integrity constraint are interpreted as a top query, i.e., as a rule without a head, whereas the operator function *and*, *or*, *xor not* defines the quantification and connective of the subgoals (the constraints) as follows:

Definition 117 ***(Integrity Validation Semantic)*** *Let KB_l be the actual knowledge base (the actual program state) and IC_m be an integrity constraint defining a set of constraints $\overline{C} = \{C_1, .., C_n\}$ connected by an operator and, or, xor or not. Then IC_m is satisfied if $KB_l \models IC_m$ which is defined for the four types of operators as follows:*

- $IC_m = and(C_1, .., C_n)$*:* $KB_l \models IC_m$ *if* $\forall\ i \in 1, .., n : KB_l \models C_i$
- $IC_m = not(C_1, .., C_n)$*:* $KB_l \models IC_m$ *if* $\nexists\ i \in 1, .., n : KB_l \models C_i$
- $IC_m = or(C_1, .., C_n)$*:* $KB_l \models IC_m$ *if* $\exists\ i \in 1, .., n : KB_l \models C_i$
- $IC_m = xor(C_1, .., C_n)$*:* $KB_l \models IC_m$ *if* $\exists\ j \in 1, .., n : KB_l \models C_j$ *and* $\nexists\ k \in 1, .., n : KB_l \models C_k$ *with* $C_j \neq C_k$ *and* $C_j \in C$, $C_k \in C$,

where a constraint literal C_i is a positive or negative (explicit negated) n-ary atom which might contain variables; $\sim$ is used in the usual sense of default negation, i.e., if a constraint literal can not be proven true, it is assumed to be false.

That is, the knowledge base KB entails an integrity constraint IC_m under WFS, denoted by $KB_l \models IC_m$, if IC_m is true in $SEM_{WFS}(KB_l)$.

Provability of an integrity constraint IC in KB_l, denoted $KB_l \vdash IC$ is defined as a standard LP deduction. That is to prove $IC = \{C_1, .., C_n\}$ in KB_l proofs for the constituent constraints must be established by standard deduction wrt the integrity operators:

- and: $KB_l \vdash IC$ if $\forall\ i \in 1, .., n : KB_l \vdash C_i$
- not: $KB_l \vdash IC$ if $\nexists\ i \in 1, .., n : KB_l \vdash C_i$
- or: $KB_l \vdash IC$ if $\exists\ i \in 1, .., n, KB_l \vdash C_i$
- xor: $KB_l \vdash IC$ if $\exists\ j \in 1, .., n : KB_l \vdash C_j$ and $\nexists\ k \in 1, .., n : KB_l \vdash C_k$ with $C_j \neq C_k$ and $C_j, C_k \in \overline{C}$

To proof integrity constraints according to the semantics defined above I have implemented a LP meta program in the ContractLog KR which meta interprets the defined integrity constraints. The meta program implements two main test axioms:

- *testIntegrity*() tests the integrity of the actual program state, i.e., it proves all integrity constrains in the knowledge base using them as goals constraining on the facts and rules in the KB.
- *testIntegrity*(*Literal*) tests the integrity of the literal, i.e., it makes a hypothetical test and proves if the stated literal, which is actually not in the KB, would violate any integrity constraint in the KB.

The first integrity test is useful to verify and validate the integrity of the actual knowledge state wrt to the integrity constraints defined in the program. The

second integrity test is useful to hypothetically test an intended knowledge base update, e.g., test whether the literal, which might be the head of a rule which should be added, would violate the integrity of the program. If the hypothetical test succeeds, i.e., all integrity constraint are satisfied wrt to the new literal, the rule can be safely added.

Example 19

```
neg(discount("Adrian",X)).  %fact
integrity(xor(discount(Customer,X), neg(discount(Customer,X))).
testIntegrity(discount("Adrian",X))?  %query test integrity
```

The example defines a conflict between a granted discount and no discount. A hypothetical test for giving a discount to the customer *Adrian* is made, which would violate integrity, since the fact defines that *Adrian* gets no discount.

For the reason of space, I only show the meta rule specifying the structure of integrity validation for the xor operator in the hypothetical validation:

```
% test XOR (mutex) integrity constraints
testIntegrity(Literal):-
    integrity([xor|Mutex]),
    delete(Literal,Mutex,NMutex),
    member([H|T],NMutex),
    derive([H|T]),!,fail().
% All tests passed - succeed.
testIntegrity().
```

The meta inference rule, iterates over all integrity constraints, with a *xor* operator and uses a list structure $Mutex$ for the defined constraints in each integrity clause. It deletes the (hypothetical) literal *Literal* from the list creating a new list $NMutex$ or fails and backtracks to then next integrity constraint, if the literal is not contained in the list. If the literal is contained, it iterates over all remaining constraint literals in the new list ($member([H|T], NMutex)$) and tries to derive any of these conflicting literals, failing if one can be derived otherwise it succeeds. Note, that I use a "cut-fail" implementation here, which raises a failure immediately when a violation is detected and hence is highly efficient, but imposes a linear processing (which is supported by the semantics of ContractLog - see section 4.1.3). If the underlying procedural semantics of the rule engine is non-linear, then default negation can be used, i.e., the integrity constraints succeeds if none of the conflicting literals $\sim L$ fails, which requires to prove all literals without the possibility to cut and fail immediately when a conflict has been detected. The implementation of the meta inference rules for the other integrity operators work similarly, e.g., for the *and* operator:

```
% test And integrity constraints
testIntegrity():-
    integrity([and|AndConstraints]),
    member(M,AndConstraints),
    not(M),!,fail().
```

The meta inference rule iterates over all constraints and immediately fails if one can not be derived. For the implementation of the other inference rules see the integrity script of the ContractLog distribution.

The following theorem now establishes a correspondence between the interpretation of the integrity meta program and the satisfaction consequences for integrity validation of a knowledge base KB containing integrity constraints.

Theorem 8 *Let KB be a knowledge state and $\overline{IC}$ a set of integrity constraints in KB. Then $KB \models \overline{IC}$ iff $KB \models testIntegrity()$*

4.4.3 Extended Prioritized Defeasible Logic

Based on integrity constraints, which indicate logical and domain-specific conflicts, the defeasible logic extension of ContractLog is used to resolve these conflicts. In the following I will detail syntax and semantics of the prioritized defeasible logic.

4.4.3.1 Syntax of Prioritized Defeasible Logic Programs

The rules and facts of a defeasible theory $\Phi_{DefL} = \langle \overline{Fa}, \overline{R}, > \rangle$, where $\overline{Fa}$ is a finite set of facts, R a finite set of rules and $>$ an acyclic superiority relation on $\overline{R}$, are defined over a signature Σ as defined in the previous sections. A rule $r : B(r) \hookrightarrow H(r)$ consist of a rule label r which is a set of meta data annotations, a body $B(r)$ which is a set of literals, a head $H(r)$ which is the conclusion literal and an arrow $\hookrightarrow$ which is a place holder for the defeasible arrow $\Rightarrow$ and strict arrow $\rightarrow$ and a priority relation $>$ on r, i.e., $r1 > r2$ which states that a rule with label $r1$ is preferred to a rule with label $r2$. Recall from section 4.3 that in ContractLog the labelling function *metadata* assigns a set of meta data annotations as a label with a rule. To be usable in the context of defeasible reasoning the relation $>$ must be acyclic, i.e., the transitive closure must be irreflexive. In contrast to the standard formulations of defeasible logics, which only support priority definitions between rules, the priority relation $>$ in ContractLog is defined as a general transitive and anti-symmetric relation between rules and between rule sets, so called modules. To unambiguously define the superiority between two rules or rule sets their rule labels resp. their module labels, i.e., their unique object ids (oid), are used. Intuitively, the intended meaning of prioritized modules is, that all rules that are defined in a superior module are considered strictly stronger than the set of rules in the lower

module. Therewith, it becomes possible to define on a global level different superiority policies for complete modules (rule sets) such as prefer more specific knowledge over general knowledge or prefer more recently added knowledge over older knowledge in order to settle conflicts between rules in different modules. The local preferences defined as superiority relations between rules within a module define a partial order between rules within the global preferences between modules. For instance, if a module P_1^{part} consists of two prioritized rules $r_1 < r_2$ and another module M_2 consists of the prioritized rules $r_3 < r_4$ and both modules are prioritized by $P_1^{part} < P_2^{part}$ then $r_1 < r_2 < r_3 < r_4$. That is, the local priorities between rules and the global priorities between modules define a prioritized program with a partial, transitive and anti-symmetric order $\leq$ between the rules.

Definition 118 *(**Prioritized Logic Program**) A prioritized logic program (PLP) P is a finite partially-ordered set of rule sets, called modules P^{part}, where $\leq$ represents a transitive and acyclic partial order defined between the modules. Each module $P_i^{part} \in P$ can be further decomposed to smaller prioritized modules with a set of two rules order by $\leq$ as the smallest unit. If no explicit superiority relation is defined between two rules r_1 and r_2 in a module then they have equal priority, i.e., $r_1 = r_2$. The same applies to priorities between modules.*

A defeasible theory Φ_{DefL} is mapped to a meta program $P(\Phi_{DefL})$ in logic programming form in ContractLog as follows:

1. Each priority relation $r_1 > r_2$ where r_i is an unique rule resp. module meta data label (typically the rule resp. module object identifier) is stated as: $overrides(r_1, r_2)$.
2. Each defeasible rule $r : H \Leftarrow B$ is translated into the following set of meta rules:

```
metadata(label(r))::defeasible(H) :-  % defeasible rule head
    testIntegrity(H),       % strictly overtuned ?
    defeasible(B),          % body holds defeasible?
    not(defeated(R,H)).     % not defeasible defeated?
% auxiliary rule used in inference rules
neg(blocked(defeasible(H))):- testIntegrity(H), defeasible(B).
% auxiliary rules used in meta program for reasoning
body(defeasible(H)) :- defeasible(B).
```

Example 20

```
% Let D be a defeasible theory consisting of the following rules
%
% discount5: discount(X,5) <= defeasible(status(X,silver))
% discount10: discount(X,10) <= defeasible(status(X,gold))
```

```
% overrides("discount10","discount5")
% integrity(xor(discount(X,5),discount(X,10)))
%
% D is mapped to the following meta program P(D)

metadata(label(discount5))::defeasible(discount(X,5)):-
    testIntegrity(discount(X,5)),
    defeasible(status(X,"silver")),
    not(defeated("discount5",discount(X,5))).
neg(blocked(defeasible(discount(X,5)))):-
    testIntegrity(discount(X,5)), defeasible(status(X,"silver")).
body(defeasible(discount(X,5))):- defeasible(status(X,"silver")).


metadata(label(discount10))::defeasible(discount(X,10)):-
    testIntegrity(discount(X,10)),
    defeasible(status(X,"gold")),
    not(defeated("discount10",discount(X,10))).
neg(blocked(defeasible(discount(X,10)))):-
    testIntegrity(discount(X,10)), defeasible(status(X,"gold")).
body(defeasible(discount(X,10))) :- defeasible(status(X,"gold")).

overrides("discount10","discount5").
integrity(xor(discount(X,5),discount(X,10))).
```

The example shows the meta program mapped from a defeasible theory consisting of two defeasible rules where the heads are in conflict according to the given "xor" integrity constraint. The second rule "*discount*10" has higher priority than the first rule "*discount*5".

4.4.3.2 Proof-Theoretic Semantics

In section 4.1.2 I have discussed defeasible well-founded semantics as default semantics for extended ContractLog LPs. In this section I detail the proof theoretic semantics which extends the proof theory defined in [Bil93] and [ABGM01] with arbitrary conflicts stated in terms of integrity constraints and local and global priorities between local rules and global rule sets (modules). A conclusion of a defeasible theory Φ_{DefL} is a tagged literal L^t of the form $+\triangle L$, $-\triangle L$, $+\partial L$, $-\partial L$, where $\triangle$ means strictly provable, ∂ means defeasible provable and $\triangle$ and ∂ may have positive or negative polarity. Provability in defeasible logic is based on a finite derivation sequence of tagged literals satisfying conditions which correspond to inference rules for each of the four kinds of conclusions that specify how a derivation may be extended. The extended inference rules are as follows:

Let $\overline{R}$ be the set of rules in Φ_{DefL}, $\overline{R}_s$ be the set of all strict rules in $\overline{R}$, $\overline{R}_d$ be the set of defeasible rules in $\overline{R}$ and $\overline{R}_{sd}$ be the set of strict and defeasible rules

in $\overline{R}$. $\overline{R}[L]$ denotes the set of rules in $\overline{R}$ with consequent L (in their head). $-L$ denotes arbitrary conflicting literals to L wrt the integrity constraints in which L is defined. A derivation is a finite sequence $\overline{L^t} = \{L^t(1), .., L^t(n)\}$ of tagged literals satisfying the following conditions, where $L^t(1..i)$ denotes the initial part of $\overline{L^t}$ of lenght i:

- $+\triangle$: If $L^t(i+1) = +\triangle L$ then either $L \in F$ or $\exists r \in \overline{R}_s[L] \forall L_b \in B(r) : +\triangle L_b \in L^t(1, .., i)$
- $-\triangle$: If $L^t(i+1) = -\triangle L$ then either $L \ni F$ and $\forall r \in \overline{R}_s[L] \exists L_b \in B(r) : -\triangle L_b \in L^t(1, .., i)$
- $+\partial$: If $L^t(i+1) = +\partial L$ the either
  1. $+\triangle \in L^t(1, .., i)$ or
  2. (3.1) $\exists r \in \overline{R}_{sd}[L] \forall L_b \in B(r) : +\partial L_b \in L^t(1, ..i)$ and
     (3.2) $-\triangle - L \in L^t(1, .., i)$ and
     (3.3) $\forall s \in \overline{R}[-L]$ either
       (3.3.1) $\exists L_b \in B(s) : -\partial L_b \in L^t(1, .., i)$ or
       (3.3.2) $\exists t \in \overline{R}_{sd}[L]$ and $\forall L_b \in B(t) : +\partial L_b \in L^t(1, ..i)$ and $t > s$
- $-\partial$: If $L^t(i+1) = -\partial L$ the either
  1. $-\triangle \in L^t(1, .., i)$ or
  2. (4.1) $\forall r \in \overline{R}_{sd}[L] \exists L_b \in B(r) : -\partial L_b \in L^t(1, ..i)$ or
     (4.2) $+\triangle - L \in L^t(1, .., i)$ or
     (4.3) $\exists s \in \overline{R}[-L]$ such that
       (4.3.1) $\forall L_b \in B(s) : +\partial L_b \in L^t(1, .., i)$ and
       (4.3.2) $\forall t \in \overline{R}_{sd}[L]$ either $\exists L_b \in B(t) : -\partial L_b \in L^t(1, ..i)$ or $t \not> s$

That means, a conclusion L is strictly derivable $+\triangle L$ when there is a proof for L using only facts and strict rules. To prove that L is not strictly provable $-\triangle L$, L must not be a fact and for every rule r with head L at least on body literal L_b of r must not be strictly provable $-\triangle L_b$, i.e., r must be inapplicable. L is defeasible provable $+\partial L$ if either (1) L is already strictly provable, or (2) there is a strict or defeasible rule with head L which is applicable, i.e., the body is defeasibly provable $+\partial L_b$, with $L_b \in B(r)$ (3.1) and no conflicting literal $-L$ is strictly provable $-\triangle - L$ (3.2) and all rules s with a conflicting literal $-L$ in their heads with attack r (2.3) either are inapplicable since their bodies fail $-\partial L_b$ with $L_b \in B(s)$ or there is a rule t with head L which is applicable $+\partial L_b$ with $L_b \in B(t)$ and t is superior to s, i.e., each attacker s is overtuned by a stronger rule t. To prove that L is not defeasibly provable $-\partial L$, first L must not be strictly provable $-\triangle L$ (1) and not defeasible provable (2), which means

that either all rules r with head L are inapplicable (2.1) or a conflicting literal $-L$ is strictly provable (2.2) or there is a rule applicable attacker rule s which is superior to all applicable rules t with head L. In other words a literal L is (defeasibly) derivable when:

- L is a fact, or
- there is an applicable strict or defeasible rule with head L and either (1) all rules with conflicting heads $-L$, so called attackers, are discarded or (2) all attackers are weaker than an applicable rule for L.

Likewise, for $-\partial$ and $-\triangle$ the conditions are negated. The semantically non-trivial difference to the original formulation of the proof-theory given in [Bil93, ABGM01] is, that ContractLog considers:

1. arbitrary conflicting attackers wrt defined integrity constraints in contrast to pure negative/positive complements, and
2. global superiorities between rule sets (modules / scopes) in addition to the local priorities between rules, i.e., prioritized defeasible logic programs.

This generalization and extension is very useful to define general superiority rules such as "prefer positive knowledge over negative knowledge": $overrides([P|Args], neg([P|Args]))$ and derive context dependent preferences dynamically, e.g., $overrides(...) : -....$ In the following subsection I present the metaprogram which implements the proof inference rules in ContractLog in a LP.

4.4.4 Defeasible Metaprogram

The inference rules for defeasible reasoning in ContractLog are implemented as a meta program specifying the structure of defeasible reasoning. This meta implementation of the defeasible inferences allows applying different semantics such as WFS or STABLE to the syntactic meta formulation of a defeasible theory. It also provides more expressiveness adding e.g., both explicit and default (proof) failure in terms of default negations and explicit negations as described in section 4.1.2. Since I do not introduce any meta constructs for strict knowledge in ContractLog the inference rules for strict provability are given by classical derivation and need no extra meta program representation. The meta inference rules defining the predicates corresponding to $+/-\partial$ are as follows:

```
% strict knowledge is also defeasible derivable
defeasible([P|Args]) :- bound(P), derive([P|Args]).

% a defeasible rule is blocked if it is not neg blocked
blocked(defeasible([P|Args])):-
    bound(P), not(neg(blocked(defeasible([P|Args])))).
```

```
% defeated rules defined by either rule oid (=rule label/name)
% or rule's head literal
defeated(OID, Literal):-
    % test XOR (mutex) integrity constraints
    integrity([xor|Mutex]), % all mutex integrities
    delete(Literal,Mutex,NMutex), % list with opposers
    member(Opposer,NMutex), % all opposers
    neg(blocked(defeasible(Opposer))), % opposer not blocked
    not(neg(overruled(OID,Literal,Opposer))), % overruled by opposer
    !.

% consider superiority between head literals of local rules
neg(overruled(OID,Literal,Opposer)):-
    overrides(Literal, Opposer),!.
% consider superiority between rule labels (rule names)
neg(overruled(OID,Literal,Opposer)):-
    metadata(defeasible(Opposer),OpposerID,label),
    overrides(OID,OpposerID),!.
% consider superiority between rule sets (modules)
neg(overruled(OID,Literal,Opposer)):-
    metadata(defeasible(Literal),ModuleID,src),
    metadata(defeasible(Opposer),OpposerModuleID,src),
    overrides(ModuleID,OpposerModuleID), % defeated
    !.
```

As discussed in the syntax section 4.4.3.1, each defeasible rule r is translated into its meta programming format:

```
metadata(label(r))::defeasible(H) :-  % defeasible rule head
        testIntegrity(H),       % strictly overtuned ?
        defeasible(B),          % body holds defeasible?
        not(defeated(r,H)).     % not defeasible defeated?
% auxiliary rule used in inference rules
neg(blocked(defeasible(H))):- testIntegrity(H), defeasible(B).
% auxiliary rules used in meta program for reasoning
body(defeasible(H)) :- defeasible(B).
```

The defeasible rule $r : defeasible(H)$ with head H is provable if it does not violate the integrity of any strict knowledge $testIntegrity(H)$ and all prerequisites in the body B defeasibly hold $defeasible(B)$, i.e., r is applicable, and it is not defeated by any other defeasible rule $not(defeated(r, H))$. The defeasible meta inference rules prove if the defeasible rule r with the head H is not defeated by testing if the rule defeasibly violates the integrity of the logic program according to the defined integrity constraints. For each integrity constraint where H is a member the defeasible integrity test meta program executed the following steps:

1. Test whether all mutual exclusive conflicting literals of H are blocked, i.e., can not be derived using the $neg(blocked(...))$ auxiliary rules: $neg(blocked(defeasible(Opposer)))$.
2. If a conflicting literal is not blocked, test whether it is overridden by the defeasible rule using the head literal H or the local rule label r or the global module label to which r belongs to, i.e., has higher priority than the opposer: $overrides(Literal, Opposer)$ or $overrides(Rule, Opposer)$ or $overrides(RuleModuleOID, OpposerModuleOID)$. If a conflicting defeasible rule, i.e., a rule which is defined to be conflicting in an integrity constraint, is not blocked and is of higher priority than the defeasible rule r, r is defeated and will not be concluded.

Theorem 7 in section 4.1.2 relates the meta program formalization to the defeasible proof theory.

4.4.5 Summary

In contrast to most non-monotonic logic approaches such as prioritized circumscription, hierarchic autoepistemic logic, prioritized default logic which represent preference information in an "external" manner expressed outside of the logical language, ContractLog implements a homogeneous meta programming approach where priorities and defeasible rules in combination with integrity constraints and other rule types are formalized as LPs. The main advantages of this meta programming approach for integrity and defeasible reasoning are:

- homogeneous representation of integrity constraints and defeasible theories as LPs
- easy to extend and combine with other logic programming formalisms, e.g., default negation
- interpretation with different LP semantics possible

In summary, the main contributions of the integrity preserving, preferenced, defeasible logic extension of ContractLog are:

- expressive integrity constraints resp. conditional integrity rules to define logical and arbitrary application-specific conflicts
- integrity test axioms to test the actual or hypothetical KB states (program states)
- conditional superiority relations between rules and modules (resp. scopes)
- defeasible reasoning and conflict handling wrt arbitrary conflicts stated in terms of integrity constraints and the priority definitions

In particular, the possibility to prioritize complete rule sets and the flexibility to define conflicts not just between positive and negative conclusions as in standard defeasible logic, but also between arbitrary conflicting literals is crucial

in the contract domain, e.g., to represent typical legal principles such as "lex posterior" or "lex superior" for resolving legal conflicts or to define contract hierarchies with alternatives and exceptional policies.

4.5 Transactional Module-based Update Logic

In order to adapt SLAs dynamically to changing requirements and allow dynamic modelling of the contract behavior, updates of the extensional fact knowledge but also of the intensional rules play an important role. Knowledge updates at runtime are not included by the standard (Horn) semantics of logic programs. Typical update primitives such as "assert" and "retract" provided by LP languages such as Prolog are constraint to updates of the extensional fact base and the semantics for updates does not offer a declarative understanding. In particular it gives no answers how update primitives should interact with other logical operators and how a sequence of updates can be logically treated. For instance, if a query triggering a sequence of updates fails, the side effects of "assert" and "retract" can not be rolled back in a transactional style. Various languages have been proposed for specifying updates of LPs leading to dynamic LPs (aka evolving LPs), e.g. [LHL95, Zan93, NK88, AV91, LAP01, ABLP02, EFST01, Lei03, Pas06b, Pas05a] and transaction logics, e.g. [BK95], which try to provide a declarative meaning to dynamic LPs.

In ContractLog I follow the general approach of dynamic LPs where updates are considered as special actions which transit the initial program (the actual knowledge base resp. knowledge state) to a new extended or reduced state. While the standard semantics of updates based on dynamic logics supports sequential composition of update sequences it fails short for complex transactional bulk updates where updates of rule sets and/or fact sets are constructed and executed as complex update actions (update sequences). Updates in ContractLog are based on the meta data annotated labelled logic which allows to bundle clause sets to modules having a unique module oid (object id). Each update asserting new knowledge to the KB is treated as a new (sub-)module which has a unique ID (key), the module ID, with which it is asserted into the KB. That is, the updates which are executed during rule derivations lead to a transition resp. a sequence of transitions of the actual knowledge state to an extended or reduced state. In the transactional mode updates in failed derivation trees are rolled-back to the state of the last back-tracking point by inverting the update primitives of the remembered transition sequence. The main advantages of this approach are:

- updates of the extensional and intensional knowledge are supported including bulk updates and imports of external modules
- updates are applied as subgoals in the body of rules in combination with standard goals
- the updates are treated as goals which have both a truth value and an

effect on the state of the knowledge base which enables a model theoretic interpretation and a SLD-style proof-theoretic treatment of transactional update execution, where in case of failed proof trees updates are rolled-back

- the treatment of updates as modules with an unique oid makes it easy to add or remove knowledge from the knowledge base
- in the absence of updates the program reduces to a standard LP

4.5.1 Syntax of Update Primitives

In ContractLog the set of function symbols F in the signature is extended with three special update functions *add*, *remove* and *transaction* and two auxiliary functions *commit* and *rollback*. These update primitives are more expressive than the simple assert/retract primitives found in typical Prolog interpreters and allow transactional as well as bulk updates of knowledge including updating of facts and rules. They enable arbitrary knowledge updates, e.g., adding (*add*) /removing (*remove*) rules or complete rule sets including the integration of knowledge from external sources and transactional update (*transaction*) which are rolled back if the execution fails. Transactions might be explicit committed or rolledback. Each update has a unique ID with which it is asserted into the KB as a module.

Example 21

```
add("./examples/test/test.prova")  % add an external script
add("http://rbsla.com/ContractLog/datetime.prova") % from URL
add(id1,"r(1):-f(1). f(1).")% add rule "r(1):-f(1)." and fact "f(1)."
add(id2,"r(X):-f(X).")         % add rule "r(X):-f(X)."
p(X,Y) :- % object/variable place holders _N: _0=X ; _1=Y.
    add(id3,"r(_0):-f(_0), g(_0). f(_0). g(_1).",[X,Y]).
remove(id1)            % remove all updates with id
remove("./examples/test/test.prova")   % remove external update
```

The examples show different variants of updates with external modules imported from their URIs, user-defined updates asserting rules and facts and updates with previously bound variables from other goals which are integrated into the updates using place holders $\_X$.

Remarkably, updates to the KB are handled as modules, i.e., as (smaller) logic programs which might contain further updates and imports of other modules, leading to nested updates with hierarchical submodules. This concept facilitates the required modular and distributed representation and management of contracts as sets of LP scripts provided at URLs on the (Semantic) Web. The scripts can be dynamically added and removed from the knowledge base using their module object ids (typically the URL or the user defined label).

4.5.2 Semantics of Updates

The semantics of updates in ContractLog is based on the notion of knowledge states and transitions from one state to another.

Definition 119 *(**Knowledge State**) A knowledge state represents a knowledge base KB_k, where $k \in \aleph$, consisting of a finite extensional database (EDB) of facts and a finite intensional database (IDB) of rules.*

Note that according to the labelled, modularized logic in ContractLog a state, i.e., a knowledge base KB_k, might consist of several EDBs and IDBs which are bundled to possibly further nested submodules, each having an unique ID (the module oid). Intuitively, a state represents the union of all clauses stored in all modules in the combined knowledge base. An update is then a transition which adds or removes facts and/or rules from/to the EDB and/or IDB and changes the knowledge base. That is, the KB transits from the initial state KB_1 to a new state KB_2. I define the following notion of positive (add) and negative (remove) transition:

Definition 120 *(**Positive Update Transition**) A positive update transition, or simply positive update, to a knowledge state KB_k is defined as a finite set $U^{pos}_{oid} := \{r_N : H \leftarrow B, fact_M : A \leftarrow\}$ with A an atom denoting a fact, $H \leftarrow B$ a rule, $N = 0, .., n$ and $M = 0, ..m$ and oid being the update oid which is also used as module oid to manage the knowledge as a new module in the KB. Applying U^{pos}_{oid} to KB_k leads to the extended state $KB_{k+1} = \{KB_k \cup U^{pos}_{oid}\}$. Applying several positive updates as an increasing finite sequence $U^{pos}_{oid_j}$ with $j = 0, .., k$ and $U^{pos}_{oid_0} := \emptyset$ to KB_0 leads to a state $KB_k = \{KB_0 \cup U^{pos}_{oid_0} \cup U^{pos}_{oid_1} \cup ... \cup U^{pos}_{oid_k}\}$.*

That is a state KB_k is decomposable in the previous knowledge state $k-1$ plus the update: $KB_k = \{KB_{k-1} \cup U^{pos}_k\}$. I define $KB_0 = \{\emptyset \cup U^{pos}_{oid_0}\}$ and $U^{pos}_{oid_0} = \{KB$: the set of rules and facts defined in the program $P\}$, i.e., loading/parsing the program P from a LP script is the first update. Likewise, I define a *negative update transition* as follows:

Definition 121 *(**Negative Update Transition**) A negative update transition, or for short a negative update, to a knowledge state KB_k is a finite set $U^{neg}_{oid} := \{r_N : H \leftarrow B, fact_M : A \leftarrow\}$ with $A \in KB_k$, $H \leftarrow B \in P$, $N = 0, .., n$ and $M = 0, ..m$, which is removed from KB_k, leading to the reduced program $KB_{k+1} = \{KB_k \setminus U^{neg}_{oid}\}$.*

Applying arbitrary sequences of positive and negative updates leads to a sequence of KB states $KB_0, .., KB_k$ where each state KB_i is defined by either $KB_i = KB_{i-1} \cup U^{pos}_{oid_i}$ or $KB_i = KB_{i-1} \setminus U^{neg}_{oid_i}$. In other words, KB_i, i.e., the set of all clauses in the KB at a particular knowledge state i, is decomposable in

the previous knowledge state plus/minus an update, whereas the previous state consists of the state $i-2$ plus/minus an update and so on. Hence, each particular knowledge state can be decomposed in the initial state KB_0 and a sequence of updates. Although an update might insert more than one rule or fact, i.e., insert or remove a complete module, it nevertheless is treated as an elementary update, a so called bulk update, which transits the current knowledge state to the next state in an elementary transition: $\langle KB_i, U_{oid}^{pos/neg}, KB_{k+1}\rangle$. Intuitively, one might think of it as a complex update action which performs all inserts resp. removes simultaneously.

Elementary updates have both a truth value and a side effect on the knowledge base. All goals after an update apply on the extended resp. reduced transition state, i.e., the truth value of a goal G depends on the actual knowledge state KB_i, denoted by $KB_i \models G$, where KB_i is the current state of the last update transition. For instance, a query "$q(X), add(u1, "p(\_0).", [X]), p(Y)$" will succeed and return all values which are bound to X in the goal $q(X)$ for the variable Y due to the serial update and the subsequent goal $p(Y)$. Note that in the non-transactional style updates in (serial) rules are not rolled-back to the original state if the derivation fails and the system backtracks. Typically this "weak" non-transactional semantics is intended when external script are imported or new rule sets are added as modules in an atomic action. That is, independently, of whether the particular derivation in which the update is performed fails from some reason the update transition to the next knowledge state subsists and is not rolled back in case of failures.

4.5.3 Transactional Updates with Integrity Tests

Transactional updates in ContractLog are inspired by the serial Horn version of transaction logics (TR) [BK95] which allows programming of update transactions using rule definitions. In contrast to TR, ContractLog makes no distinction between a transaction base which specifies the defined transactions and a database which is a set of logical formulae. In ContractLog there is no seperation between standard rules consisting of only classical literals and serial "update" rules consisting of a mixture of classical and update literals. Moreover, update literals are prohibited in the rules' heads, since sequences of active rules, i.e., rules where update actions trigger other updates, are represented as reactive rules in ContractLog (see section 4.7).

In the following I will describe syntax and semantics of transactional updates which adopts a notion of executional entailment from TR, where the truth of a query or goal is defined on sequences of state transitions.

4.5.3.1 Syntax of Transactional Updates

Definition 122 ***(Transactional Update)*** *A transactional update is an update, possibly consisting of several atomic updates, which must be executed completely or not at all. In case a transactional update fails, i.e., it is only partially*

executed or violates integrity wrt to integrity constraints, it will be rolled back otherwise it will be committed. Formally, a transactional update is defined as follows:

$U_{oid}^{trans} := U_{oid_1}^{pos/neg}, .., U_{oid_n}^{pos/neg} \& \overline{IC}$

, where $\overline{IC} = \{IC_1, .., IC_m\}$ *is a possibly empty set of integrity constraints which must hold after the update has been executed. In case an integrity constraint is violated the update is rolled back.*

Syntactically, a transactional update is represented in ContractLog by the special function *transaction* which takes positive or negative updates and optionally tests on integrity constraints (or test cases) as arguments.

Example 22

```
transaction(remove(...)) % transactional remove
transaction(add(...)) % transactional update
% transaction update with explicit test case test.
transaction(add(...),"testcase1.prova")
commit(id5)        % commit transaction with ID id5
rollback(id5)      % rollback transaction with ID id5.
```

The major difference to the non-transactional updates is in the semantics of transactional updates which is defined by executional entailment on transition sequences.

4.5.3.2 Model-Theoretic Semantics of Transactional Updates

The semantics of transactional updates is built on the concept of sequences of state transitions, called execution paths.

Definition 123 ***(Execution Path)*** *An execution path* π *is a sequence of state transitions* $\pi = \langle KB_0, U_{oid_1}^{pos/neg} \rangle \rightarrow \langle KB_1, U_{oid_2}^{pos/neg} \rangle \rightarrow \ldots \rightarrow \langle KB_n, \emptyset \rangle$*, where* KB_i *is a knowledge state and* $U_{oid_{i+1}}^{pos/neg}$ *a positive or negative update which transits* KB_i *to the next state* KB_{i+1}*.*

I abbreviate the notation of an execution path of length k as a finite and arbitrary sequence of state identifiers (given by the update oids): $\langle KB_1, ..., KB_k \rangle$. As in transaction logics the truths of update goals in ContractLog rules are defined on execution paths. Thus, in case of updates in rules the answer to a query is not determined by the current knowledge base alone, but depends on the entire execution paths. An execution path in ContractLog is defined

on sequences of KB states. Free queries with variables are supported that non-deterministically execute along any one of many possible paths returning a set of answers. Intuitively, the execution corresponds to truth on paths. Accordingly, only execution paths which return a non-empty answer are considered and their executed transactional updates are committed, whereas paths with empty answer sets are backtracked and the processed transactional updates within such paths are rolled back to the state of the last backtracking point. A query fails if the answer set is empty for every possible path. The logical account of transactional execution (derivation with transactional updates) is given by the concept of executional entailment adapted from TR.

Definition 124 ***(Executional Entailment)*** *Let KB_0 be an initial KB state and Q be a query which might contain free variables $X_1, .., X_n$ then $KB_0, \pi \models Q$, i.e., Q is true in KB_0, iff there exists a path $\pi = \langle KB_0, .., KB_k \rangle$ which returns a non-empty answer for all variables in Q. Q fails if it returns an empty answer set for every possible execution path π.*

That is, a query involves a mapping from sequences of KB states to sets of tuples of ground terms in each state. In case of free queries with variables several execution paths are considered to produce answers for the query Q, then the final state $\overline{KB}$ which becomes the new knowledge base (state) is the union of all final states of valid execution paths π_i which entail Q. Queries which do not involve any updates, i.e., which do not consider serial update rules but only LP rules with standard atoms without any side effects on the KB, have an execution path with length $k = 0$. In this case, a goal Q is entailed if $KB, \pi \models Q$ and $\pi = \{KB\}$, i.e., there is not state transition and accordingly executional entailment reduces to standard LP entailment $KB \models Q$.

4.5.3.3 Proof-Theoretic Semantics of Transactional Updates

The procedural semantics of transactional updates in ContractLog naturally originates from the common top-down SLD-style proof procedures of logic programming, i.e., SLD(NF) and variants such as the SLE resolution described in section 4.1.3. Obviously, the linearity of these resolutions is an important property in the context of transactional updates which are processed on serial execution paths. The following inference system describes a natural extension of the unification based, top-down, linear systems with transactional updates. As discussed in the previous section an inference is successful, if it finds an execution path for the query Q, i.e., a sequence of state transitions such that $KB_i, \pi \models Q$ where i is the actual state of the KB. The inference system is given by the following inference rules and one axiom which are adapted from TR:

Definition 125 ***(Transactional Inference Rules)***

- *Axiom:* $KB, \pi \vdash ()$

- *Serial Modus Ponens: Let $H \leftarrow B$ be a rule in KB, with B being a mixed serial body with transactional update literals and normal literals and $(L_b, rest)$ be a conjunction of (serial) goal literals. If L_b and H unify with the mgu θ and $\overline{X}$ being a set of universally quantified variables, then* $\frac{KB,\pi\vdash(\forall\overline{X})(B,rest)\theta}{KB,\pi\vdash(\forall\overline{X})(L_b,rest)}$
- *Querying: If $L_b\theta$ and $rest\theta$ share no variables and $KB \models (\forall\overline{X})L_b\theta$, then* $\frac{KB,\pi\vdash(\forall\overline{X})rest\theta}{KB,\pi\vdash(\forall\overline{X})(L_b,rest)}$
- *Elementary Updates: If $L_b\theta$ and $rest\theta$ share no variables and $KB_2 \models (\forall\overline{X})L_b\theta$, then* $\frac{KB_2,\pi\vdash(\forall\overline{X})rest\theta}{KB_1,\pi\vdash(\forall\overline{X})(L_b,rest)}$

The inference system takes expression of the form $KB, \pi \vdash B$ as input, meaning that the query or update formed by the instantiated body goals can be executed on the path π starting at the initial/actual state KB. If the upper sequent of the inference rules can be inferred, then the lower sequent can also be inferred. The axiom of the inference system formalizes an empty goal which always succeeds and has no side effect. The first inference rule says that if $(B, rest)$ succeeds from KB, π and L_b is defined by π then $(L_b, rest)$ also succeeds from KB, π. The second inference rule says if $rest$ succeeds from KB, π and L_b is true at the actual state KB, then $L_b, rest$ also succeeds from KB, π. Inference rule three says if L_b updates the knowledge state from KB_1 to KB_2 and $rest$ succeeds from KB_2 then $(L_b, rest)$ succeeds from KB_1. The inference system executes serial rules in backward-reasoning SLD-style by constructing executional deductions based on execution paths which transit the knowledge state according to the sequentially processed updates mixed with standard queries in the rule bodies. In the presence of variables several deductions might fail and their update transitions must be rolled back to the state of the last backtracking point in the resolution process. To achieve this the sequence of state transitions is remembered on each execution path. As discussed each update has a unique id and is inserted as a module resp. submodule of another module in the combined KB. A roll back to the previous state before the update means to apply the complement update function on the added or removed modules:

Definition 126 ***(Rollback of Update)*** *A transactional update is rolled back by inverting the update primitive:*
$KB_i = KB_{i+1} \setminus U^{trans}_{oid}$ *iff* $KB_{i+1} = KB_i \cup U^{trans}_{oid}$
$KB_i = KB_{i+1} \cup U^{trans}_{oid}$ *iff exists* $KB_{i+1} = KB_i \setminus U^{trans}_{oid}$

Note, that due to the module concept in ContractLog only the transition sequence consisting of the update state oids and the update primitive needs to be remembered to rollback a sequence of transactional updates.

4.5.4 Summary

Dynamic updates of the extensional facts as well as the intensional rules are crucial in the SLA domain in order to adapt the contracts to changing requirements such as new service levels or renegotiated policies, rights and obligations. Moreover, in a distributed environment the contract structure typically consists of scattered contract modules, data sources and domain-specific vocabularies which are dynamically imported and added to the combined KB. The ability to perform bulk updates and imports on rule sets and apply conditional updates and transactional update sequences during serial rule execution is one important part of the behavioral and reactive logic of active rule-based SLA monitoring and enforcement.

The declarative semantics of updates is directly build on top of the labelled logic and module concept of the ContractLog KR which allows to treat updates in serial rule executions as transition sequences of knowledge states. Updates have both a truth value and a side effect on the knowledge base. The proof theory is an extension of the standard SLD-style resolution (SLE resolution) with serial processing of goal and update literals which is essential for the non-deterministic treatment of transactional updates. The operational semantics of transactional updates supports rollbacks of transition sequences of failed update transactions with possible subtransactions along their execution paths. Integrity constraints or more expressive test cases can be used to safeguard the outcome (the effect) of transactional updates on the knowledge system.

4.6 Temporal Event/Action Logic

Updates, as described in the previous section, are a special case of actions which materialize state changes in the knowledge base. This kind of (transactional) update logic is important to efficiently and actually manage, maintain and adapt distributed rule bases. However, it is mainly aimed for performing updates to the KB and it is not intended for reasoning about events, actions and their temporal effects on changeable properties of the knowledge systems, so called fluents. This expressiveness to describe sophisticated relationships between different states in workflow-like settings with transitions as effects of events/actions is provided by event / action logics. The ability to formally reason about the effects of events or actions on changeable states is vital in SLA monitoring and enforcement in order to track the contract states and derive the respective contractual consequences in each state. Typical examples are rights and obligations which hold in certain states or penalties which have to be payed for the duration of e.g., unavailability states. A particular advantage of event/action logics is their characterization in classical logic and their representation as logic programs. Hence, the formalisms are based on a clear logical semantics which is crucial when results need to be traceable and verifiable, as in the contract domain where agreements are legal entities. In section 3.5.1.4 I have discussed different event/action logics. The event calculus [KS86] is a well understood event/action logic formalism which

due to its linear treatment of time and its rich expressive power to specify and reason about event/actions and their effects qualifies as an adequate formalism for SLA representation.

In this section I introduce the core syntax and semantics of the event calculus formalism implemented in the ContractLog KR. It forms the basis for other formalisms of the KR such as the deontic logic or the complex event/action algebra, which will be described later. I first present the history and basic concepts of the event calculus and then the syntax and semantics.

4.6.1 History and Basic Concepts of Event Calculus

In this section I will recall the basic concepts in the event calculus (EC). For an overview on event and action logic formalisms see section 3.5.1.4. Kowalski and Sergot´s EC [KS86] is a formalism for temporal reasoning about events and their effects on a logic programming system as a computation of earlier events (long-term "historical" perspective). It defines a model of change in which events happen at time-points and initiate and/or terminate time-intervals over which some properties (time-varying *fluents*) of the world hold. Time-points are unique points in time at which events take place instantaneously. The basic idea is to state that fluents are true at particular time-points if they have been *initiated* by an event at some earlier time-point and not *terminated* by another event in the meantime. Similarly, a fluent is false at a particular time-point, if it has been previously terminated and not initiated in the meantime. That is, the EC embodies a notion of default persistence according to which fluents are assumed to persist until an event occurs which terminates them. This principle follows the axiom of inertia first proposed by McCarthy and Hayes which says: "Things normally tend to stay the same" [MH69]. A central feature of the EC is that it establishes or assumes a narrative time structure which is independent of any event occurrences. The time structure is usually assumed or stated to be linear although the underlying ideas can equally be applied to other temporal notions, e.g., branching structures, i.e., event occurrences can be provided with different temporal qualifications. Variants range from the normal structures with absolute times and total ordering to loose ones with only relative times and partial ordering. Given a history of events (set of event occurrences), the EC is able to infer the set of maximal validity intervals (MVIs) over which the fluents initiated and/or terminated by the events maximally hold. Therefore, a central feature of the Event Calculus, in comparison to other event/action logics such as Situation Calculus, is that an explicit linear time-structure, which is independent of any events, is assumed.

In ContractLog I apply absolute times with total ordering which can be easily represented and computed using predefined predicates and functions for date and time values of the ContractLog library. The granularity of time is application dependent. It may be a second or a minute in some contexts and a day in others. In the EC action occurrences are often special event types, i.e., the EC

deals with actions and events interchangeably. The core axioms of the classical EC are:

```
happens(Ev,Ti)         event Ev happens at time point Ti
initiates(Ev,Fl,Ti)    event Ev initiates fluent Fl for all time>Ti
terminates(Ev,Fl,Ti) event Ev terminates fluent Fl for all time>Ti
holdsAt(Fl,Ti)         fluent Fl holds at time point Ti
```

The Event Calculus was originally formulated as a logic program in [KS86] and many alternative logic program formulations and extensions which also formalize non-deterministic actions, concurrent actions, actions with delayed effects, gradual changes, actions with duration, continuous change, and non-inertial fluents have been subsequently proposed, see e.g. [Sha90, DMB92, Kow92, SK95, KM97a, Sha97a, Sha97b, KMT99]. The EC has also been formulated in modal logic, see e.g. [CCM96].

4.6.2 Syntax of the Event Calculus Logic

A classical logic axiomatizations of the EC has been given in [MS99]. In contrast, ContractLog exploits extended logic programs to axiomatizes EC domains as meta programs in logic programming form. The axioms describe when events / actions *occur* (transient view), *happens* (non-transient view) or are *planned* to happen (future view) within the EC time structure and which properties (*fluents*) are *initiated* and/or *terminated* by these events/actions under various circumstances.

Definition 127 *(Event Calculus Language) The EC signature in ContractLog is a multi-sorted signature with equality, with a sort Ev for events resp. actions, a sort Fl for fluents, and a sort Ti for timepoints. The EC language Σ^{EC} is a tuple $\langle \overline{Ev}, \overline{Fl}, \overline{Ti}, \leq \rangle$ where $\leq$ is a partial ordering defined over the non-empty set $\overline{Ti}$ of time points, $\overline{Ev}$ is a non-empty set of events/actions and $\overline{Fl}$ is a non-empty set of fluents. Timepoints, events/actions and fluents are n-ary functional literals L or $\neg L$ which might be reified.*

The calculus for event/actions and their effect is implemented as a meta program in LP format. The main EC axioms to axiomatizes an Event Calculus domain in ContractLog are:

```
occurs(Ev,Ti):   event Ev occurs at time interval Ti:=[Ti1,Ti2]
happens(Ev,Ti):  event Ev happens at time point Ti
planned(Ev,Ti):  event Ev is panned to happen at time point Ti

initially(Fl):        fluent Fl holds initially
initiates(Ev,Fl,Ti):  event Ev initiates fluent Fl for all time>Ti
```

```
terminates(Ev,Fl,Ti): event Ev terminates fluent Fl for all time>Ti

holdsAt(Fl,Ti):                      fluent Fl holds at time point Ti
holdsInterval([Ev1,Ev2],[Ti1,Ti2]):         holds between interval
holdsInterval([Ev1,Ev2],[Ti1,Ti2],[<Terminators>]): with terminators

trajectory(Fluent,Ti1,Parameter,Ti2, X): trajectory
valueAt(Parameter,Ti):        gives quantitative value at given
time

countMVI(Fluent,Number):     counts the number of MVIs
mvi(Fluent,Timespan):        computes all validity intervals
overallMVI(Fluent,Timespan): computes the overall mvi
```

Example 23

```
initiates(e1,maintenance,Ti).
terminates(e2,maintenance,Ti).

happens(e1,t1). happens(e2,t5).

:-solve(holdsAt(maintenance,t3))
:-solve(holdsAt(maintenance,t7))
```

The example states that an event $e1$ initiates a fluent *maintenance* and an event $e2$ terminates the fluent. The event $e1$ happens at timepoint $t1$ and the event $e2$ happens at timepoint $t5$. Accordingly the first query succeeds, i.e., the fluent state *maintenance* holds at timepoint $t3$, since *maintenance* is initiated at timepoint $t1$ and not terminated in between. The second query on timepoint $t7$ fails since *maintenance* is terminated at timepoint $t5$.

The implementation of the EC meta program is fully declarative and untyped. That is the terms used within the EC axioms might be constant, variable or complex (functional). In particular fluents and event/actions might be reified statements formalized as functions. For instance, events, fluents or time points can be represented as complex logical functions, constant String values, external Java objects or XML based or relational database serializations.

Example 24

```
happens(e1,datetime(2005,11,23,10,30,0)).
initiates(e1,permit(S,O,A), Ti).
terminates(e2, permit(S,O,A), Ti).
```

The example shows the use of complex terms in the EC. It defines a permission for a subject S to perform the action A on the object O which will be initiated

by the event $e1$ and terminated by the event $e2$. An event $e1$ happens at the timepoint $datetime(2005, 11, 23, 10, 30, 0)$. This reified treatment of complex terms qualifies the EC as a general inference system for various domains which need to reason about complex actions and events and their effects on complex states (see section 4.7.4 and section 4.8).

4.6.3 Semantics of the Event Calculus Logic

Circumscription [McC80] has been used by Shanahan [Sha97a] as a semantics in the EC, in particular as semantic for negation-as-failure and the frame problem. In ContractLog the semantics is based on a meta programming formalization which is represented as an extended LP. This enables a homogenous representation of EC axiomatizations as standard LPs in combination with the other formalisms of the ContractLog KR. Informally the declarative semantics is given by interpretations which map pairs of fluents and timepoints to truth values.

Definition 128 *(**Event Calculus Interpretation**) An interpretation is a mapping* $I : \overline{Ti}\, x\, \overline{Fl} \mapsto \{true, false\}$.

Definition 129 *(**Event Calculus Satisfaction**) An interpretation I satisfies a fluent Fl at timepoint Ti if $I(Fl, Ti) = true$ and $I(\neg Fl, Ti) = false$.*

Definition 130 *(**Instantiation and Termination**) Let Σ^{EC} be an EC language, D^{EC} be a domain description (an EC program) in Σ^{EC} and I be an interpretation of Σ^{EC}. Then a fluent Fl is instantiated at time point Ti_1 in I iff there is an event Ev_1 such that there is a statement in D^{EC} of the form $happens(Ev_1, Ti_1)$ and a statement in D^{EC} of the form $initiates(Ev_1, Fl, Ti)$. A fluent Fl is terminated at time point Ti_2 in I iff there is an event Ev_2 such that there is a statement in D^{EC} of the form $happens(Ev_2, Ti_2)$ and a statement in D^{EC} of the form $terminates(Ev_2, Fl, Ti)$.*

An interpretation qualifies as a model for a given domain description, if:

Definition 131 *(**Event Calculus Model**) Let Σ^{EC} be an EC language, D^{EC} be a domain description in Σ^{EC}. An interpretation I of Σ^{EC} is a model of D^{EC} iff $\forall Fl \in \overline{Fl}$ and $Ti_1 \leq Ti_2 \leq Ti_3$ the following holds:*

1. *If Fl has not been instantiated or terminated at Ti_2 in I wrt D^{EC} then $I(Fl, Ti_1) = I(Fl, Ti_3)$*
2. *If Fl is initiated at Ti_1 in I wrt D^{EC}, and not terminated at Ti_2 the $I(Fl, Ti_3) = true$*
3. *If Fl is terminated at Ti_1 in I wrt D^{EC} and not initiated at Ti_2 then $I(Fl, Ti_3) = false$*

The three conditions define the persistence of fluents as time progresses. That is, only events/actions have an effect on the changeable fluents (condition 1) and the truth value of a fluent persists until it has been explicitly changed by another event/action (condition 2 and 3). A domain description is consistent if it has a model. I now define entailment wrt to a meta program domain description:

Definition 132 ***(Event Calculus Entailment)*** *Let D^{EC} be an EC domain description. A fluent Fl holds at a timepoint Ti wrt to D^{EC}, written $D^{EC} \models holdsAt(Fl, Ti)$, iff for every interpretation I of D^{EC}, $I(Fl, Ti) = true$. $D^{EC} \models neg(holdsAt(Fl, Ti))$ iff $I(Fl, Ti) = false$.*

The inference calculus for the EC logic is implemented as a meta program in ContractLog. The core inference rules are:

```
%----------------
% Optimized version with cut to answer bound queries holdsAt(f,t)?
% ==> true/false
%----------------
holdsAt(Fluent,Time):-
    bound(Fluent),
    bound(Time), % Ti must be input / bound
    not(derivedFluent(Fluent)), % compute only non derived fluents
    initiates(AnEvent, Fluent, Time),
    happens(AnEvent, Before),
    less(Before, Time),
    notclipped(Before, Fluent, Time),  % assumes closed world
    !.

%----------------
% Optimized version to answer free queries holdsAt(Fl,t)?
% ==> All Fluents which hold at t
%----------------
holdsAt(Fluent,Time):-
    bound(Time), % Ti must be input / bound
    happens(AnEvent, Before),
    less(Before, Time),
    initiates(AnEvent, Fluent, Before),
    not(derivedFluent(Fluent)), % compute only non derived fluents
    notclipped(Before, Fluent, Time).  % assumes closed world
```

The basic calculus in ContractLog implements optimized variants for free and ground queries on the validity of fluents at time points: $holdsAt \subseteq \overline{Fl} \times \overline{Ti}$, where $\overline{Fl}$ a terms of sort fluent and $\overline{Ti}$ are terms of sort time. The first inference rule implements the variant for ground queries, i.e., $holdsAt(+,+)$, and the second inference rule is used for free queries, i.e., $holdsAt(-,+)$. For the rest

of this subsection I focus on various extensions to the basic formulation of the EC adopted from [MS99]. Due to lack of space I can only highlight some of the EC features which I have implemented in the ContractLog KR. I focus on the most interesting ones in the context of SLA representation and event/action processing.

For some domains, it is appropriate to categorize fluents into *normal fluents* (aka frame fluents), which hold over intervals of time with non-zero duration (as described above) and *derived fluents* (aka non-frame fluents) which represent indirect effects (e.g. state constraints). Therefore, I exclude derived fluents from the principle of default persistence and direct event initiation by introducing a new predicate $derived \subseteq \overline{Fl} : derived(fluent)$ which states that the fluent is a derived fluent. Derived fluents hold if an other (normal) fluent holds, i.e., $holdsAt(f2, Ti) \leftarrow holdsAt(f1, Ti)$ and $derived(f2)$.
This frame concept can be further extended to a fully *dynamic management of frames* in order to express that particular fluents have a default persistence during some intervals but not during special other intervals. That is, in certain states the default persistence of a fluent is disabled, so that the truth value of the fluent can fluctuate until the frame principle is reinitiated. To achieve this I introduce a new predicate $releases \subseteq \overline{Ev} \times \overline{Fl} \times \overline{Ti} : releases(e, f, t)$ which expresses that if an event e occurs at time point t it will disable the default persistence of fluent f until the axioms initiates or terminates reinitiate its default persistence. According to this a fluent can have four different truth values: *persistently true*, *persistently false*, *free true* and *free false*. To describe these states explicitly, I introduce a new predicate $releasedAt \subseteq \overline{Fl} \times \overline{Ti} : releasedAt(f, t)$ and add two auxiliary predicates $released \subseteq \overline{Ti} \times \overline{Fl} \times \overline{Ti}$: $released(t1, f, t2)$ which means that fluent f is released from persistence between $t1$ and $t2$ and $persistent \subseteq \overline{Ti} \times \overline{Fl} \times \overline{Ti} : persistent(t1, f, t2)$ which means a fluent is not released between $t1$ and $t2$.
To describe *delayed effects of events* in the EC we use a flexible approach based on trajectories and parameters [Sha90] I distinguish between the already introduced "normal" fluents and *parameters* which are non-persistent fluents representing an arbitrarily-valued function of time. I introduce the predicate $trajectory \subseteq \overline{Fl} \times \overline{Ti} \times \overline{Fl} \times \overline{Ti} \times \overline{C}$: $trajectory(f, t1, p, t2, v)$, which states that if fluent f is initiated at time $t1$ and continues to hold until time $t2$ then the parameter p has a value of v at time $t2$. I translate this into Event Calculus terms by a special extra axiom $valueAt \subseteq \overline{Fl} \times \overline{Ti} \times \overline{C} : valueAt(p, t2, x)$ which states that parameter p has value x at time $t2$. Applications of parameters are manifold. They can be used to represent continuous change, simple mathematical functions based on time or delayed effects such as time based countdowns or deadlines, e.g., to state that an event $e2$ happens if the countdown parameter reaches the value 0 while the decrease function is defined by a trajectory.

Example 25

```
trajectory(serverIsDown, Before, serverDownDuration, After, X):-
    math_sub(After,Before,X).
```

```
:-solve(valueAt(serverDownDuration,datetime(2006,1,1,13,34,10),Ti)).
```

The example defines a parameter *serverDownDuration* which computes the down time of a server as trajectory function over the fluent (state) $serverIsDown$. The query asks for the value of the parameter at the stated time point. This concept of parameters might be further extended to a more general approach using further axioms which explicitly utilize the mathematical definitions of continuity and differentiability of real-valued functions of time (cf. [San89b]) and might be further extended with a axiomatization of the peano axioms to reason of numbers. However, in ContractLog I do not follow this approach, because complexity of reasoning in these domain has very high complexity bounds and the (business) logical component is minimal. Hence, I shift such computations to highly-optimized procedural languages (Java) by procedural attachment (see section 4.2.3.1).
Further extensions, such as hypothetical reasoning with future events, which are implemented in ContractLog, can not described in detail in this dissertation. In short, the basic idea is to distinguish between *happened events* ($happens(e,t)$) and *future believed events* ($planned(e,t)$). This can be very effective in the context of hypothetical planning (e.g., to become proactive [PB05]) which can be viewed as the deduction of sentences of the form $plan \rightarrow goal$, e.g., $plan\{planned(x1,t1), planned(x2,t2), ...\}$ and $goal\{holdsAt\}$ where by deduction theorem, $Theory \models [Plan \rightarrow Goal]$ and $[Theory \wedge Plan] \models Goal$, planning can be done in terms of abduction, i.e., finding plans to add to the theory so that the goal is entailed.

4.6.4 Summary

Temporal reasoning on the effects of events and actions on changeable properties, so called fluents, is crucial to specify and track contract states and their relations in workflow-like settings. In this section I have described the core syntax and semantics of the event calculus implemented in the ContractLog KR. It provides a precise formal definition and declarative semantics and hence produces traceable and verifiable results. The formalization of the EC is given as a LP meta program which allows interpretation with different LP semantics and execution in standard LP inference engines. Hence, the EC logic in ContractLog provides adequate expressive power to specify sophisticated relations between events/actions and contract states and produces verifiable and traceable results. On the other hand, the EC logic in ContractLog is inherently goal-driven and provides no means to actively monitor occurred events and react to them. In the next two sections I will further extend and exploit the event calculus in the domain of active reaction rules and in the domain of deontic reasoning with deontic norms stated as fluents.

4.7 Reactive Behavioral Logic

In SLA execution and enforcement event-driven reactive functionalities are an obvious necessity. SLA rules often describe (re)active behavioral logic, e.g., "*if the service becomes unavailable (event) and it is not maintenance time (conditional state) then send a notification to the system administrator, create a trouble ticket and trigger escalation level one (complex action)*". Such rules typically follow the Event-Condition-Action (ECA) paradigm which has been developed and intensively explored in the active database domain (see section 3.5.1.1). In a nutshell, this paradigm states that an ECA rule autonomously and actively reacts to occurring events by evaluating a condition or a set of conditions and by executing a reaction whenever the event happens and the condition is true: "*on **Ev**ent and **Co**ndition do **Ac**tion*". In ContractLog I have implemented a tight integration of extended global ECA rules into logic programming in order to represent reaction rules in a homogenous syntax and knowledge base in combination with derivation rules and facts and use the backward-reasoning rule engine also as execution environment for reactive rules. A particular advantage of this approach is, that arbitrary complex functionalities of reaction rules can be implemented in terms of derivation rules in combination with the expressive other formalisms of the ContractLog KR.

Furthermore, ContractLog's reaction rule component intergrates the Prova Agent Architecture (Prova AA [KPS06, KS04]). The AA language includes constructs allowing for sending messages via tyical transport protocols such as JMS and allows for specifying message-oriented reaction rules for processing incoming evemt messages. In contrast to the global ECA rules which apply in a global context to detected (complex) events, the AA reaction rules apply local to a conversation / conversation set or process workflow, enabling typical process flows with splits and joins. That is, complex event processing is based on event messages which are interchanged between the enpoint nodes of the communication network (an ESB). I will detail the implementation in chapter 5 which describes the RBSLM tool. The ContractLog KR works on top of Prova AA reaction rules. Both global ECA rules and local messaging reaction rules are needed for distributed SLA management and enforcement and they form the basis for the integration of the ContractLog KR into languages such as BPEL and architectures such as EDAs, CEP and BAM systems. In the following subsections I will first describe the syntax and semantics of the ECA-LP extension of ContractLog which introduces global ECA rules and then describe the combination with AA reaction rules where event processing is done in a push mode using event notification and communication mechanisms.

4.7.1 Syntax of ECA-LP

The Event-Condition-Action logic programming (ECA-LP) [Pas05a] extension of the ContractLog language represents an extended ECA rule as a 6-ary truth-valued function $eca(Ti, Ev, Co, Ac, Po, EL)$, where Ti (time), Ev (event), Co

(condition), *Ac* (action), *Po* (post condition), *EL*(se) are complex term arguments. A complex term is a logical function of the form $c(C_1, .., C_n)$ with a bound number of arguments (terms) which might be constant, variable or again complex. That is, reactive rules are integrated as special functions into the (combined) signature $\overline{S}$ of ContractLog returning a truth value and taking arguments which are defined over the language of ContractLog. Boolean-valued procedural attachments, as defined in section 4.2.3.1, are also supported in ECA rules and can be directly used instead of a complex term. While the *Ev*, *Co* and *Ac* parts of an ECA rule comply with the typical definitions of standard ECA rules, the *Ti*, *Po* and *EL* part are extensions to standard ECA rules:

- The *time part* (*Ti*) of an ECA rule defines a pre-condition (an explicitly stated temporal event) which specifies a specific point in time at which the ECA rule should be processed by the ECA processor, either absolutely (e.g., "at 1 o'clock on the 1st of May 2006), relatively (e.g., 1 minute after event X was detected) or periodically (e.g., "every 10 seconds").
- The *post-condition* (*Po*) is evaluated after the action. It might be used to prevent backtracking from different variable bindings carrying the context information from the event or condition part by setting a cut. Or, it might be used to apply verification and validation tests using integrity (action) constraints or test cases which must be satisfied after the action execution.
- The else part (*EL*) defines an alternative action which is execute alternatively in case the ECA rule can not be applied, e.g., to specify a default action or trigger some failure handling (re-)action.

ECA parts might be left out (i.e., always true) stated with "_", e.g., $eca(time(...), event(...), \_, action(...), \_, \_)$ or omitted, e.g., $eca(e(...), c(...), a(...))$. This leads to specific types of reactive rules, e.g. *production rules* (CA: $eca(condition(), action())$) or *extended ECA rules with post condition* (ECAP: $eca(event(), condition(), action(), postcondition())$) or reactive rules in if-then-else style (EAA: $eca(event(...), action(...), else(...))$. During interpretation the smaller rule variants are expanded to the full 6-ary ECA rule syntax, where the omitted parts are automatically stated as true with "_".

The complex terms are interpreted as queries/goals on derivation rules which are used to implement the respective functionality of each of the ECA rules' parts. That is, the full expressiveness of derivation rules in extended LPs with logical connectives, variables, finite functions, (non-monotonic) default and explicit negation as well as linear sequential operators such as "cuts", serial updates and procedural attachments can be used to describe complex behavioral reaction logic in term of derivation rules which is actively queried from the reactive ECA rules. As a result the ECA rules' syntax stays compact, reuses the extended logic programming language and the implemented global derivation rules can be reused several times in different ECA rules, leading to a compact homogenous KB.

Example 26 *Every 10 seconds it is checked (time) whether there is a service request by a customer (event). If there is a service request a list of all currently unloaded servers is created (condition) and the service is loaded to the first server (ac-tion). In case this action fails, the system will backtrack and try to load the service to the next server in the list. Otherwise it succeeds and further backtracking is prevented (post-condition cut) . If no unloaded server can be found, the else action is triggered, sending a notification to the customer.*

```
eca(
    every10Sec(), % time
    detect(request(Customer, Service),T), % event
    find(Server), % condition
    load(Server, Service), % action
    !, % postcondition
    notify(Customer, "Service request temporarily rejected").
).

% time
every10Sec() :-  sysTime(T), interval( timespan(0,0,0,10),T).

% event
detect(request(Customer, Service),T):-
    occurs(request(Customer,Service),T),
    consume(request(Customer,Service)).

% condition
find(Server) :- sysTime(T), holdsAt(status(Server, unloaded),T).

% action
load(Server, Service) :-
    sysTime(T),
    rbsla.utils.WebService.load(Server,Service),
    add(key(Server), "happens(loading(_0),_1).",[Server, T]).

% alternative action
notify(Customer, Message):-
        sendMessage(Customer, Message).
```

The state of each server might be managed by an EC formalization:

```
terminates(loading(Server),status(Server,unloaded),T).
initiates(unloading(Server),status(Server,unloaded),T).
```

The example includes possible backtracking to different variable bindings. In the condition part all server which are in the state *unloaded* are bound to the

variable *Server*. If the action which tries to load a server with the service succeeds further backtracking is prevented by the post-conditional cut. If no unloaded server can be found for the customer request, the "else" action is executed which notifies the customer.

4.7.2 Declarative Semantics of ECA-LP

The declarative semantics of ECA rules in ContractLog LPs is directly built on top of the semantics of the underlying rule/inference system. ECA rules are defined globally and unordered in the KB; homogeneously in combination with other rule types. In order to integrate the (re)active behavior of ECA rules into goal-driven backward-reasoning the goals defined by the complex terms (truth valued functions) in the ECA rules are actively used to query the KB and evaluate the derivation rules which implemented the functionality of the ECA rules' parts.

Definition 133 ***(Reaction Rule)*** *A reaction rules is an extended ECA rule which is interpreted as a conjunction of (sub)goals (the complex terms) which must be processed in a left-to-right order starting with the goal denoting the time part, in order to capture the forward-directed operational semantics of an ECA rule:* $ECA_i? = Ti \wedge Ev \wedge ((Co \wedge Ac \wedge Po) \vee EL)$*, where* $ECA?$ *is the top goal/query which consists of the subgoal Ti,Ev,Co,Ac,Po, EL. An ECA rule succeeds, i.e., is entailed in the KB, if the subgoals succeed:*

$SEM(KB) \models ECA_i$ *iff* $SEM(KB) \models (\forall \overline{X})(Ti \wedge Ev \wedge ((Co \wedge Ac \wedge Po) \vee EL))$*, where* $\overline{X}$ *is a set of variables.*

The semantics for the respective parts of an ECA rule is defined by the semantics of the ContractLog formalisms. In subsection 4.7.4 I will elaborate on the complex event/action processing capabilities of ContractLog which allow complex events and actions in reaction rules. Note, that the "else action" EL is an alternative to the normal action sequence leading to an "if-then-else" rule style represented as a disjunction. The post-condition acts as a constraint on the KB state after the action has been performed. In particular, actions with effects on the KB such as knowledge updates (as described in section 4.5) which transit the actual KB state to the next state can be tested by integrity constraints as described in section 4.4.2. In case the integrity tests fail transactional knowledge updates are rolled back by the semantics of the transaction logic in ContractLog. In case of external actions compensating actions can be called, if the external system provides respective API methods which support transactions. That is the action part of a reaction rule only succeeds if the (pre)condition before the action and the postcondition after the actions are true. Formally:

$\forall \overline{X}(Co \wedge Ac \wedge Po)$

4.7.3 Operational Semantics of ECA-LP

In order to integrate the (re)active behavior of ECA rules into goal-driven backward-reasoning the goals defined by the complex terms in the ECA rules are meta-interpreted by an additional ECA interpreter. The interpreter implements the forward-directed operational semantics of the ECA paradigm. The ECA interpreter provides a general Wrapper interface which can be specialized to a particular query API of an arbitrary backward-reasoning inference engine. That means, the ECA meta interpreter is used as a general add-on attached to a LP system extending it with reasoning and processing features for reactive rules. The task of processing an ECA rule by querying the respective derivation rules using the defined complex terms in an ECA rule as queries on the KB is solved by a Daemon (implemented within the ECA interpreter). The daemon is a kind of adapter that frequently issues queries on the ECA rules in order to simulate the active behavior in passive goal-driven LP systems. Proof-theoretically it applies the ECA subgoals of a top query formed by an ECA rule one after the other on the KB (the actual KB state) using the inference rules of the underlying backward-reasoning inference engine to deductively prove the syntactic drivability from the clauses in the KB. The process is as follows:

1. it queries (repeatedly - in order to capture updates to reactive rules) the KB and derives all ECA rules represented in the KB by the universal query $eca(Ti, Ev, Co, Ac, Po, EL)?$,
2. it adds the derived ECA rules to its internal active KB, which is a kind of volatile storage for reactive rules and temporal event data, and
3. finally, it processes the ECA rules sequentially or in parallel depending on the configuration using a thread pool.

The forward-directed execution of ECA paradigm is given by the strictly positional order of the terms in the ECA rules. That is, first the time part is queried/evaluated by the ECA processor (daemon), when it succeeds then the event part is evaluated, then the condition and so on. The computed (ground) substitutions θ of the variables for each subgoal in a rule ECA_i are unified by the ECA interpreter with their variable variants in the subsequent subgoals of the top ECA query. The interpreter also implements the common LP backtracking mechanism to backtrack from different variable bindings. Figure 4.1 illustrates this process.

In order to enable parallel processing of ECA rules the ECA processor implements a thread pool where each ECA rule is executed in a separated thread, if its time part succeeds (see figure 4.2).

Variables and negation in ECA rules are supported as described in section 4.1.2. For example, the ECA processor enables variable binding to ground knowledge derived from the queried derivation rules and facts of the KB, knowledge transfer from one ECA part to another by variable unification and backtracking to different variable bindings as in logic programming. This is in particular useful to interchange context information, e.g., between the event and the

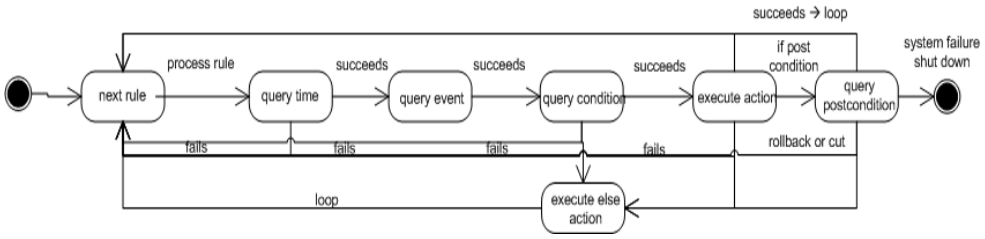

Figure 4.1: Forward-directed Execution Model of Reaction Rules

Figure 4.2: Parallel Processing of Reaction Rules with Threads

action or the condition and the action part. It is worth noting, that the implementation of the ECA interpreter is designed to be general and to be applicable to different derivation mechanisms such as variants of linear SLDNF-resolution. However, depending on the configuration of the ECA interpreter for a particular logic class (definite, stratified, normal, extended, disjunctive LP) and depending on the procedural semantics of the underlying inference engine, e.g., standard SLDNF with negation-as-finite-failure test rule (Naf), query answering might become undecidable (due to infinite functions), non-terminating (due to loops) or floundering (due to free variables in Naf tests). Hence, depending on the underlying logic this amounts for configuring the ECA interpreter with special safety conditions such as Datalog restriction or "allowedness" restriction where variables in default negated literals are not allowed.

The homogeneous combination of reaction rules and derivation rules within the same representation language paves the way to (re-)use various other useful logical formalisms in active rules, such as procedural attachments, defeasible rules with rule priorities for conflict resolution, transactional update actions, event/action logics for temporal event calculations and complex event/action reasoning and it relates ECA rules to other rule types such as integrity constraints or normative rules. As a result, the high expressive power and the clear logical semantics of these formalisms is also adopted for reaction rules. In particular, representation and processing of complex events and actions and active rules, which are rules which actively trigger other reaction rules leading to non-deterministic execution sequences, benefits from this logical treatment, as I will describe in the next section.

4.7.4 Complex Event / Action Processing

Events resp. actions in reactive ECA rules are typically not atomic but are complex consisting of several atomic events or actions which must occur in the defined order and quantity in order to detect complex events resp. execute complex actions. This topic has been extensively studied in the context of active databases and event algebras, which provide the operators to define complex event types (see section 3.5.1.1 for an overview). Typical event algebras in the active database domain define the following operations or variants of them:

- Sequence operator (;): the specified event instances have to occur in the order determined by this operator
- Disjunction operator ($\vee$): at least one of the specified instances has to occur
- Conjunction operator ($\wedge$): the specified event instances can occur in any order, where the detection time is the timestamp of the latest occurred event
- Simultaneous operator (=): the specified instances have to occur simultaneously
- Negation operator ($\neg$): the specified instance(s) are not allowed to occur in a given interval
- Quantification (Any): the complex event occurs when n events of the specified type have occurred
- Aperiodic Event Operator (Ap): The aperiodic operator Ap allows one to express the occurrence of an event $Ev2$ within the interval defined by two other events $Ev1$ and $Ev3$: $Ap(Ev2, Ev1, Ev3)$
- Periodic Event Operator (Per): The periodic operator Per is an event type which occurs every t time-steps in between $Ev1$ and $Ev2$

The detection time of a complex event is typically the occurrence time of its terminating event (according to the defined selection and consumption policy for events). This leads to inconsistencies and irregularities in typical event algebras, such as Snoop [CKAK94], SAMOS [GD93] as I will illustrate now:

1. Consider the following event type $Ap(A, B, C)$ defined by the aperiodic operator Ap in Snoop, i.e., the complex event is detected when A occurs between B and C. An event instance sequence (EIS) $\{acbb\}$ will trigger two events of this type, according to the type specific instance sequence $Ap(A, B, C) = \{\{a, b\}, \{a, b\}\}$, even though the event instances of B occur outside of the detection interval defined by a and c.
2. The sequence $B; (A; C)$ in Snoop is detected if A occurs first, and then B followed by C, i.e., $EIS = \{a, b, c\}$, because the complex event $(A; C)$ is detected with associated detection time of the terminator c and accordingly the event b occurs before the detected complex event $(A; C)$. But

the intended result is that only $EIS = b, a, c$ should lead to the detection of the complex event $B;(A;C)$ since the correct event type pattern for $EIS = a, b, c$ should be $A;(B;C)$.

3. Consider the event type $(A;B)[A,C]$ in SAMOS, i.e., the event of type A sequentially followed by B in between A and C causes the complex event to be detected. For an $EISaabbbc$ one may expect (under a strict interpretation) that it does detect one or at least two complex events, because the sequence $(a;b)$ occurs once (or twice) in the interval determined by a and c. However, the event detection algorithm used in SAMOS detects three events of this type, because it combines all occurrences of a and b in between $[A,C]$ regardless of the sequence operator and occurrence intervals of $[A,B]$ in between $[A,C]$. Although this might be acceptable in some domains, under a strict interpretation it violates the semantics of the sequence operator and the usual event consumption, where events do not contribute to several event detections but are consumed after they have contributed to a complex event.

4. Given the event type $(A;B)$ ACCOD (Active Object Oriented Database System) causes the recognition of six complex events from the EIS: $\{aaabb\}$. While this may be acceptable, or even desirable, in some applications, strictly speaking, if the events unify with the detection interval that is already progressing, i.e., which is initiated by a, multiple repeated occurrences of a should be ignored during the monitoring interval $[a,b]$.

All these problems arise from the fact that complex events, in the active database sense, are simply detected and treated as if they occur at an atomic instant, i.e., at a certain time point, and not in the KR event/action logics sense, where complex events occur over an extended interval and where the information about how far into the past their occurrence interval (maximum validity interval) reaches is derived. To overcome such unintended semantics and provide verifiable and traceable complex event computations (resp. complex actions) based on a logical calculi I have implemented an interval-based Event Calculus (EC) variant in the ContractLog KR and refined the typical database event algebra operators based on it. In the interval-based Event Calculus all events are regarded to occur over a time interval, i.e., an event interval $[e1,e2]$ occurs during the time interval $[t1,t2]$ where $t1$ is the occurrence time of the initiator event $e1$ and $t2$ is the occurrence time of the terminator event $e2$. An atomic event occurs in the interval $[t,t]$, where t is the occurrence time of the atomic event. The basic *holdsAt* axiom used for temporal reasoning about fluents is redefined to the axiom $holdInterval \subseteq \overline{Ev} \times \overline{Ev} \times \overline{Ti} \times \overline{Ti}$: $holdsInterval([Ev1,Ev2],[Ti1,Ti2])$ to capture the semantics of event intervals which hold between a time interval:

```
holdsInterval([Ev1,Ev2],[Ti11,Ti22]):-
      event([Ev1],[Ti11,Ti12]), event([Ev2],[Ti21,Ti22]),
      [Ti11,Ti12]<=[Ti21,Ti22], not(broken(Ti12,[Ev1,Ev2],Ti21).
```

The event function *event* is a meta function to translate instantaneous event occurrences into interval-based events: $event([Ev],[Ti,Ti]) : -occurs(Ev,Ti)$. It is also used in the event algebra meta program to compute complex events from occurred raw events according to their event type definitions. The *broken* function tests whether the event interval is not broken between the initiator event and the terminator event by any terminating other event.

Based on this interval-based event logics formalism, I now redefine the typical (SNOOP) event algebra operators and treat complex events resp. actions as occurring over an interval rather than in terms of their instantaneous detection times. In short, the basic idea is to split the occurrence interval of a complex event into smaller intervals in which all required component events occur, which leads to the definition of event type patterns in terms of interval-based event detection conditions, e.g., the sequence operator (;) is formalized as follows:

Example 27

```
(A;B;C): detect(e,[Ti1,Ti3]) :-
            holdsInterval([a,b],[Ti1,Ti2],[a,b,c]),
            holdsInterval([b,c],[Ti2,Ti3],[a,b,c]),
            [Ti1,Ti2]<=[Ti2,Ti3].
```

Using the holdsInterval axioms the typical event algebra operators are formalized in terms of the interval-based EC formulation as follows: "

- Sequence operator (;): $(A;B;C)$

  ```
  detect(e,[T1,T3]) :-
      holdsInterval([a,b],[T1,T2],[a,b,c]),
      holdsInterval([b,c],[T2,T3],[a,b,c]), [T1,T2]<=[T2,T3].
  ```

- Disjunction operator ($\vee$): $(A \vee B)$

  ```
  detect(e,[T1,T2]) :-
      holdsInterval([a],[T1,T2]).
  detect(e,[T1,T2]) :-
      holdsInterval([b],[T1,T2]).
  ```

- Mutual exclusive operator (xor): $(AxorB)$

  ```
  detect(e,[T1,T2]) :-
      holdsInterval([a],[T1,T2]),
      not(holdsInterval([b],[T3,T4]).
  detect(e,[T1,T2]) :-
      holdsInterval([b],[T1,T2]),
      not(holdsInterval([a],[T3,T4]).
  ```

- Conjunction operator ($\wedge$): $(A \wedge B)$

```
detect(e,[T1,T2]):-
    holdsInterval([a],[T11,T12]),
    holdsInterval([b],[T21,T22]), min([T11,T12,T21,T22],T1),
    max([T11,T12,T21,T22],T2).
```

- Simultaneous operator (=): $(A = B)$

```
detect(e,[T1,T2]) :-
    holdsInterval([a],[T1,T2]),
    holdsInterval([b],[T1,T2]).
```

- Negation operator ($\neg$): $(\neg B[A, C])$

```
detect(e,[T1,T2]) :-
    holdsInterval([a,c],[T1,T2],[b]).
```

- Quantification (Any): $(Any(n, A))$

```
detect(e,[T1,T2]):-findall([T1,T2],
holdsInterval([a],[T1,T2],List), modulo(List, n, [T1,T2]).
```

  It is possible to define the any operator as a conjunction of all events. However, this results in long event expressions. Therefore, I use the second-order predicate "findall", which derives all occurrence intervals of a and collects them in a list, which is the type specific event instance sequence (EIS) of all events of type A. The modulo predicate iterates over the complete list and returns all elements where size of the list modulo n is null, i.e., size mod $n = 0$.

- Aperiodic Event Operator (Ap): $Ap(A, [B, C])$

```
detect(e,[T1,T2]):-
    holdsInterval([b,c],[T11,T12],[b,c]),
    event([a],[T1,T2]), between([T1,T2],[T11,T12]).
```

- Periodic Event Operator (Per): $Per(A, t, , C)$
  The periodic event operator can be efficiently represented as an reaction rule which defines a periodic time interval function t which is valid during the occurrence interval of [a,c].

In order to make definitions of complex events/actions in terms of event algebra operators more comfortable and remove the burden of defining all interval conditions for a particular complex event type, as described above, I have implemented a meta program which implements an interval-based EC event/action algebra in terms of typical event operators with the following axioms:

- Sequence: $sequence(E1, E2, .., En)$
- Disjunction: $or(E1, E2, .., En)$
- Mutual exclusive: $xor(E1, E2, .., En)$
- Conjunction: $and(E1, E2, .., En)$

- Simultaneous: $concurrent(E1, E2, .., En)$
- Negation: $neg([ET1, .., ETn], [E1, E2])$
- Quantification: $any(n, E)$
- Aperiodic: $aperiodic(E, [E1, E2]$

In order to reuse detected complex events in rules, e.g., in reaction rules or other complex events, they need to be remembered until they are consumed, i.e., the contributing component events of a detected complex event should be consumed after detection of a complex event. This can be achieved by the previously described ID-based update primitives (see section 4.5, which allow adding or removing knowledge from the KB. I use these update primitives to add detected event occurrences as new transient facts to the KB and consume events which have contributed to the detection of the complex event by removing them from the KB.

Example 28

```
detect(e,Ti):-
    event(sequence(a,b),Ti), % detection condition for the event e
    add(eis(e), "occurs(e,_0).", [Ti]), % add e with key eis(e)
    consume(eis(a)), consume(eis(b)). % consume all a and b events
```

In the example, if the detection conditions for the complex event e are fulfilled, the occurrence of the detected event e is added to the KB with the key $eis(e)$ (eis = event instance sequence). Then all events that belong to the type specific event instance sequences of type a and type b are consumed using their ids $eis(a)$ resp. $eis(b)$. Different consumption policies are supported such as "remove all events which belong to a particular type specific eis" or "remove the first resp. the last event in the eis". If no consume predicate is specified in a detection rule, the events are reused for the detection of other complex events several times.

4.7.5 Event Notification / Communication Reaction Rules

The ECA rules of ECA-LP apply in a global context, i.e. apply on detected (complex) events (detected by complex event queries on the derivation rule base as described in the previous section). Such global ECA rules are best suited to represent reation rules which actively detect internal and external events in a global context. For instance, actively monitor an external system or service for availability and trigger a reaction whenever the system/service becomes unavailable. In a distributed environment with independent system nodes which communicate with each other relative to a certain context (e.g. a workflow/conversation protocol state), event processing is often done using event notification and communication mechanisms. Systems either communicate events according to a predefined or negotiated communication/coordination protocol or they

subscribe to specific event types with a server. In the latter case, the server monitors its environment and upon detecting an atomic or complex event (situation), notifies the concerned clients. Complex events may correspond to pre-defined protocols or be based on event algebras including time restricted sequences and conjunctions/disjunctions, which permits events like A occurs more than t time after B to be expressed.

The Prova Agent Architecture (AA) [KPS06, KS04] implements a derivation rule related language that includes constructs for sending messages via various communication protocols and for specifying reaction rules for processing inbound messages. The ContractLog KR integrates Prova AA in order to provide event messaging reaction rules in addition to global ECA-style reaction rules. The AA reaction rules do not require separate threads for handling multiple conversation situations simultaneously. In the following I will first describe the main syntax constructs and features of AA and then give an example for the intergration of AA into ContractLog.

Prova-AA provides three main constructs for enabling agent communication: *sendMsg* predicates, reaction *rcvMsg* rules, and *rcvMsg* or *rcvMult* inline reactions [KPS06]:

```
sendMsg(XID,Protocol,Agent,Performative,Payload |Context)
rcvMsg(XID,Protocol,From,queryref,Paylod|Context)
rcvMult(XID,Protocol,From,queryref,Paylod|Context)
```

where *XID* is the conversation identifier (conversation-id) of the conversation to which the message will belong. *Protocol* defines the communication protocol. More than 30 protocols such as JMS, HTTP, SOAP, Jade are supported by the underlying ESB [Mul06]. *Agent* denotes the target of the message. *Performative* describes the pragmatic context in which the message is send. A standard nomenclature of performatives is FIPA Agents Communication Language ACL. *Payload* represents the message content sent in the message envelope. It can be a specific query or answer or a complex rule base (set of rules).

Example 29

```
% Upload a rule base read from File to the host
% at address Remote via Prova-JMS
upload_mobile_code(Remote,File) :-
    % Opening a file returns an instance
    % of java.io.BufferedReader in Reader
    fopen(File,Reader),
    Writer = java.io.StringWriter(),
    copy(Reader,Writer),
    Text = Writer.toString(),
    % SB will encapsulate the whole content of File
    SB = StringBuffer(Text),
    sendMsg(XID,jms,Remote,eval,consult(SB)).
```

The example [KPS06] shows a AA reaction rule that sends a rule base from an external *File* to the agent service *Remote* using JMS as transport protocol. The corresponding receiving reaction rule could be:

```
rcvMsg{XID,jms,Sender,eval,[Predicate|Args]):-
    derive([Predicate|Args]).
```

This rule receives all incoming JMS based messages with the pragmatic context *eval* and derives the message content. The list notation *[Predicate|Args]* will match with abitrary n-ary predicate functions, i.e., it denotes a kind of restricted second order notation since the variable *Predicate* is always bound, but matches to all predicates in the signature of the language with an abitrary number of arguments *Args*).

I will now illustrate the combination of active global ECA and passive messaging reaction rules by a typical use case found in industry:

Example 30

A Manager node is responsible for holding housekeeping information about various servers playing different roles. When a server fails to send a heartbeat for a specified amount of time, the Manager assumes that the server failed and coop-erates with the Agent component running on an unloaded node to resurrect it. An AA reaction rule for receiving and updating the latest heartbeat in event notification style is:

```
rcvMsg(XID,Protocol,FromIP,inform,heartbeat(Role,RemoteTime)) :-
    time(LocalTime)
    update(key(FromIP,Role),"heartbeats(_0, _1, _2, _3).",
                  [ FromIP, Role, RemoteTime, LocalTime] ).
```

The rule responds to a message pattern matching the one specified in the rcvMsg arguments. *XID* is the conversation-id of the incoming message; *inform* is the performative representing the pragmatic context of the message, in this case, a one-way information passed between parties; *heartbeat(...)* is the payload of the message. The body of the rule enquires about the current local time and updates the record containing the latest heartbeat from the controller. This rule follows a push pattern where the event is pushed towards the rule systems and the latter reacts. A pull-based global ECA rule that is activated every second by the rule engine and for each server that fails to have sent heartbeats within the last second will detect server failures and respond to it by initiating failover to the first available unloaded server. The accompanying derivation rules *detect* and *respond* are used for specific purpose of detecting the failure and organising the response.

```
eca(
    every('1S') ,
    detect(controller_failure(IP,Role,'1S')) ,
    respond(controller_failure(IP,Role,'1S')) ) .

every('1S'):-
    sysTime(T),
    interval(timespan(0,0,0,1),T).

detect(controller_failure(IP,Role,Timeout)) :-
    sysTime(LocalTimeNow),
    heartbeats(IP,Role,RemoteTime,LocalTime),
    LocalTimeNow-LocalTime > Timeout.

respond(controller_failure(IP,Role,Timeout)) :-
    sysTime(LocalTime),
    first(holdsAt(status(Server,unloaded),LocalTime)),
    add(key(Server),
        "happens(loading(_0),_1).",[ Server, Local-Time]),
    sendMsg(XID,loopback,self,initiate,failover(Role,IP,Server)).
```

The ECA logic involves possible backtracking so that all failed components will be resurrected. The state of each server is managed via an event calculus formulation:

```
initiates(loading(Server),status(Server,loaded),T).
terminates(unloading(Server),status(Server,loaded),T).
initiates(unloading(Server),status(Server,unloaded),T).
terminates(loading(Server),status(Server, loaded),T).
```

The actual state of each server is derived from the happened loading and unloading events and used in the ECA rule to detect the first server which is in state *unloaded*. This EC based formalization can be easily extended, e.g. with new states such as a maintenance state which terminates an unloaded state, but is not allowed in case a server is already loaded:

```
initiates(maintaining(Server),status(Server,maintenance),T):-
    not(holdsAt(status(Server,loaded),T)).
terminates(maintaining(Server),status(Server,unloaded),T).
```

As it can be already seen from this examples further, higher-level decision logic and process oriented logic relating to state machines- or workflow-like logic might be easily implemented using a combination of global active and passive messaging reaction rules. In summary, the messaging style AA reaction rules complement the global ECA rules and the Event Calculus complex event algebra.

4.7.6 Summary

I have described a general extension of the ContractLog KR for the interpretation and execution of extended global active ECA rules in arbitrary backward reasoning rule engines and combined it with passive messaging reaction rules. Reaction rules of the various types can be represented in a homogenous KB in combination with other rule types. The logic of each integral part is decoupled from the reaction rule and represented in terms of derivation rules, leading to a compact global representation which facilitates modularization and reuse. The high expressiveness and declarative semantics of LPs and LP based formalisms are reused to describe complex behavioral reaction logic and typical process and workflow semantics such as pi-calculus, petri-nets or state machines can be implemented using AA reaction rules. Model-theoretically an ECA reaction rule is interpreted as a conjunctive query (with the alternative action as disjunction) which succeeds if each subgoal is entailed in the KB. An AA reaction rule applies event queries and LP based pattern matching (unification of complex functions and terms) on event messages which trigger the reactive behaviour. Variables for transporting the event/action data, Boolean-valued procedural attachments, built-ins as described in section 4.2.3.1 and serial transactional updates as described in section 4.5 which return a true/false value are directly supported in reaction rules. This allows to actively call external systems and their API methods, e.g., to implement push or pull models for events from external sources such as event notification systems, databases, network or system management tools and to trigger external API methods as actions. The operational semantics provides means to execute global ECA reaction rules sequentially or in parallel in different threads or wait passively for incoming event messages which match with the defined conversation state, pragmatic context and event pattern in case of AA reaction rules. The semantics implements the standard LP backtracking mechanisms for variable bindings within the forward-directed operational semantics of the ECA interpreter with support for cuts in the post condition part. Furthermore, the (pre-)condition and post condition part in global ECA rules can be used to apply action constraints and integrity constraints with rollbacks of transactional updates as described in section 4.5.

Standard reaction rules and complex event processing techniques as used in active database or production rule systems are mainly intended for immediate reactions on atomic or complex events which are defined by means of a complex event algebra. Typical active database systems only provide an operational semantics, but lack a declarative semantics which often leads to unintended results. The tight integration of reaction rules into logic programming in ContractLog and Prova AA allows to unify the major approaches for complex event processing and reaction rules, namely global ECA rules from active databases, event notification and messaging reaction rules from distributed complex event processing and event/action logics from the KR domain. In particular, I have developed an event algebra based on an interval-based Event Calculus variant which bases complex event and action processing on a precise declarative semantics producing highly reliable and verifiable results.

In summary the main benefits of this integration of reaction rules into logic programming are:

- based on a precise, declarative logic-based semantics
- highly expressive and computationally efficient
- reaction rules might be represented and used in combination with other rule types, such as derivation rules (e.g., business rules) or integrity constraints, within the same framework of logic programming
- might be easily combined and extended with other logical formalisms (e.g., defeasible/default for conflict handling) to build reliable high-level decision logics upon.
- the inference engine is used as generic interpreter to detect complex events and enable immediate but also delayed and inferred complex reactions and effects
- the ECA interpreter works as an add-on for arbitrary backward-reasoning engines
- the integration of messaging reaction rules provides means to communicate events and apply local reactions according to certain process and coordination protocols (e.g. with mappings to BPEL specifications)

4.8 Deontic Logic

One of the main objectives of a SLA is to define and reason with the normative relationships relating to permissions, obligations and prohibitions between contract partners, i.e., to define the rights and obligations each role has in each particular state of the contract. Deontic Logic (DNL) studies the logic of normative concepts such as *obligation* (O), *permission* (P) and *prohibition* (F). Adding deontic logic is therefore a useful concept for SLM tools, in particular wrt traceability and verifiability of derived contract norms (rights and obligations). Unfortunately, standard deontic logic (SDL) [FH71] offers only very a static picture of the relationships between co-existing norms and does not take into account the effects of events on the given norms, temporal notions and dependencies between norms, e.g., violations of norms or exceptions. Another limitation is the inability to express personalized norms, i.e., explicitly define the subject and object a certain norm pertains to. In this section I integrate deontic logic into the event calculus such that deontic norms are modelled as changeable fluents which are initiated or terminated by events. The main advantages are:

- reified fluents can be used to represent complex norms
- complex (temporal) relations between norms and effects of events/actions such as violations or exceptions of norms can be naturally represented

- the event calculus is used to compute all norms which hold at a particular time point wrt the sequence of happened events/actions
- retrospective and prospective reasoning on norms is possible

I first review the history and basic concepts of deontic logic. Then I describe the syntax and the semantics of the EC based deontic logic.

4.8.1 History and Basics in Deontic Logic

Deontic Logic (DNL) studies the logic of obligation, permission and prohibition. A first theory of normative concepts was given by Mally [Mal26], but most applications of DNL such as standard deontic logic (SDL) [FH71] are derived from the work of Wright [Wri51]. Several computational tractable inference mechanisms for DNL have been proposed, e.g. [Bel87, McC83] The basic operator in SDL is O, which reads as "it is obligatory that". O is a mapping of a well-formed formula A into another well-formed formula OA. The other operators P (permission) and F (forbid) are defined in terms of O as $PA \equiv \neg O \neg A$ resp. $FA \equiv O \neg A$, i.e., A is permitted if $\neg A$ is not obligatory and A is forbidden if $\neg A$ is obligatory. The axioms of SDL are:

1. If A is a theorem, then so is OA
2. All classical tautologies
3. $O(A \rightarrow B) \rightarrow (OA \rightarrow OB)$
4. $OA \rightarrow PA$
5. If A and $A \rightarrow B$ are theorems, then so is B (modus ponens)

Usually SDL is given a Kripke style modal semantics, defined by accessibility relations between possible worlds, e.g. [Han65, Mak86, Aqu87, AS96]. But, among others, Horty [Hor93] and McCarthy [McC92] have proposed nonmonotonic logic and defeasible obligations (aka prima facie obligations) as a theoretical formalism for deontic reasoning, which can handle some of the paradoxes of SDL, i.e., set of sentences that derive sentences with a counterintuitive reading [Cas81]. One source for such problems are violated obligations which lead to so called contrary-to-duty obligations (CTDO), i.e., secondary obligations which arise in case primary obligations are violated [Hag01, CJ02, Lew74]: $A_1 \rightarrow OA_2$, where A_1 is the condition which violates the primary obligation $O \neg A_1$ and triggers the CTDO OA_2. In SLAs such situations typically arise. For instance, an obligation to restart a service in a certain time frame is violated and initiates a secondary CTDO to pay a penalty for this violation. In SDL, both norms the primary obligation and the CTDO hold which might lead to contradictions between the two obligations, e.g., both OA and $O \neg A$ can be derived which leads to inconsistency in SDL. Well-known examples of paradoxes wrt CDTOs and conditional norms are e.g., Forrester's gentle murderer paradox, the Reykjavic paradox or the Chrisholm paradox.

Besides these paradoxes which are specific to violations another source are defeasible obligations which are subject to exceptions, where defeasible obligations are overtuned by more specific ones in case of exceptions: normally OA_1 holds but in case of Ev the exceptional norm $Ev \rightarrow OA_2$ holds, which again can lead to contradictions and hence inconsistency in SDL. As discussed in the beginning exceptional situations which lead to different norms are also quite common in the SLA domain. Many systems and intuitions to the different problems of SDL have been proposed, but a generally accepted solution for all paradoxes is still missing. In ContractLog I deal with the violation and exception problem by combining deontic logic with the temporal event calculus. That is, norms are considered as changeable states which are initiated or terminated by events (i.e., action events). In case a violation or exception event happens the primary norm is terminated and the secondary reparations or exceptional norm is initiated. Hence, there is never a situation where both conflicting norms are valid.

Another problem of SDL in the context of contract representation is that it offers only a static picture of the relationships between co-existing norms and does not take into account the effects of events on the given norms. That is norms are not considered as changeable states which are initiated or terminated by action events. Moreover, in SDL it is impossible to express personalized statements. In the SLA domain deontic norms refer to an explicit concept of an agent, e.g., the service provider or the service consumer. These limitations make it difficult to satisfy the needs of practical contract management. In the event calculus norms are formalized as changeable complex functions which can take arguments.

4.8.2 Syntax of Event Calculus based Deontic Logic

In ContractLog I extended the general concepts of SDL and integrated it into the event calculus in order to model the effects of events/actions on personalized deontic norms. A deontic norm in ContractLog consists of the normative concept (norm), the *subject* ($Subj$) to which the norm pertains, the *object* (Obj) on which the action is performed and the *action* (Ac) itself. A deontic norm is represented as a reified EC fluent of the form: $norm(Subj, Obj, Ac)$. Accordingly, norms in ContractLog are modelled as reified complex functions which may be negated, may contain variables and typed terms such as object-oriented composite structures or DL classes, e.g., roles from a role model described as a Semantic Web ontology.

Example 31

```
initiates(unavailable(Server), escl(1),Ti).
terminates(available(Server), escl(1),Ti).
initiates(maintaining(Server),status(Server,maintenance),Ti).
terminates(maintaining(Server),escl(1),Ti).
derived(oblige(processManager, Service, restart(Service))).
```

```
holdsAt(oblige(processManager, Service, restart(Service)),Ti):-
holdsAt(escl(1),Ti).
```

In the example escalation level 1 is initiated resp. terminated, when a service becomes unavailable resp. available, e.g., $happens(unavailable(s1), t1)$. The deontic obligation for the process manager to restart the service is defined as a derived fluent, i.e., it holds whenever the state $escl(1)$ holds. If the process manager is permitted to start maintenance (e.g., between 0 a.m. and 4 a.m. - not shown here) the second and third rule state that the event maintaining(Server) will initiate maintenance and terminate escalation level 1.

4.8.3 Semantics of Event Calculus based Deontic Logic

The declarative semantics of the deontic logic implementation in ContractLog is directly inherited from the semantics of the EC formalization 4.6.3.

Definition 134 *(**Deontic Norm Satisfaction**) Let N be a deontic norm. An EC interpretation I satisfies a deontic norm N (fluent) at timepoint Ti if $I(N, Ti) = true$ and $I(\neg N, Ti) = false$.*

Definition 135 *(**Deontic Norm Entailment**) Let D^{DNL} be a DNL domain description. A norm N holds at a timepoint Ti wrt to D^{DNL}, written $D^{DNL} \models holdsAt(N, Ti)$, iff for every interpretation I of D^{DNL}, $I(N, Ti) = true$ and $I(\neg N, Ti) = false$. $D^{DNL} \models neg(holdsAt(N, Ti))$ iff $I(N, Ti) = false$ and $I(\neg N, Ti) = true$.*

In addition to the EC meta inference rules (see section 4.6) the typical SDL axioms are defined as an EC meta axiomatization, e.g., $OA \Rightarrow PA$: $holdsAt(permit(S, O, A), T) : -holdsAt(oblige(S, O, A), T)$.

The EC formalization by default implies a positive authorization policy where by default everything is forbidden naturally unless it is explicitily initiated by a happened event/action. But negative authorizations where a norm holds initially ($initially(N, Ti)$) can be also formulated, i.e., everything is permitted until the prohibition is explicitly terminated by an event.

Conditional deontic rules such as $A1 \Rightarrow OA2$ are naturally captured by the domain axiomatization of the EC.

Example 32

```
happens(A1,t1).
initiates(A1,oblige(S,O,A2),T).
```

In the example the event $A1$ initiates the obligation for $A2$. As discussed conditional deontic rules in SDL might lead to inconsistencies and paradoxes. Due to the temporal event logic in ContractLog most of these conflicts in SDL such as situations where a violated primary obligation and a secondary CTD obligation are true at the same time can be avoided by terminating the violated primary obligation so that only the consequences of the violation (CTD obligation) are in effect.

Example 33 $(\neg A1 \wedge OA1) \Rightarrow OA2$

```
happens(violation,T) :-
    happens(neg(a1),T), holdsAt(oblige(s,o,a1),T).
initiates(violation, oblige(s,o,neg(a1)),T).
terminates(violation, oblige(s,o,a1),T).
```

In the example a violation is derived if the action $a1$ is obliged but in fact $neg(a1)$ happens. The violation event then terminates the primary obligation for $a1$ and initiates the secondary CTD obligation for $neg(a1)$. Hence, there is never a situation where both obligations hold at the same time point and the valid sceme of SDL $\neg(OA \wedge O\neg A)$ is preserved.

Other problematic examples in SDL are defeasible prima facie obligations which are subject to exceptions and might lead to contradictions for example between OA and the exceptional obligation $O\neg A$ which holds in a execptional situation. Again the temporal treatment of deontic norms as changebale fluents can be used to overcome this inconsistency and establish the exceptional norm in case of the exceptional situation.

Example 34

```
happens(Ex,t1).
terminates(Ex,oblige(s,o,a),T).
initiates(Ex,oblige(s,o,neg(a)),T).
```

In the example the exceptional event Ex terminates the general obligations and initiates the conditional more specific obligation till the exceptional situation is terminated by another event (not shown in the example).

Note that there are known time-less paradoxes in SDL such as Forrester's gentle murder paradox which can not be solved by the temporal EC-based treatment as described above. In this case defeasible deontic rules with defined priorities between conflicting norms as described in section 4.4.3 can be used. That is the notion of implication is weakened in such a way that the counterintuitive norms are no longer derived. However, such time-less paradoxes rarely occur in service contracts, since usually the rights and obligations are defined in a time-based and situational context.

4.8.4 Summary

In this section I have embedded deontic logic into the temporal event calculus formalism of the ContractLog KR. The treatment of deontic norms as reified fluents allows expressing complex norms with possibly nested and typed arguments, e.g., in order to use external role models which are represented as Semantic Web ontologies. The EC formalization provides a sound and complete declarative semantics for deontic reasoning where norms are true or false at a particular time point according to the happened events/actions which have an effect on the norm. As a consequence, the produced results are highly reliable and traceable which is crucial in the domain of legal reasoning and contract enforcement. Moreover, the particular contract state and the right and obligations in each state can be derived in retrospective for each time point or even in prospective, which makes possible proactive planning with future events. This KR treatment of events/actions and their effects on changeable norms is different from the active processing of events and actions in reaction rules, where occurred events trigger actions are consumed afterwards, i.e., their information for later use, e.g., in accounting, is lost. The temporal treatment of deontic norms in the EC helps to overcome typical paradoxes of SDL and allows specifying complex, conditional deontic rules with possibly concurrent state transitions between norms. It is possible to define reparational norms which apply in exceptional situations or CDT obligations which apply as secondary norms of violated primary obligations. Typical examples in the SLA domain are e.g., obligations to restart a service in a certain time frame or to provide an extra service during an exceptional campaign of a customer. With this exact contract state tracking during monitoring of SLAs becomse possible.

4.9 Test Logic

Rule-based policy and contract systems need to be studied in terms of their software engineering properties. The domain imposes some specific needs on the engineering and life-cycle management of formalized policy / contract specifications: The contract/policy rules must be necessarily modelled evolutionary, in a close collaboration between domain experts, rule engineers and practitioners and the statements are not of static nature and need to be continuously adapted to changing needs. The future growth of policies or contract specifications, where rules are often managed in a distributed way and are interchanged between domain boundaries, will be seriously obstructed if developers and providers do not firmly face the problem of quality, predictability, reliability and usability also w.r.t. understandability of the results produced by their rule-based policy/contract systems and programs. Furthermore, the derived conclusions and results need to be highly reliable and traceable to count even in the legal sense. This amounts for verification, validation and integrity testing (V&V&I) techniques, which are much simpler than the rule based specifications itself, but nevertheless adequate (expressive enough) to approximate their intended semantics, deter-

mine the reliability of the produced results, ensure the correct execution in a target inference environment and safeguard the life cycle of possibly distributed and unitized rules in rule-based policy projects which are likely to change frequently. In other words, V&V&I of rule bases is vital to assure that the LP used to represent a contract or policy, performs the tasks which it was designed for.

Different approaches and methodologies to V&V of rule-based systems have been proposed in the literature such as model checking, code inspection or structural debugging. Simple operational debugging approaches which instrument the policy/contract rules and explore its execution trace place a huge cognitive load on the user, who needs to analyze each step of the conclusion process and needs to understand the structure of the rule system under test. On the other hand, typical heavy-weight V&V methodologies in SE such as waterfall-based approaches are often not suitable for rule-based systems, because they induce high costs of change and do not facilitate evolutionary modelling of rule-based policies with collaborations of different roles such as domain experts, system developers and knowledge engineers. Moreover, they can not check the dynamic behaviors and the interaction between dynamically updated and interchanged policies/contracts and target execution environments at runtime. Model-checking techniques and methods based e.g., on algebraic-, graph- or Petri-net-based interpretations are computationally very costly, inapplicable for expressive policy/contract rule languages and presuppose a deep understanding of both domains, i.e., of the the testing language / models and of of the rule language and the rule inferences. Although test-driven XP techniques and similar approaches to agile SE have been very successful in recent years and are widely used among mainstream software developers, its values, principles and practices have not been transferred into the rule-based policy and contract representation community yet

In this section, I adopt a successful methodology of XP, namely test cases (TCs), to verify and validate correctness, reliability and adequacy of rule-based policy and contract specifications. Accordingly, the term V&V&I is used as a rough synonym for "evaluation and testing". Both processes guarantee that the rule program provides the intended answers, but also imply other goals such as assurance of security, maintenance and service of the rule-based system. It is well understood in the SE community that test-driven development improves the quality and predictability of software releases and I argue that TCs and integrity constraints also have a huge potential to be a successful tool for declarative V&V of rule-based policy and contract systems. TCs in combination with other SE methodologies such as *test coverage measurement* which is used to quantify the completeness of TCs as a part of the feedback loop in the development process and *rule base refinements* (a.k.a. *refactorings*) [DP05] which optimize the existing rule code (e.g., remove inconsistencies, redundancy or missing knowledge without breaking its functionality) qualify for typically frequently changing requirements and models of rule-based policies and contracts. Since, the test-driven approach focuses on the behavioral aspects and the drawn conclusions instead of the structure of the rule base and the causes

of faults, it is independent of the complexity of the rules and the system under test. Due to this inherent simplicity TCs, which provide an abstracted black-box view on the rules, better support different roles such as domain experts or business practitioneers which are involved during the engineering and enforcement process and give contract engineers an expressive but nevertheless easy to use testing language. In open distributed environment TCs can be used to ensure correct execution of interchanged specifications in target execution environments by validating the interchanged rules with the attached TCs. The presence of test cases safeguards the life cycle of rules, e.g., enabling V&V at design time but also dynamic testing of (transactional) self-updates of the knowledge base, where test cases are used as highly expressive integrity constraints.

In this section I show that test cases can be represented homogeneously as LPs in the ContractLog KR. The major advantage is, that test cases and test suites can be managed, maintained and executed within the same rule based environment, based on a well defined LP semantics. I further describe, how the concept of test coverage from XP can be adapted to logic programming, in order to quantify the quality of test cases for V&V of LPs. Test coverage gives feedback and hints on how to optimize and refine the test cases and the rule code in an iterative process.

4.9.1 Concepts and Related Work

There are many definitions of V&V in the SE literature. In the context of rule-based policies/contracts V&V is defined as follows:

Definition 136 ***(Verification and Validation)***

1. Verification *ensures the logical correctness of a LP. Akin to traditional SE a distinction between structurally flawed or logically flawed rule bases can be made with structural checks for redundancy or relevance and semantic checks for consistency, soundness and completeness.*
2. *As discussed by Gonzales [GB00]* validation *is concerned with the correctness of a rule-based system in a particular environment/situation and domain.*

During runtime certain parts of the rule based decision logic should be static and not subjected to changes or it must be assured that updates do not change this part of the intended behavior of the policy/contract. A common way to represent such constraints are ICs as discussed in section 4.4.2. Roughly, if validation is interpreted as: "*Are we building the right product?*" and verification as: "*Are we building the product right?*" then integrity might be loosely defined as: "*Are we keeping the product right*?", leading to the pattern: **V&V&I**. Hence, ICs are a way to formulate consistency (or inconsistency) criteria of a dynamically updated knowledge base (KB). Another distinction which can be made is between errors and anomalies:

Definition 137 ***(Errors and Anomalies)***

- Errors *represent problems which directly effect the execution of rules. The simplest source of errors are typographical mistakes which can be solved by a verifying parser. More complex problems arise in case of large rule bases incorporating several people during design and maintenance and in case of the dynamic alteration of the rule base by adding, changing or refining the knowledge which might easily lead to incompleteness and contradictions.*
- Anomalies *are considered as symptoms of genuine errors, i.e., they may not necessarily represent problems in themselves.*

Much work has been done to establish and classify the nature of errors and anomalies that may be present in rule bases, see e.g., the taxonomy of anomalies from Preece and Shinghal [PS94]. I briefly review the notions that are commonly used in the literature [AHPV98, Pre01], which range from semantic checks for consistency and completeness to structural checks for redundancy, relevance and reachability:

1. *Consistency*: No conflicting conclusions can be made from a set of valid input data. The common definition of consistency is that two rules or inferences are inconsistent if they succeed at the same knowledge state, but have conflicting results. Several special cases of inconsistent rules are considered in literature such as:

   - *self-contradicting rules* and *self-contradicting rule chains*, e.g., $p \wedge q \rightarrow \neg p$
   - *contradicting rules* and *contradicting rule chains*, e.g., $p \wedge q \rightarrow s$ and $p \wedge q \rightarrow \neg s$

   Note that the first two cases of self-contradiction are not consistent in a semantic sense and can equally be seen as redundant rules, since they can be never concluded.

2. *Correctness/Soundness*: No invalid conclusions can be inferred from valid input data, i.e., a rule base is correct when it holds for any complete model M, that the inferred output from valid inputs by the rule base are true in M. This is closely related to *soundness* which checks that the intended outputs indeed follows from the valid input. Note, that in case of partial models with only partial information this means that all possible partial models need to be verified instead of only the complete models. However, for monotonic inferences these notions coincide and a rule base which is sound is also consistent.

3. *Completeness:* No valid input information fails to produce the intended output conclusions, i.e., completeness relates to gaps (incomplete knowledge) in the knowledge base. The iterative process of building large rule bases where rules are tested, added, changed and refined obviously can

leave gaps such as missing rules in the knowledge base. This usually results in intended derivations which are not possible. Typical sources of incompleteness are missing facts or rules which prevent intended conclusions to be drawn. But there are also other sources. A KB having too many rules and too many input facts negatively influences performance and may lead to incompleteness due to termination problems or memory overflows. Hence, superfluous rules and non-terminating rule chains can be also considered as completeness problems, e.g.,:

- *Unused rules and facts*, which are never used in any rule/query derivation (backward reasoning) or which are unreachable or dead-ends (forward reasoning).
- *Redundant rules* such as identical rules or rule chains, e.g., $p \rightarrow q$ and $p \rightarrow q$.
- *Subsumed rules*, a special case of redundant rules, where two rules have the same rule head but one rule contains more prerequisites (conditions) in the body, e.g., $p \wedge q \rightarrow r$ and $p \rightarrow r$.
- *Self-contradicting rules*, such as $p \wedge q \wedge \neg p \rightarrow r$ or simply $p \rightarrow \neg p$, which can never succeed.
- *Loops* in rules of rule chains, e.g., $p \wedge q \rightarrow q$ or tautologies such as $p \rightarrow p$.

Verification and Validation (V&V) of knowledge based systems (KBS) and in particular rule based systems such as logic programs with Prolog interpreters have received much attention from the mid '80s to the early '90s, see e.g. [AHPV98]. Criteria for verification and validation range from e.g., structural checks for relevance, redundancy and reachability to semantics tests for completeness and consistency. For a survey see [Pre01]. Several verification and validation methods have been proposed, such as:

- Methods based on *operational debugging* [Byr80] by instrumenting the rule base and exploring the execution trace using break points in the rule program (e.g., between the expand and branch steps of the debugging algorithm using trance and spy commands in Prolog). However, these methods presuppose a deep understanding of the inference processes by the user to detect the inconsistencies.
- *Tabular methods*, e.g. [VSB84], which pairwise compare the rules of the rule base to detect relationships among premises and conclusions. Comparing only pairs of rules excludes detection of inconsistencies in rule chains with several rules.
- Methods based on Graphs, e.g. [NK80, RSYS97], using *formal graph theory* to detect inconsistencies by simulating the execution of the system for every possible initial fact base, which might be very costly.

- Methods based on *Petri Nets*, e.g. [HCYY99] which model the rule base as a Petri net and test the complete models starting with all possible initial states, which is very costly.
- Methods based on *declarative debugging* [Sha82] which build an abstract model representing the execution trace and elicit feedback from an oracle (e.g., the user) to navigate through the model till the inconsistency/error is reached.
- Methods based on *algebraic interpretation*, e.g. [LRLLM99] transform a KB into an algebraic structure, e.g., a boolean algebra which is then used to verify the KB. This approach can not be applied to expressive rule bases with variables, object-valued functions or meta predicates and non-monotonic negations.

While these approaches mainly focus on monotonic reasoning, there are also some approaches on verifying non-monotonic rule bases such as [Ant97] which analyzes rule bases expressed in default logic or [WL97] which tests rule bases with production rules. For further details concerning inconsistency checking techniques see e.g. [CBC93]. Much research has been directed at the automated refinement of rule bases, e.g. [BLR97, CS90], and on the automatic generation of test cases, e.g. [CCS90]. For an overview on rule base debugging tools see e.g. [Pla03]. Test coverage for imperative programs has been intensively investigated in the past decades [ZHM97], but there are only a few attempts addressing test coverage measurement for test cases of backward-reasoning rule based programs [Den91, LBSB92, Jac96] or forward-reasoning production rule systems [AJ98].

4.9.2 Syntax of Test Cases for LPs

In SE the general idea of TCs is to predefine the intended output of a program or method and compare the intended results with the derived results. If both match, the TC is said to capture the intended behavior of the program/method. Although there is no 100% guarantee that the TCs defined for V&V of a program exclude every unintended results of the program, they are an easy way to approximate correctness and other SE-related quality goals (in particular when the TCs and the program are refined in an evolutionary, iterative process with a feedback loop). In analogy to TCs in SE I define a TC as follows:

Definition 138 *(**Test Case**) A test case* $TC := \{\overline{As}, \overline{Te}\}$ *for a LP* P *consists of:*

1. *a set of possibly empty input assertions* $\overline{As}$ *being the set of temporarily asserted test input facts (and meta test rules). The assertions are used to temporarily setup the test environment. They can be e.g., used to define test facts, result values of (external) functions, events and actions for testing reactive rules or additional meta test rules.*
2. *a set of one or more tests* $\overline{Te}$*. Each test* Te_i *consists of:*

- *a test query Q, where $Q \in \overline{H}(P)$ and $\overline{H}(P)$ is the set of literals in the head of rules (since only rules need to be tested)*
- *a result l being either "true", "$false$" or "unknown" label.*
- *an intended answer set θ of expected variable bindings for the variables of the test query Q. For ground test queries $\theta := \emptyset$.*

A test case TC is then formally written as $TC = \overline{As} \cup \{Q => l : \theta\}$. $\overline{TC} = \{Q => l : \theta\}$ if a TC has no assertions.

For instance, a TC $TC = \{p(X) => true : \{X/a, X/b, X/b\}, q(Y) => false\}$ defines a TC with two test queries $p(X)$ and $q(Y)$. The query $p(X)$? is intended to succeed and return three answers a,b and c for the free variable X. The query $q(Y)$ should fail. In case we are only interested in the existential success of a test query we shorten the notation of a TC to $TC = \{Q => l\}$.

TCs in the ContractLog KR are homogeneously integrated into the ContractLog KR as meta programming implementation. A TC script consists of (1) a unique ID denoted by the function *testcase(ID)*, (2) optional input assertions such as input facts and test rules which are added temporarily to the KB as partial modules by expressive ID-based update functions, (3) a positive meta test rule defining the test queries and variable bindings *testSuccess(Test Name,Optional Message for Junit)*, (4) a negative test rule *testFailure(Test Name,Message)* and (5) a *runTest* rule.

Example 35

```
% testcase oid
testcase("./examples/tc1.test").
% assertions by ID-based updates adding one rule and two facts
:-solve(add("tc1.test","a(X):-b(X). b(1). b(2).")).
% positive test with success message for JUnit report
testSuccess("test1","succeeded"):-
    testcase(./examples/tc1.test),testQuery(a(1)).
% negative test with failure message for Junit report
testFailure("test1","can not derive a"):-
    not(testSuccess("test1",Message)).
% define the active tests - used by meta program
runTest("./examples/tc1.test"):-testSuccess("test 1",Message).
```

The main axioms of the test logic implemented in the ContractLog KR (testcase.prova) are:

```
test()                    test all test cases in the knowledge base
test(OID/URL)             test the test case with the ID <OID>
                          or load and test test case from URL
```

```
loadTestCase(TestCaseOID)   load a test case temporarily to knowledge base
runTestCase(TestCaseOID)    run the test case and execute all tests
unloadTestCase(TestCaseOID) Unload a test case from the knowledge base
```

Predicates/Functions to define tests within test cases are:

```
testQuery(Literal)                 test the literal (= rule head or fact)
testNotQuery(Literal)              negatively test the literal with default negation
testNegQuery(Literal)              negatively test the literal with explicit negation
testNumberOfResults(Literal, Number) test number of results derived for the literal
testNumberOfResults(Literal, Var, Number)   test number of results for the variable
                                            in the literal
testNumberOfResultsMore(Literal,Number)     test number of results for the
                                            literal > given value
testNumberOfResultsLess(Literal,Number)     test number of results for the
                                            literal < given value
testNumberOfResultsMore(Literal,Var,Number) test number of results for the variable
                                            in the literal > given value
testNumberOfResultsLess(Literal,Var,Number) test number of results for the variable
                                            in the literal < given value
testResult(QueryLiteral,ResultLiteral)      test if the second literal is an answer of
                                            the query literal
testResults(Literal,Var,[<BindingList>])    test if the list of binding results
                                             can be derived
testResultsOrder(Literal,Var,[<BindingList>])   test if the list of ordered
                                                   binding results can be derived
testQueryTime(Literal, MaxTime)      test if the literal can be derived with
                                     < time in milliseconds
testNotQueryTime(Literal, MaxTime)   test if the literal can be derived
                                     negatively by default
                                     in less than the stated time in milliseconds
testNegQueryTime(Literal, MaxTime) test if the literal can be derived strongly negative
                                   in less than the stated time in milliseconds
getQueryTime(Literal, Time)    get the query time for the literal
getNotQueryTime(Literal,Time)  get the default negated query time for the literal
getNegQueryTime(Literal,Time)  get the explicitly negated query time for the literal
```

Example 36

```
% test success of query p(X)?
testQuery(p(X))
% test success of query neg(p(X))? (explicitly negated p(X))
testNegQuery(p(X))
% test if ten bindings for the variable X can be derived
testNumberOfResults(p(X),X,10))
% test if p(a) is an answer to the query p(X)?
testResult(p(X),p(a))
% test if a, b and c are result bindings for the variable X
testResults(p(X),X,[a,b,c])
% test if a, b and c are answer bindings in exactly this order
testResultsOrder(p(X),X,[a,b,c])
testResultsOrder(p(X),X,[a,a,a,b,c,c])
% test if the query p(X)? can be derived in less than 1 second
testQueryTime(p(X),1000)
% get the time to answer the query p(X); output is the variable T
```

```
getQueryTime(p(X),T)
% nested test
testQueryTime(testResults(p(X),X,[a,b]))
```

4.9.3 Semantics of Test Cases

Semantically, TCs are highly expressive integrity contraints.

Definition 139 *(**Test Case Semantics**) Let KB_l be the actual knowledge base (the actual program state) and TC be a test case. Then TC is satisfied wrt KB_l if $KB_l \cup U^{pos}_{TC(\overline{As})} \models TC$ where $U^{pos}_{TC(\overline{As})}$ is the temporarily added set of test assertions.*

A TC is temporarily loaded to the KB for testing purposes, using the ID-based update functions for dynamic LPs (see section 4.5). Provability of a TC in KB_l, denoted $KB_l \vdash TC$ is defined as a standard proof of the tests $\overline{Te}$ wrt to the intended answers θ and the result label l. The proof-theoretic inference rules are axiomatize as a LP meta program. The TC meta program implements various functions, e.g., to define positive and negative test queries (*testQuery, testNotQuery, testNegQuery*), expected answer sets (variable bindings: *testResults*) and quantifications on the expected number of result (*testNumberOfResults*).

From a operational point of view, to become widely accepted and useable to a broad community of policy engineers and practitioners existing expertise and tools in traditional SE and flexible information system (IS) development should be adapted to the declarative test-driven programming approach. Well-known test frameworks like JUnit facilitate a tight integration of tests into code and allow for automated testing and reporting in existing IDEs such as eclipse by automated Ant tasks. The ContractLog KR implements support for JUnit based testing and test coverage reporting where TCs can be managed in test suites (represented as LP scripts) and automatically run by a JUnit Ant task. The ContractLog distribution comes with a set of functional-, regression-, performance- and meta-TCs for the V&V of the inference implementations, semantics and meta programs of the ContractLog KR.

4.9.4 Declarative Test Coverage Measurement

Test coverage is an essential part of the feedback loop in the test-driven engineering process. The coverage feedback highlights aspects of the formalized policy/contract specification which may not be adequately tested and which require additional testing. This loop will continue until coverage of the intended models of the formalized policy specification meets an adequate approximation level by the TC resp. test suites (TS) which bundle several TCs. Moreover, test coverage measurements helps to avoid atrophy of TSs when the rule-based specifications

are evolutionary extended. Measuring coverage helps to keep the tests up to a required level if new rules are added or existing rules are removed/changed.

However, conventional testing methods for imperative programming languages rely on the control flow graph as an abstract model of the program or the explicitly defined data flow and use coverage measures such as branch or path coverage. In contrast, the proof-theoretic semantics of LPs is based on resolution with unification and backtracking, where no explicit control flow exists and goals are used in a refutation attempt to specialize the rules in the declarative LP by unifying them with the rule heads. Based upon this central concept of unification a test covers a logic program P, if the test queries (goals) lead to a least general specialization of each rule in P, such that the full scope of terms (arguments) of each literal in each rule is investigated by the set of test queries. That is, the instantiation of the rules in P with the test goals should be as general as possible. Finding general information from specific goals is a task approached by inductive logic programming (ILP) techniques. ILP allows inductively deriving general information from specific knowledge and computing the least general generalization (lgg), i.e., the most specific clause (e.g., wrt theta subsumption) covering two input clauses. A lgg is the generalization that keeps an generalized term t (or clause) as special as possible so that every other generalization would increase the number of possible instances of t in comparison to the possible instances of the lgg. Efficient algorithms based on syntactical anti-unification with θ-subsumption ordering for the computation of the (relative) lgg(s) exist and several implementations have been proposed in ILP systems such as GOLEM, or FOIL. θ-subsumption introduces a syntactic notion of generality: A rule (clause) r (resp. a term t) θ-subsumes another rule r', if there exists a substitution θ, such that $r \subseteq r'$, i.e., a rule r is *as least as general as* the rule r' ($r \leq r'$), if r θ-subsumes r' resp. *is more general than* r' ($r < r'$) if $r \leq r'$ and $r' \not\leq r$. (see e.g. [Plo70]). In order to determine the level of coverage the specializations of the rules in the LP under test are computed by specializing the rules with the test queries by standard unification. Then by generalizing these specializations under θ-subsumption ordering, i.e., computing the lggs of all successful specializations, a reconstruction of the original LP is attempted. The number of successful "recoverings" then give the level of test coverage, i.e., the level determines those statements (rules) in a LP that have been executed/investigated through a test run and those which have not. In particular, if the complete LP can be reconstructed by generalization of the specialization then the test fully covers the LP. Formally we express this as follows:

Definition 140 ***(Test Coverage)*** *Let T be a test with a set of test goals $\{G_1, .., G_n\}$ for a program P, then T is a cover for a rule $r_i \in P$, if the $lgg(r'_i) \simeq r_i$ under $\theta - subsumption$, where $\simeq$ is a variant equivalence relation denoting variants of clauses/terms and the r'_i are the specializations of r_i by the goals G_j. It is a cover for a program P, if T is a cover for each rule $r_i \in P$. With this definition it can be determined whether a test covers a LP or not. The coverage measure for a LP P is then given by the number of covered*

rules r_i divided by the number k of all rules in P:

$cover_P(T) : -\frac{\sum_{i=1}^{k} cover_{r_i}(T)}{k}$

For instance, consider the following simplified business policy P:

Example 37

```
discount(Customer, 10%) :- gold(Customer).
gold(Customer) :- spending(Customer, Value) , Value > 3000.
spending('Moor',5000).
spending('Do',4000). %facts
```

Let $T = \{discount('Moor', 10\%)? => true, discount('Do', 10\%)? => true$ be a test with two test queries. The set of directly derived specializations by applying this tests on P are:

```
discount('Moor',10%) :- gold('Moor').
discount('Do',10%) :- gold('Do').
```

The computed lggs of this specializations are:

```
discount(Customer,10%) :- gold(Customer).
```

Accordingly, the coverage of P is 50%. We extend T with the additional test goals: $\{gold('Moor')? => true, gold('Do')? => true)?\}$. This leads to two new specializations:

```
gold('Moor') :- spending('Moor',Value) , Value > 3000.
gold('Do') :- spending('Do',Value) , Value > 3000.
```

The additional lggs are then:

```
gold(Customer) :- spending(Customer, Value) , Value > 3000.
```

T now covers P, i.e., coverage = 100%.

The coverage measure determines how much of the information represented by the rules is already investigated by the actual tests. The actual lggs give feedback how to extend the set of test goals in order to increase the coverage level. Moreover, repeatedly measuring the test coverage each time when the rule base becomes updated (e.g., when new rules are added) keeps the test suites (set of TCs) up to acceptable testing standards and one can be confident that there will be only minimal problems during runtime of the LP because the rules do

not only pass their tests but they are also well tested. In contrast to other computations of the lggs such as implication (i.e., a stronger ordering relationship), which becomes undecidable if functions are used, θ-subsumption has nice computational properties and it works for simple terms as well as for complex terms with or without negation, e.g., $p() : -q(f(a))$ is a specialization of $p : -q(X)$. Although it must be noted that the resulting clause under generalization with θ-subsumption ordering may turn out to be redundant, i.e., it is possible find an equivalent one which is described more shortly, this redundancy can be reduced and since we are only generalizing the specializations on the top level this reduction is computationally adequate. Thus, θ-subsumption and least general generalization qualify to be the right framework of generality in the application of my test coverage notion.

The defined coverage measure is based on the central concept of unification and uses ILP techniques for the generalization of the derived specializations of the rule base. It is worth noting, that the measure might be applied also in the context of forward-directed reaction rules, since in the ContractLog KR reaction rules are homogeneously represented together with derivation rules. To compute the least general generalizations (lgg) I have adapted Plotkin's least general generalization and extended it in ContractLog to a full meta inference engine which allows computing the substitution sets of terms and clauses (rules/facts/goals), apply the substitutions to compute the specialisations (the rule instances), generalize clauses/terms and compute the lgg, the coverage level and give coverage feedback (e.g., the covered clauses, not covered clauses, the coverage level etc.). The following axioms are provided by the test logic meta program:

```
Specialization

% compute and return the substitution
substitution(Term1,Term2,Subst)
substiute(Clause,ClauseInstance,Subst)
substiute(Term,TermInstance,Subst)

% specialize, i.e., unify and return the specialization
specializations(Goal,Clause,Instances)
specialize(Goal,InputLP,OutputLP)

Generalization

% compute the lgg
lgg (Clause1,Clause2,LGG)
lgg (Term1,Term2,LGG)

% compute all lggs
lggs (Clause,LP,LGGs)

% generalize an input LP
```

```
generalize(InputLP,OutputLP)

Cover / Coverage

% return the covered clause from both LPs
cover(LP1,LP2,CoveredClause)
% compute the test coverage
coverage(Goal,LP,CoveredClauses,NotCoveredClauses,CoverageLevel)
```

The specialization and generalization axioms might be used as a complete meta inference system which implements a standard top-down derivation on a meta level.

Here are some examples to illustrate the use of the inductive logic / meta inference functions implemented in the ContractLog KR:

Example 38

```
% compute the substitution set for the two complex terms
:-solve(substitution(f(g(A),B),f(g(h(a)),i(b)),Subst)).

% substitute a complex term with the substitution set
% {(A / h(a)),(B / h(b))}
:-solve(substitute(f(g(A),A),Instance,
            [[A,["h","a"]],[B,["h","b"]]])).

% compute the lgg = f(X, g(Y,Z), c).
:-solve(lgg(f(a, g(b, h(X)), c), f(d, g(j(X), a), c),LGG)).

% Generalize a LP
% and return the generalized LP (set of general rules)
:-solve(generalize([
        [p(a),q(a)],
        [p(a),q(a),r(a)],
        [p(b),q(b)],
        [p(c),q(c)],
        [r(a)],[q(a)],[q(b)],[q(c)]],
    Generalization)).
```

A special built-in predicate $metaLP(LP)$ automatically translates the internal rules/facts of the knowledge base into the list representation format and binds it to the variable LP.

4.9.5 Summary

The majority of V&V approaches for rule-based systems rely on debugging the derivation trees and giving explanations (e.g., by spy and trace commands) or

transforming the program into other more abstract representation structures such as graphs, petri nets or algebraic structures which are then analyzed for inconsistencies. Typically, the definition of an inconsistency, error or anomaly is then given in the language used for analyzing the LP, i.e., the V&V information is not expressed in the same representation language as the rules. This is in strong contrast to the way people would like to engineer, manage and maintain rule-based contracts and policies. Different skills for writing the formalized specifications and for analyzing them are needed as well as different systems for reasoning with rules and for V&V. Moreover, the used V&V methodologies (e.g., model checking or graph theory) are typically much more complicated than the rule-based programs. In fact, it turns out that even writing rule-based systems that are useful in practice is already of significant complexity, e.g., due to non-monotonic features or different negations, and that simple methods are needed to safeguard the engineering and maintenance process w.r.t. V&V&I. Therefore, what policy engineers and practitioners would like to have is an "easy-to-use" approach that allows representing rules and tests in the same homogeneous representation language, so that they can be engineered, executed, maintained and interchanged together using the same underlying syntax, semantics and execution/inference environment.

In this section, I have transfered a development methodology of XP, namely test-driven development with test cases, as a tool for declarative verification and validation of LP-based rule specifications, which is suitable for typically frequently changing requirements and models of rule-based SLM projects. Test cases for rule based policies are particular well-suited when policies/contracts grow larger and more complex and are maintained, possibly distributed and interchanged, by different people. In this section I have adopted the test-driven techniques developed in the Software Engineering community to the declarative rule based programming approach for engineering high level policies such as SLAs. I have elaborated on an approach using logic programming as a common basis and have extended this test-driven approach with the notion of declarative test coverage.

Clearly, test cases and test-driven development is not a replacement for good programming practices and rule code review. However, the presence of test cases helps to safeguard the life cycle of policy/contract rules, e.g., enabling V&V at design/development time but also dynamic testing at runtime. In general, the test-driven approach follows the well-known 80-20 rule, i.e., increasing the approximation level of the intended semantics of a rule set (a.k.a. test coverage) by finding new adequate test cases becomes more and more difficult with new tests incrementally delivering less and less. Hence, under a cost-benefit perspective one has to make a break-even point and apply a not too defensive development strategy to reach practical levels of rule engineering and testing in larger rule based policy or contract projects.

4.10 Summary and Discussion

In this chapter I have described syntax and semantics of the ContractLog KR formalisms and illustrated their features with several examples. The formalisms have been carefully selected and implemented on the basis of KR adequacy criteria to fulfil the practical real-world requirements of the domain, in particular wrt the expressiveness needed to fulfill the requirements for a SLA representation language (as discussed in section 3.6), and wrt adequate computational complexity. A detailed discussion of these adequacy criteria will follow in chapter 7.

I have implemented the ContractLog formalisms on the basis of declarative logic programs and meta programming techniques as stand-alone LP scripts which can be individually imported according to the needed expressiveness for a particular SLA domain. The formal semantics basically qualifies the ContractLog KR to be applicable to different LP execution environments and LP semantics. However, as discussed in section 3.6 there is also a need for non-standard and new language constructs in order to e.g., integrate external system components, merit object-oriented programing with declarative programing and manage distributed "webized" rule bases in an open web-based environment. I have given these practical language constructs a formal semantics which however is non-standard in terms of typical LP and FOL semantics. Hence, the procedural semantics of standard LP rule engines need to be extended to adopt these practical constructs. I have payed special attention in ContractLog to this trade-off and described the non-classical (non-monotonic) formal and procedural semantics of these formalisms in order to allow mappings of these important and intuitive extensions to standard rule engines (e.g. Prolog interpreters), e.g. via wrapper interfaces as in the case of the ECA interpreter for global ECA rules (see section 4.7).

In summary the ContractLog formalisms allow to adequately express and implement business and contract rules found in SLAs and policies. Without strictly following classical FOL and their theoretical issues such as decidability, which prevent the implementation of a practically usable rule language, the ContractLog formalisms provide a non-classical formal semantics which produces highly verifiable, traceable and reliable results. The KR merits the benefits of declarative rule-based logic programming and state-of art object-oriented, relational and Semantic Web technologies. As a result, the KR provides rich interfaces to external data sources and systems, and provides the necessary flexibility and openness to integrate external (business) domain vocabularies into the rule specifications. As such, the integration and interoperation of ContractLog in and with existing technologies, systems and tools such as Semantic Web languages, workflow systems, web services (WSDL, BPEL), business process and business acitivity management systems becomes possible by their application programming interfaces and their close semantic relations to certain formalisms of the ContractLog KR.

5 Rule Based Service Level Agreement Markup Language (RBSLA)

The ContractLog KR provides a declarative rule language based on logic programming (LP) providing a clean separation of concerns by explicitly expressing contractual logic in a formal interpretable and executable fashion with a high degree of flexibility. But, real usage of a formal rule-based representation language which is usable by others than its inventors immediately makes rigorous demands on the syntax: declarative machine-readable syntax, comprehension, usability of the language by human users and machines, compact representation, exchangeability with other formats, means for serialization and persistence, tool support in writing and parsing rules (verification/validation) etc. In this section I introduce a declarative Rule-based Service Level Agreement language (RBSLA) which addresses these requirements. Therefore, it adapts and extends the emerging Semantic Web rule standard RuleML to the needs of the SLA domain in order to facilitate interoperability with other rule languages and XML-based tool support.

5.1 Rule Markup and Rule Interchange Languages

Recently, there have been many efforts aiming on rule interchange and building a general, practical, and deployable rule markup standard for the (Semantic) Web. This includes several important *general standardization or standards-proposing efforts* including RuleML [BT00], W3C RIF [RIF05], SWRL (DAML Rules) [HPSB+04], Metalog [MS98], and others. Several markup approaches have been proposed with a more or less specialized purposes, e.g., in the domains of *production rules* (SRML [TK01], PRR [TWS+04]), *reaction rules / ECA rules* (Reaction RuleML [PKB+06] (former ECA-RuleML), Active XML [ABM+02], Active Rules for XML [BCP01], the ECA language for XML proposed by Bailey et.al [BPW02], RDFTL for RDF [PPW04], XChange [BP05]), *business rules* (BRML [BRM02], SBVR [SBV06]), *Semantic Web Service and policies* (WS-Policy [HK03], SWSL [SWS04], WSML [WSM05a]) and other areas as well. There have been also several *SLA related markup language* proposals such as the widely known Web Service Level Agreements (WSLA) [DDK+04] and comparable approaches such as the SLA language (SLAng [LSE03]), the Web Services Offering Language (WSOL [WSO05]) or the WS-Agreement proposal [ACD+05].

As discusses in section 2.5 these SLA markup languages are more or less pure syntactical specifications languages with very limited expressiveness to represent sophisticated and dynamic rule sets and complex conditionals. The semantics of these languages is not based on a mathematical (model) theory and derivations are not based on a resolution based proof theory as in my RBSLA/ContractLog approach but uses pure procedural logic for interpretation of truth-functional constructions expressing only simple material implication rules. In particular, they do not provide means for declarative programming new functionalities which are automatically interpreted by standard rule engines and lack required expressive features such as non-monotonic default negation, default reasoning with exceptions and priorities to cope with conflicts in open domain setting or event, action, state based processing and reasoning capabilities. Although, syntactically rich they are only suitable for more or less standardized static SLA specifications but not for highly dynamic, individualized and frequently changing service contracts in modern service oriented computing settings and ITSM models which need to be tightly integrated and interact with many other functionalities, processes and decision logics captured, e.g., in terms of business rules, integrity rules or policies, data base models or procedural implementations. Due to the missing declarative programming features and the lack of a precise formal semantics extensions of these languages become difficult requiring time-consuming and costly reimplementations of the non-standard procedural interpreters. Moreover, traceability, verifiability and correctness of produced results can not be ensured.

On the other hand, general rule markup languages (GRML) [Wag02] provide this required expressiveness, act as a "lingua franca" to exchange rules and are based on a semantics of formal logic, usually a variation of first order predicate logic, where the responsibility to interpret the rules and to decide on *how* to do it is delegated to a standard interpreter (a rule engine). Accordingly, one of my design goals for the RBSLA language is to stay as close as possible to the current quasi-standard, the Rule Markup language (RuleML [BT00]), and reuse the existing language constructs as much as possible. That is, RBSLA is implemented on top of RuleML extending it with additional modelling power for serializing/programming arbitrary SLA rules.

5.2 RuleML: The Rule Markup Language Initiative and Language

The Rule Markup Language (RuleML) [WTB03] is a markup language developed to express both forward (bottom-up) and backward (top-down) rules in XML for deduction, rewriting, and further inferential-transformational tasks. It is defined by the Rule Markup Initiative [BT00], an open network of individuals and groups from both industry and academia that was formed to develop a canonical Web language for rules using XML markup and transformations from and to other rule standards/systems. It develops a modular, hierarchical specification for different types of rules comprising reaction rules

(Reaction RuleML [PKB+06]), derivation rules, facts, queries and integrity constraints (consistency-maintenance rules) as well as transformations via XSLT and ANTLR from and to other rule standards/systems. [WTB03] It is expected that RuleML will be the declarative method to describe rules on the Web and distributed systems. Certain parts of RuleML have most recently contributed to the W3C RIF Core [RIF05]. RuleML allows the deployment, execution, and exchange of rules between different major commercial and non-commercial rules systems like e.g. Jess, Prova, JDrew or Mandarax via XSLT transformations. RuleML is not intended to be executed directly, but transformed into the target language of an underlying rule-based systems (e.g. Prova) and then executed there. It addresses machine-readability, interoperability with other rule languages and (XML) tool support. Since the object oriented RuleML (OO RuleML) specification 0.85 [Bol03] it adds further concepts from the object-oriented knowledge representation domain namely user-level roles, URI grounding and RDF term typing and offers first ideas to prioritize rules with quantitative or qualitative priorities.

For the rest of this section I will briefly summarize the key components of the RuleML language and then introduce the RBSLA language and the Reaction RuleML sublanguage [PKB+06] (former ECA-RuleML [Pas05c]) which extends RuleML with additional language constructs for representing reaction rules and SLA/policy rules. An overview of the current RuleML structure can be found in appendix D. Here, I focus on the RuleML horn logic layer extended with negations and equality which is chosen as the basis for the Reaction RuleML and the RBSLA language. The building blocks are: [WTB03]

- Predicates (atoms) are n-ary relations defined as an <Atom> element in RuleML. The main terms within an atom are variables <Var> to be instantiated by ground values when the rules are applied, individual constants <Ind>, data values <Data> and complex terms <Cterm>.
- Derivation Rules (<Implies>) consist of a body part (<body>) with one or more conditions (atoms) connected via <And> or <Or> and possibly negated by <Neg> which represents classical negation or <Naf> which represents negation as failure and a conclusion (<head>) which is derived from existing other rules or facts applied in a forward or backward manner.
- Facts are deemed to be always true and are stated as atoms: <Atom>
- Queries <Queries> can either be proven backward as top-down goals or forward via bottom-up processing. Several goals might be connected within a query and negated.

Besides facts, derivation rules and queries RuleML defines further rule types such as integrity constraints and transformation rules [WTB03].

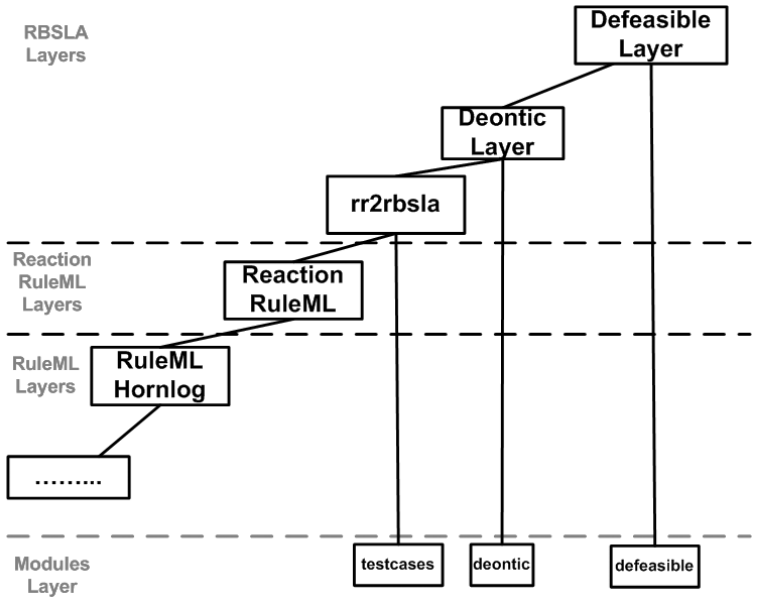

Figure 5.1: RBSLA Layers and Modularization

5.3 RBSLA: Rule Based Service Level Agreement Markup Language

The Rule Based Service Level Agreement language (RBSLA) [Pas04a] is a mark-up language to serialize rule-based policy- and contract specifications such as SLAs in XML. It is implemented as an extension to the emerging XML-based Rule Markup language (RuleML) [BT00] and Reaction RuleML (Reaction RuleML) [PKB+06] in order to address interoperability with other rule languages and tool support - see figure 5.1.

It adds additional modelling power and expressiveness to RuleML to declaratively implement higher-level policies and SLAs. In a nutshell, it adds the following features to RuleML:

- expressive procedural attachments on external procedural code integrated into complex functions
- integration of external data sources / facts and imports of external unitized modules
- Reaction RuleML (former ECA-RuleML): reaction rules with events and action language constructs
- ID-based update primitives and module support with "need-to-know principle"
- complex event processing and state changes (fluents) a la event calculus

- complex interval based event / action algebra constructs
- deontic norms for normative reasoning
- defeasible rules and rule priorities
- test cases and expressive integrity constraints for verification, validation and integrity testing (V&V&I)
- typed logic with Semantic Web ontology or object-oriented types and input / output modes

In the following subsections, I will first outline the design goals of RBSLA and then describe its sub-languages, namely *Reaction RuleML* [PKB+06] and the *deontic and defeasible layer* of RBSLA.

5.3.1 Design Goals

RBSLA is designed to fulfil typical criteria for good language design [Cod71] such as *minimality*, *symmetry* and *orthogonality*. With minimality I mean that RBSLA provides only a small set of needed language constructs in addition to the existing constructs in RuleML, i.e., the same meaning cannot be expressed by different language constructs. Symmetry is fulfilled in so far as the same language constructs always expresses the same semantics regardless of the context they are used in. Orthogonality permits every meaningful combination of a language constructs to be applicable. Moreover, RBSLA satisfies typical KR adequacy criteria such as epistemological adequacy: "*A representation is called epistemologically adequate for a person or a machine if it can be used practically to express the facts that one actually has about the aspects of the world.*" [MH69]. This compact design is congruent with the primary intention of RBSLA (and ContractLog) which is to enable declarative programming of SLA related functionalities for the specification of flexible SLA rules, i.e. providing maximum flexibility and extensibility for SLA representation which is different to the syntactical specification approaches of WS-Agreement or WSLA.

Basic requirements which are addressed by RBSLA are, e.g., declarative machine-readable and machine-interpretable syntax, comprehension, usability of the language by human users and automated agents, compact representation, interchangeability with other formats, means for serialization and persistence, tool support in writing and parsing rules as well as verification etc.

RBSLA follows the modularization design principle of RuleML which itself adopts from XHTML [XHT06] and defines new constructs within separated modules which are added to the RuleML family as additional layers on top of the hornlog layer of RuleML. The layers are not organized around complexity, but add different modelling expressiveness to the RuleML core for the representation of contractual logic and behavioral event/(re)action logic. The layered and uniform design makes it easier to learn the language and to understand the relationship between the different features and it provides certain guidance to

users who might be interested only in a particular subset of the features and do not need support for the full expressiveness of RBSLA. The modularization allows for easy extension of RBSLAs representation capabilities, using the extensibility of XML Schema (e.g. redefines of XML Schema group definitions), without breaking the core RBSLA language standard. This development path provides a stable, useful, and implementable language design for SLA developers to manage the rapid pace of change on the Semantic Web and modern IT service management. Apart from that modules facilitate the practical and extensible development of RBSLA via bundling them to layers which can be developed, compiled, tested and managed separately. The modularization also enforce the principle of information hiding and can provide a basis for data abstraction.

The possibility to integrate external (contract) vocabularies such as Semantic Web ontologies into the logical RBSLA rules in order to give them domain-specific meaning provides easy extensibility, interchange and rich expressiveness. Arbitrary existing Semantic Web ontologies such as WSMO [WSM05b], WS-Policy OWL ontology [PKH05, VAG05], OWL-S [OS03] or KAoS [JCJ+03] or other ontologies such as OWL time [PH04] can be used to type rule terms in RBSLA; hence, providing the same syntactical expressiveness as languages such as WSLA or WS Agreement, but in contrast to these approaches with formal logic-based semantics supporting verification, this uses interpretation and inference reasoning using standard semantic web technologies and rule interpreters.

RBSLA is not intended to be executed directly, but its various sublanguages can be transformed into target execution languages of underlying rule-based systems. XSLT stylesheets are provided that transform RBSLA (sublanguages) into executable rule scripts for execution in the targeted computation environments. The reference execution model is the ContractLog KR and the Prova rule engine as execution environment, but translations into other execution languages are also possible.

5.3.2 Reaction RuleML: A Rule Markup Language for Reaction Rules

Reaction RuleML [PKB+06, Pas06d, Pas06c, PKH06] is a general, practical, compact and user-friendly XML-serialized language for the family of reaction rules. It incorporates different kinds of production, action, reaction, and KR temporal/event/action logic rules into the native RuleML syntax using a system of step-wise extensions. In particular, the approach covers different kinds of reaction rules from various domains such as active-database ECA rules and triggers, forward-directed production rules, backward-reasoning temporal-KR event/action/process logics, event notification and messaging and active update, transition and transaction logics. The current version Reaction RuleML 0.1 [PKB+06] directly evolves from ECA-RuleML [Pas05c, Pas06b, Pas05b, Pas06a] a sublanguage of the former RBSLA versions [Pas04a, Pas05f]. Figure 5.2 shows the scope of Reaction RuleML and figure 5.3 shows the structure of Reaction RuleML 0.1.

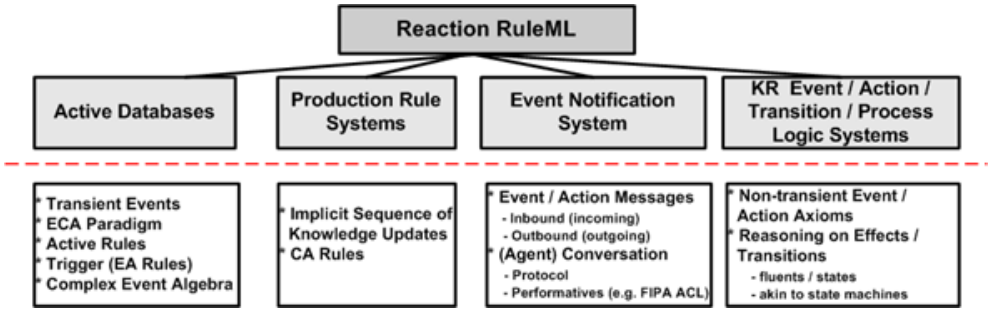

Figure 5.2: Scope of Reaction RuleML

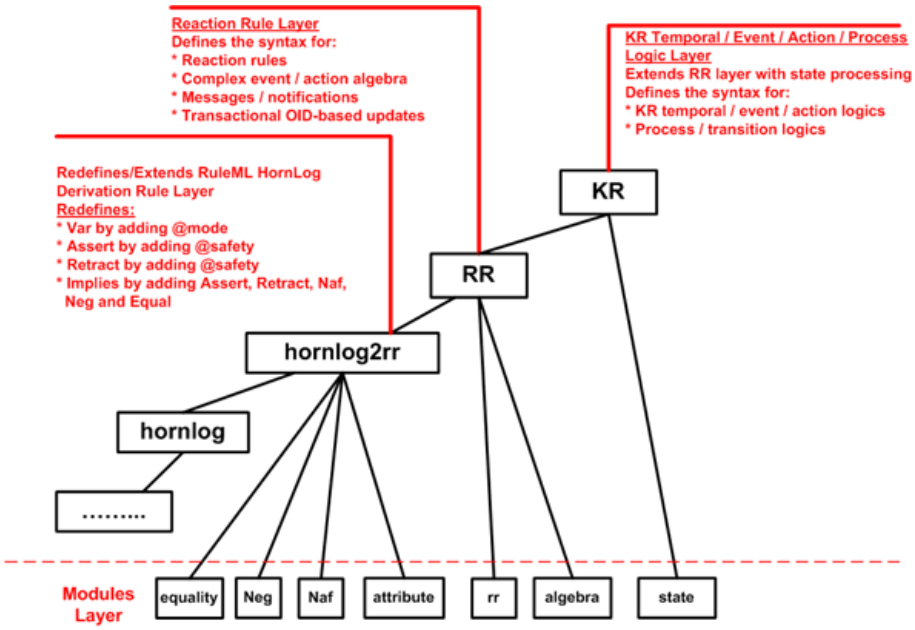

Figure 5.3: Structure of Reaction RuleML 0.1

Hornlog2rr Layer The hornlog2eca layer acts as an intermediate between RuleML and the Reaction RuleML language. It redefines several RuleML constructs and makes several extensions to the RuleML horn logic layer such as transactional (@$safety$) update primitives ($< Assert >/< Retract >$) on the level of atoms in rules ($< Implies >$) to integrate external modules and add/remove knowledge or type (@$type$) and mode (@$mode$) declarations to define webized (Semantic Web ontology) or procedural (Java) types and input/output definitions.

Modes are states of instantiation of the predicate described by mode declarations, i.e. declarations of the intended input-output constellations of the predicate terms with the following semantics:

- "+" The term is intended to be input
- "−" The term is intended to be output
- "?" The term is undefined/arbitrary (input or output)

Modes are frequently used in inductive logic programming (ILP) to reduce the space of clauses actually searched, i.e. to narrow the hypothesis space of program clauses structured e.g. by theta-subsumption generality ordering (refinement graph). They reflect the data flow of a rule set. In Reaction RuleML I define modes with an optional attribute @$mode$ which is added to terms, e.g. $< Varmode = "-" > X < /Var >$, i.e. the variable X is an output variable. By default the mode is undefined "?".

Types in Reaction RuleML can be assigned to terms using a @$type$ attribute. Reaction RuleML supports primitive built-in data types such as String, Integer and XML Schema built-in data types such as $xs : dateTime$ as well as external type definitions such as Java class hierarchies (fully qualified Java class names), e.g. $type = "java.lang.Integer"$ or Semantic Web taxonomies based on RDFS or OWL, e.g. $type = "rbsla : Provider"$, where $rbsla$ is the namespace prefix and $Provider$ is the concept class.

Another important extension to RuleML are *procedural attachments*. Procedural attachments are in particular relevant for actively accessing / monitor external systems and detecting events in a pull mode. They allow receiving/integrating external information, affecting the outside world or delegating computation-intensive tasks to optimized procedural code (e.g. Java methods). Hence, they are a crucial extension of the pure logical inferences used in logic programming.

Example 39

```
...
 <Atom>
    <oid> <!-- class/object -->
     <Ind uri="java://rbsla.utils.TroubleTicketSystem"/>
```

```
     </oid>
    <Rel in="effect"> <!-- method -->
        processTicket
    </Rel>
    <Var type="event:EventType1" mode="+">
        TroubleTicket
    </Var> <!-- parameter -->
</Atom>
...
```

The example show a boolean-valued procedural attachment calling the method *processTicket* of the Java class *rbsla.utils.TroubleTicketSystem* with the valued of the bound ($mode = " + "$) variable *TroubleTicket* of type "$event$: $EventType1$" as argument. Object-valued procedural attachments, i.e. calls to methods/constructors which return an object or a set of objects can be bound to variables using equalities

Example 40

```
<Equals>
    <Var type="java.lang.Integer">X</Var>
    <Expr>
        <oid><Ind uri="java://java.lang.Integer"/></oid>
        <Fun in="effect">parseInt</Fun>
        <Ind>1</Ind>
    </Expr>
</Equals>
```

The example calls the method *parseInt* of the class *java.lang.Integer* with the argument 1 and binds the returned Java Integer object from the external function call with side effect ($in = effect$) to the variable, i.e. $X = java.lang.Integer.parseInt("1")$. Note, that the bound (external) objects can be used anywhere in the rules to call their methods, e.g.:

Example 41

```
 ...
 <Expr>
    <oid><Var type="java.lang.Integer">X</Var</oid>
    <Fun in="effect">toString</Rel>
</Expr> ...
```

In this example the method "*toString*" of a object of type *java.lang.Integer* which was previously bound to the variable X is called.

ID based update primitives, allow dynamically adding or removing internal and external knowledge and evolve the knowledge system at runtime. Reaction RuleML reuses the KQML like primitives of RuleML, which are defined on the top level, as update actions within rules. That is the update primitives are defined on the level of atoms and can be used within rules. An attribute @*safty* states whether the update should be performed in transactional mode, i.e. it might be rolled back in case of failures. External knowledge files which should be asserted can be referenced within the oid tag.

Example 42

```
<Assert>
  <And>
    <oid>
      <Ind uri="http://ibis.in.tum.de/research/
      projects/rbsla/math.rbsla"/>
    </oid>
  </And>
</Assert>

<Assert safety="transactional">
   <And>
      <oid><Ind>update1</Ind></oid>
      <Atom><Rel>f</Rel></Atom>
       <Implies>
         <Atom><Rel>f</Rel></Atom>
         <Atom><Rel>p</Rel></Atom>
      </Implies>
   </And>
</Assert>
```

The example shows two updates of the KB. The first one imports an external rule set from an RBSLA script and the second one asserts a set of clauses as a module with the user-defined module ID "*update*1" to the KB.

The **RR Layer** (Reaction Rules layer) defines the syntax for reaction rules and their component parts. I have defined one *general* < *Reaction* > *tag* to describe different *kinds* (@*kind*) of production, action, reaction, and KR temporal/event/action logic rules. Different *execution styles* (@*exec*) for processing the reaction rules such as *active*, where the reaction rules actively pull or detect the events possibly clocked by a monitoring/validity time function, *passive*, where the reaction rules passively wait (listen) on matching event instances, e.g. incoming event messages, which match with the define event definition patterns, and *reasoning*, where the focus is on the formalization of events and actions and reasoning on their effects on changeable knowledge states (fluents). Reaction rules can be specified on a *global level* in a tight combination with other

rule types such as derivation rules or integrity constraints or *locally*, i.e. nested within other derivation or reaction rules. There are different *evaluation styles* (@*eval*) for reaction rules such as *strong* and *weak* interpretation which are used to manage the "justification lifecycle" of local reaction rules in the derivation process of the outer rules. Reaction RuleML supports procedural calls on external procedures with side effects and enables expressive transactional OID-based updates on the extensional and intensional knowledge base, i.e. on facts and rules. Sophisticated post-conditional verification, validation and integrity tests (V&V&I) using integrity constraints or test cases can be applied as post-conditional tests, which possibly might lead to roll-backs of the update actions. Complex events and actions can be expressed in terms of complex event / action algebra operators and different selection and consumption policies can be configured.

The basic constructs of Reaction RuleML 0.1 are:

- $<$ *Reaction* $>$ - General reaction rule construct
- @*exec* = "*active*|*passive*|*reasoning*"; *default* = "*passive*" - Attribute denoting "active", "passive" or "reasoning" execution style
- @*kind* - Attribute denoting the kind of the reaction rule, i.e. its combination of constituent parts, e.g. "*eca*", "*ca*", "*ecap*"
- @*eval* - Attribute denoting the interpretation of a rule: "*strong*|*weak*"
- $<$ *event* $>$,$<$ *body* $>$,$<$ *action* $>$,$<$ *postcond* $>$, $<$ *alternative* $>$ - role tags; may be omitted when they can be uniquely reconstructed from positions
- $<$ *Message* $>$ - defines an inbound or outbound message
- @*mode* = *inbound*|*outbound* - Attribute defining the type of a message
- @*directive* = [directive, e.g. FIPA ACL]
- $<$ *Assert* $>$ | $<$ *Retract* $>$ - Performatives for internal knowledge updates

The basic content models are given in EBNF notation, i.e. alternatives are separated by vertical bars (|); zero to one occurrences are written in square brackets ([]) and zero to many occurrences in braces ({}):

```
Reaction ::= [oid,] [event,] [body,] [action] [,postcond]
             [,alternative]
event ::= Naf | Neg | Atom | Message | Reaction
body ::= Naf | Neg | Atom | And | Or
action ::= Atom | Assert |Retract | Message
postcond ::= Naf | Neg | Atom | And | Or
alternative ::= Atom | Assert | Retract
```

The **KR Layer** (Knowledge Representation layer) defines the syntax for KR even / action logics such as Event Calculus or Fluent Calculus and extends the RR layer with state processing constructs:

```
state ::= Ind | Var | Expr
Initiates ::= [oid,] state | Ind | Var | Expr
Terminates ::= [oid,] state | Ind | Var | Expr
```

Example 43

```
<Reaction kind="ea" exec="reasoning">
   <event>
      <Atom>
        <Rel>happens</Rel>
        <Ind>startMaintenance</Ind>
        <Var>T</Var>
        </Atom>
    </event>
    <action>
        <Initiates>
           <state>
            <Ind>maintenance</Ind>
           </state>
        </Initiates>
    </action>
</Reaction>
```

The example shows a trigger rule (EA rule) with *reasoning* evaluation style: an initiating event "$startMaintenance$" initiates a state "$maintenance$".

Events (and actions) might be complex:

Example 44

```
<Sequence>
   <Concurrent>
        <Ind>a</Ind>
        <Ind>b</Ind>
   </Concurrent>
   <Ind>c</Ind>
</Sequence>
```

The example defines a complex event $sequence(concurrent(a, b), c)$.

5.3.3 RBSLA Deontic Layer

The further layers of RBSLA are decoupled from Reaction RuleML by the *rr2rbsla layer*. In this layer to support distributed management and rule interchange I have added test cases into the RuleML syntax. The markup serialization syntax for test suites and test cases includes the following constructs

```
assertions ::= And
test ::= Test | Query
message ::= Ind | Var
TestSuite ::= [oid,] content | And
TestCase ::= [oid,] {test | Test,}, [assertions | And]
Test ::= [oid,] [message | Ind | Var,] test | Query,
         [answer | Substitutions]
Substitutions ::= {Var, Ind | Expr}
```

Example 45

```
<TestCase @semantics="semantics:STABLE"
class="class:Propositional">
    <Test @semantics="semantics:WFS" @label="true">
        <Ind>Test 1</Ind><Ind>Test 1 failed</Ind>
        <Query>
            <And>
                <Atom><Rel>p</Rel></Atom>
                <Naf><Atom><Rel>q</Rel></Atom></Naf>
            ...
</TestCase>
```

The example shows a test case with the test: $test1 : \{p => true, not\ q => true\}$.

The *deontic layer* defines the syntax for describing deontic norms. To enable an extensible design of deontic norms, e.g. to add other modalities, the norms are no longer introduced as extra constructs (as described in RBSLA 0.1 [Pas05f]) but the attribute *@in* is extended with the value *modal*.

```
in ::= yes | no | effect | modal
```

Example 46

```
<Atom>
  <Rel in="modal">believe</Rel>
  <Expr> ... A ... </Expr>
</Atom>
```

The example shows the definition of a believe modality $\Diamond A$, i.e. A is believed to be true (is possible true).

An optional role tag *norm* is added as alternative to the role tag *state* in order to enable the definition of time-based (deontic) norms as described in section 4.8.

Example 47

```
<Initiates>
    <norm>
        <Expr in="modal">
            <Fun>oblige</Fun>
            <Var>Subject</Var>
            <Var>Object</Var>
            <Expr><Fun in="effect">...</Fun></Expr>
        </Expr>
    </norm>
</Initiates>
```

The example defines an obligation norm which is initiated as a norm "state".

5.3.4 RBSLA Defeasible Layer

In order to deal with conflicts (e.g. positive and negative contradictions) and rules of precedence (rule priorities), as described in section 4.4, RBSLA supports in the defeasible layer defeasible rules "$body \Rightarrow head$" in addition to strict rules (derivation rules of the form "$head \rightarrow body$"). To distinguish both rule types the attribute @*variety* is used in $< Implies >$. The variety attribute is restricted to the values "*strict*" and "$defeasible$". An $< Overrides >$ element defines the priority of rules or rule sets / modules.

Example 48

```
<Overrides>
    <Ind>rule1</Ind>
    <Ind>rule2</Ind>
</Overrides>
```

The example defines that the rule with object id "*rule*1" has higher priority than "*rule*2".

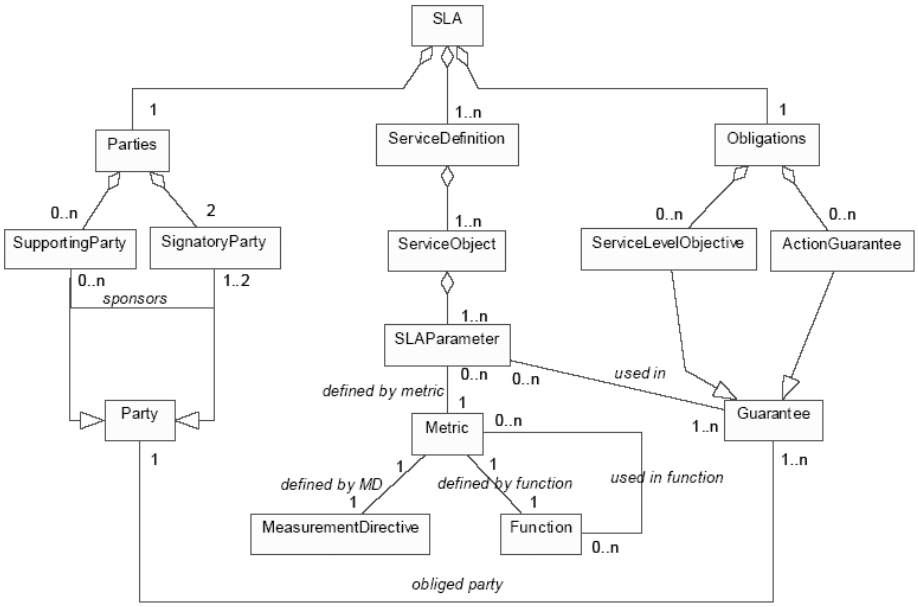

Figure 5.4: WSLA Domain Vocabulary

5.4 Discussion and Conclusion

RBSLA simplifies writing and interchanging SLAs and policies by means of a XML-based mark-up language (platform independent model PIM). It provides means for optimizing/refactoring and validation of rule sets during the transformation into an executable rule language (platform-specific model PSM), e.g., into ContractLog by XSLT based transformation services (see RBSLA project). Remarkably, the RBSLA approach in contrast to existing XML languages in the SLA domain such as WSLA, WSOL, WS-Agreement, is not just a pure syntactical specification language with predefined language constructs, but is a declarative rule-based programming language with an operational and a declarative logic-based semantics for formalizing and implementing arbitrary contract-related functionalities and specifications. This declarative implementation-oriented design provides maximum flexibility and extensibility for SLA representation including the use of different contract vocabularies represented as Semantic Web ontologies such as WSMO, WS-Policy OWL ontology or KAoS and other ontologies such as OWL time, which can be integrated as external webized type systems into the logical SLA rules, giving them a domain-specific meaning. For instance, figure 5.4 shows the domain vocabulary of WSLA [DDK+04].

The vocabulary might be used as a type system for RBSLA rules.

Example 49

```
<Initiates>
    <Expr in="modal" type="wsla:Obligations">
        <Fun>oblige</Fun>
        <Var type="wsla:SignatoryParty">Subject</Var>
        <Var type="wsla:serviceObject">Object</Var>
        <Expr type="wsla:ServiceLevelObjective">
            ...
        </Expr>
    </Expr>
</Initiates>
```

The example initiates an obligation of type $wsla : Obligations$ for a subject of type $wsla : SignatoryParty$ and an object of type $wsla : serviceObject$ to fulfil a $wsla : ServiceLevelObjective$.

Example 50

```
<Atom>
    <Rel>datetime</Rel>
    <Ind type="owlTime:Year">2005</Ind>
    <Ind type="owlTime:Month">11</Ind>
    <Ind type="owlTime:Day">23</Ind>
    <Ind type="owlTime:Hour">10</Ind>
    <Ind type="owlTime:Minute">30</Ind>
    <Ind type="owlTime:Second">0</Ind>
</Atom>
```

The example uses the OWL Time ontology [PH04] as type system to describe the datetime fact $datetime(2005, 11, 23, 10, 30, 0)$.

6 Rule Based Service Level Management (RBSLM) Prototype

6.1 Architecture

Based on the described ContractLog concepts and the RBSLA language, I have implemented the rule based service level management (RBSLM) prototype as a proof of concept implementation. Figure 6.1 shows the general architecture and components of the prototype.

The open-source, backward-reasoning ContractLog rule engine which extends Mandarax (1) (http://mandarax.sourceforge.net/) and Prova [KPS06] serves as inference engine for the LP contract rules. The rules are represented on the basis of the ContractLog framework (2) and are imported into the internal KB of the rule engine using the Prova scripting language [KPS06]. The high-level declarative RBSLA mark-up language (see chapter 5) is used for rule interchange, serialization, tool based editing and verification of rules. XSLT based mappings provided as web-based services transform RBSLA into executable ContractLog rules. A graphical user interface (UI) - the Contract Manager (4) - is used to write, edit and maintain the SLAs which are managed as RBSLA projects. The projects are either persistently stored in the contract base (3) or are imported from distributed web resources. The repository (5) contains typical rule templates and predefined domain specific objects, built-in metrics and contract vocabularies (ontologies) which can be reused in the SLA specifications and loaded together with the projects. During the enforcement and monitoring of the SLAs external data sources, network and system management tools, IT services and business objects (e.g. EJBs) can be integrated (6) (via expressive procedural attachments). Event messages can be communicated between distributed RBSLA entities (rule nodes) and external components using Mule [Mul06] as an enterprise service bus. Finally, the Service Dash Board (7) acts as a runtime environment that visualizes the monitoring results and supports further SLM processes, e.g., reports on violated service levels or metering and accounting functions. Figure 6.2 illustrates the layers of the rule-based approach. Figure 6.3 shows the existing and the newly implemented components of the RBSLM prototype.

Several RBSLM instances might be deployed as distributed web-based services (see figure 6.4). Each service instance runs a ContractLog/Prova rule engine which dynamically imports the distributed RBSLA projects and rule-based SLAs from the web or from the local storage in which the RBSLA projects are

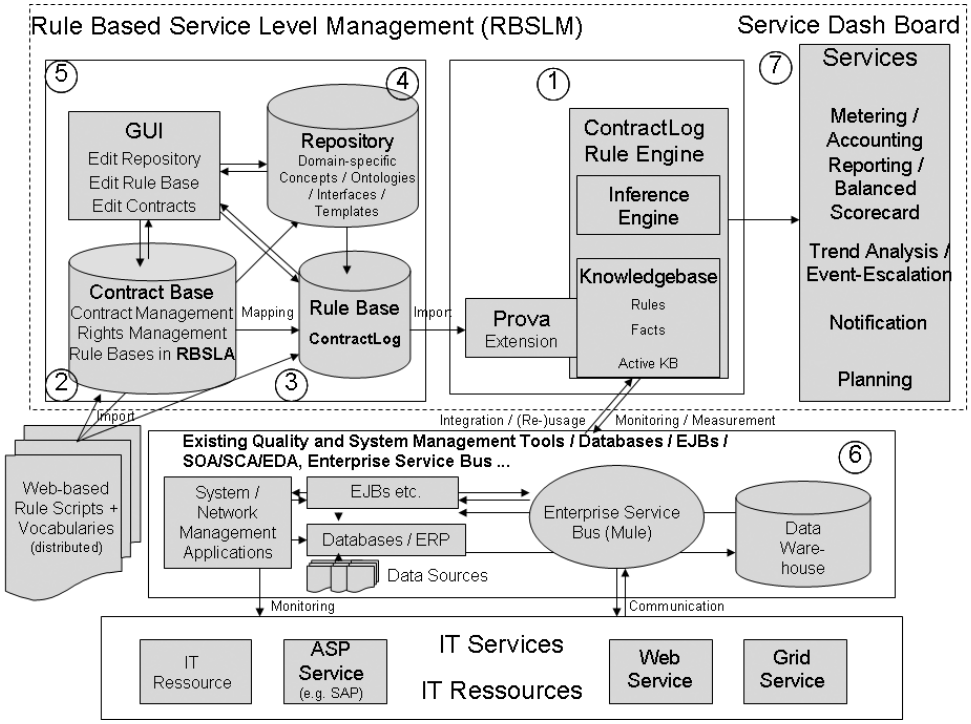

Figure 6.1: Architecture of Rule Based Service Level Management Tool

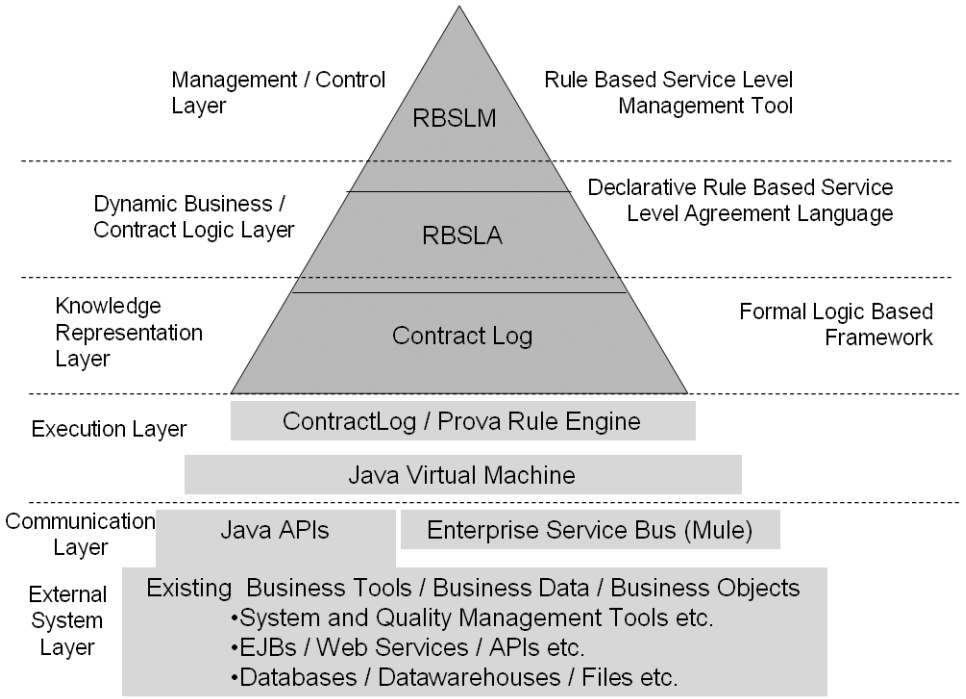

Figure 6.2: Layers of the RBSLM Tool

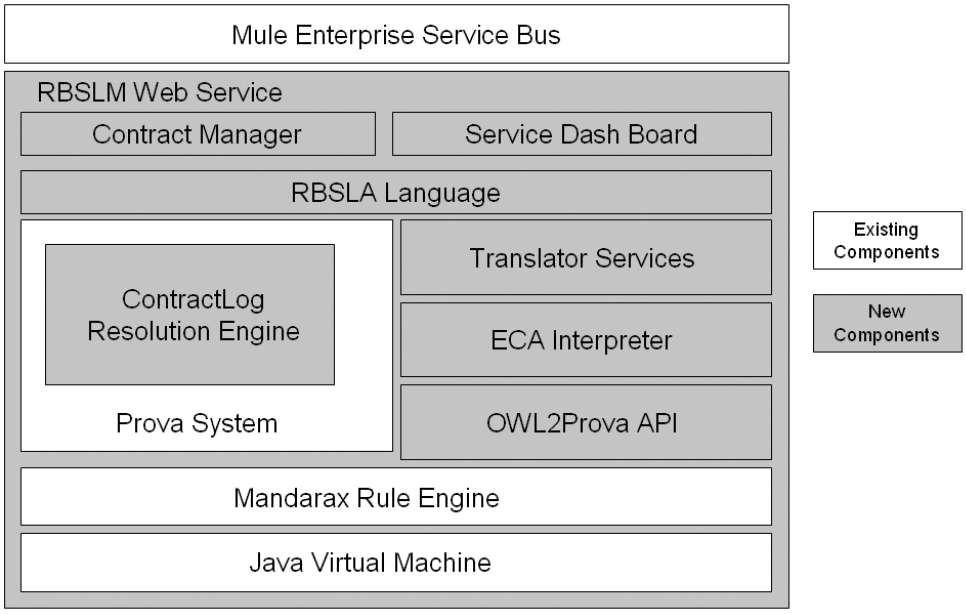

Figure 6.3: RBSLM Service Components

managed. An ESB is used as object broker and asynchronous messaging middleware for the RBSLM services. Different transport protocols such as JMS, HTTP or SOAP (Rest) can be used to transport rule sets, queries and answers as payload of event messages between the internal inference services or external systems and applications. RBSLA is used as common rule interchange format.

In the following sections I will detail the implementation of the RBSLM components.

6.2 ContractLog Rule Engine

In this section I describe the implementation of the ContractLog Rule Engine.

6.2.1 Inference Engine

I have implemented the ContractLog rule engine as an extension of the Prova rule engine and language [KS04] which is loosely based on Mandarax, a Java-based open source backward reasoning derivation rule engine [DH04, Die04]. The ContractLog resolution engine implements the procedural semantics (SLE resolution - see section 4.1.3). The resolution algorithm is written as a Trampoline in order to prevent stack overflows in case of large SLD(NF) trees, which typically occur in recursive implementations of SLD variants of Robinson's resolution algorithm. The knowledge (rules, facts, queries, rule sets/modules) are

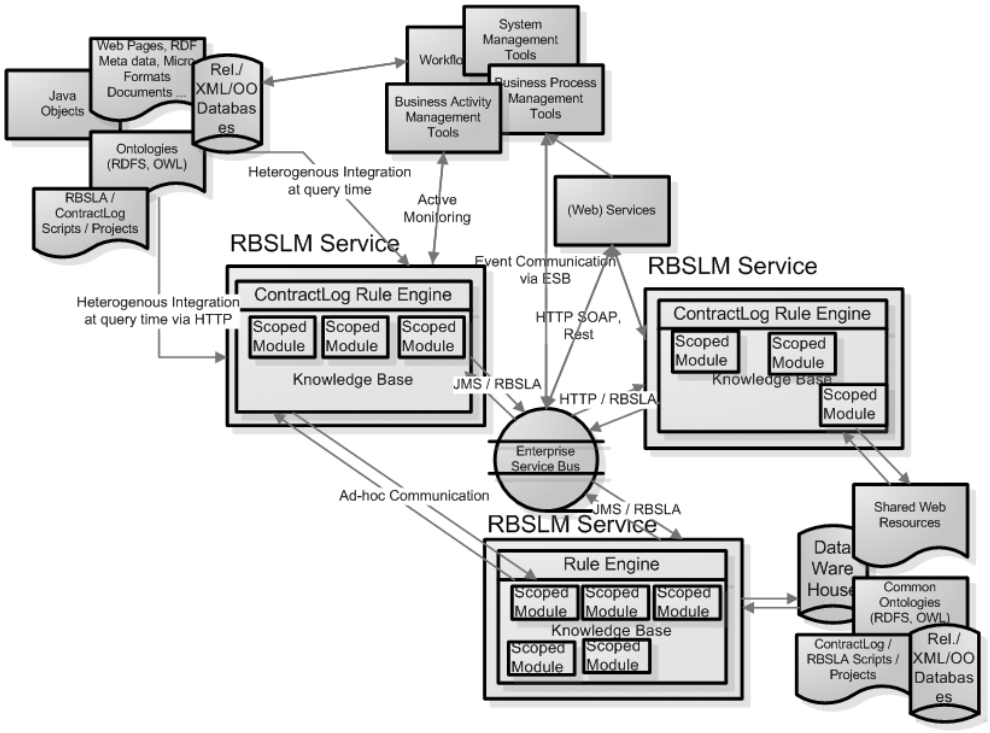

Figure 6.4: Distributed RBSLM Services with ESB

managed as objects with an unique object id, an extended key for indexing and a set of meta data properties (meta annotations) in highly efficient data structures. A cache is used to store memorized goals and their answers for later reuse in SLE resolution. The ContractLog inference service is user-configurable enabling, e.g., the selection of different SLE variants which possibly reduce to standard SLDNF resolution, and the configuration of the cache size and memorization time cached results. Figure 6.5 shows relevant classes and interfaces of the ContractLog inference engine; with the core classes implementing:

- ResolutionInferenceEngine6: the inference engine
- RBSLAKnowledgeBase: the knowledge base with extended clause sets (ExtendedClauseSet)
- RBSLARobinsonUnificationAlgorithm: the typed unification algorithm
- Cache: the linear tabling cache storing goals (CachedGoal)

6.2.2 ECA Interpreter

Figure 6.6 shows the relevant parts of the class diagram of the ECA interpreter which is decoupled from the rule engine via a general wrapper interface (it might be implemented for different rule engines):

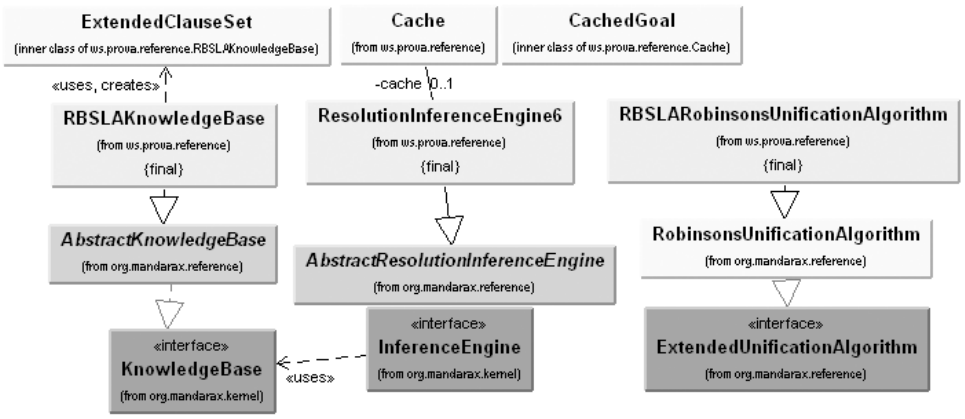

Figure 6.5: Class Diagram of the ContractLog Inference Engine

- KnowledgeBaseWrapper: a general wrapper interface used to decouple the ECA interpreter from arbitrary rule engines
- ActiveKnowledgeBase: provides methods to manage (add, remove) reaction rules and event data
- ActiveKnowledgeTask: abstract class for the definition of tasks to be solved in the ActiveKnowledgeBase
- ActiveRule: represents a reaction rule
- Event, Condition, Action: interface for the parts of ECA rules with evaluation / execution functions
- Daemon, DaemonMultiThreading: The daemon continuously evaluates the reaction rules by querying the active event base and the derivation rule base; according to the defined monitoring schedules and possibly using multiple threads
- EventConditionActionExecutor: Implements the forward directed operational semantics of the ECA paradigm

6.2.3 OWL2PRova API

The OWL2Prova API implements the heterogenous (uses external DL reasoners) and homogenous integration (translation into LP facts and rules) of external Semantic Web vocabularies which might be used as type systems or fact data (property relations) (see section 4.2.3.2). Figure 6.7 illustrates the general architecture for the heterogenous integration exploiting external DL reasoners such as Pellet and Jena.

Figure 6.8 shows relevant parts of the class diagram:

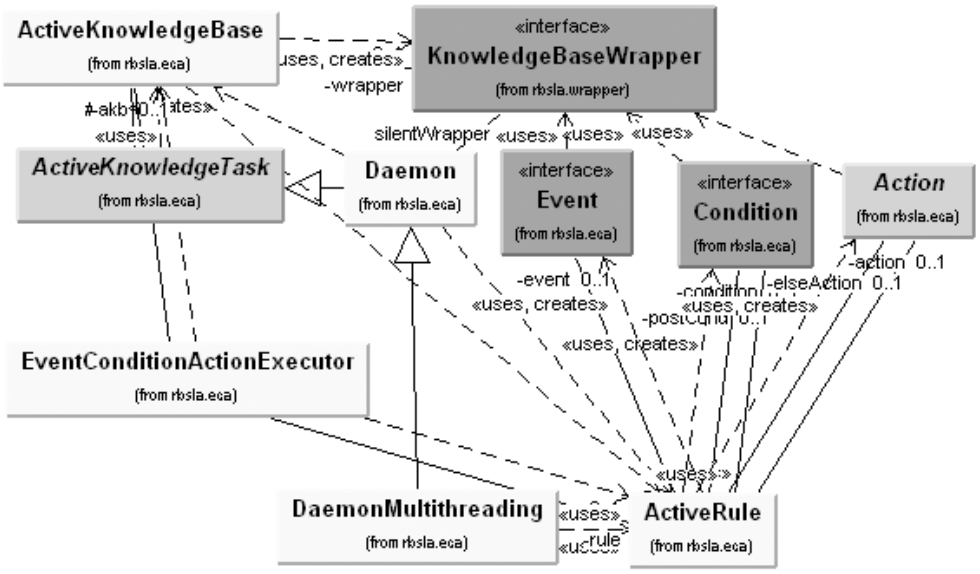

Figure 6.6: Class Diagram of the ECA interpreter

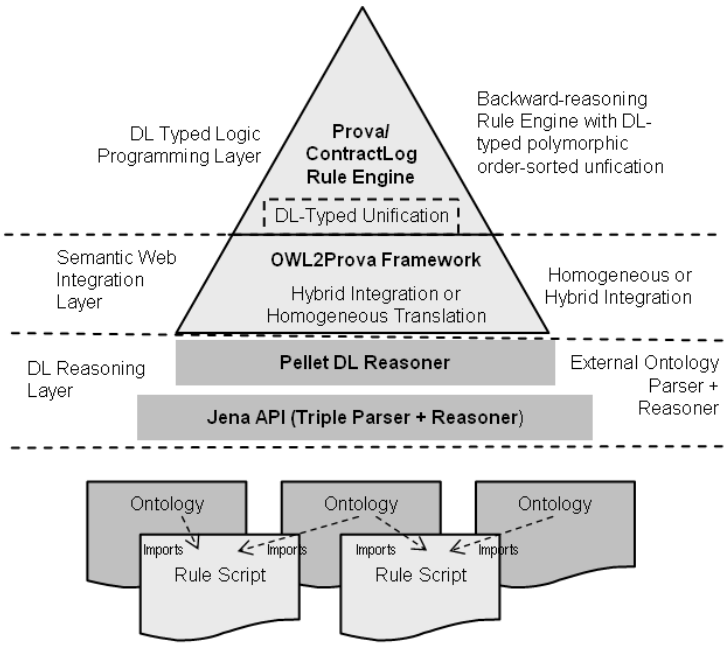

Figure 6.7: The OWL2Prova API

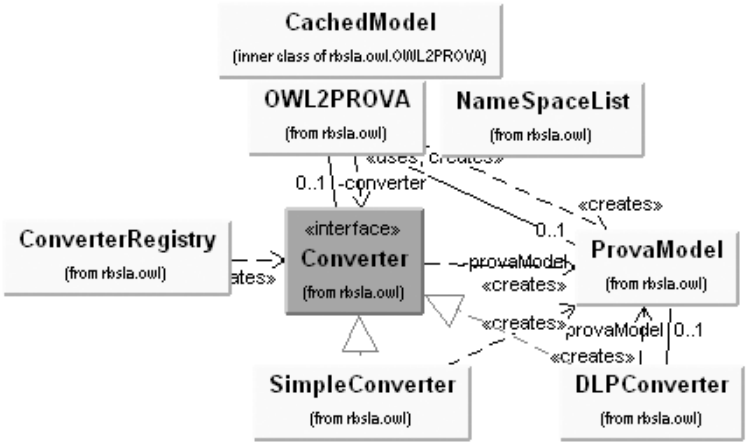

Figure 6.8: Class Diagram of the OWL2Prova API

- OWL2Prova: is responsible for the control flow of all integration processes. Triggers the homogenous converters or the heterogenous rdf triple queries which are passed to a specified external DL reasoner.
- Converter: interface for converter implementations which translate ontologies into a specific LP representation format, e.g. a user-define forma (SimpleConverter) or homogenous description logic programs [GHVD03] (DLPConverter)
- ConverterRegistry: A registry to register converters
- ProvaModel: stores the (translated) ontology model
- CachedModel: a configurable cache implementation which memorizes previously answered queries to the ontology model
- NameSpaceList: defines default namespaces and stores the name spaces of the queried ontology models
- DLP Package: includes different converters to translate ontologies into homogenous DLPs
- Defeasible Package: includes different converters to translate ontologies into defeasible homogenous DLPs

6.2.4 Translator Services

The translator services of RBSLA/ContractLog translate the platform specific execution rule language Prova/ContractLog into the platform-independent rule interchange format RBSLA and vice versa. For the translation of RBSLA XSLT stylesheets are used. For the serialization of Prova/ContractLog into RBSLA an ANTLR grammar and respective parser and "lexer" implementations are used

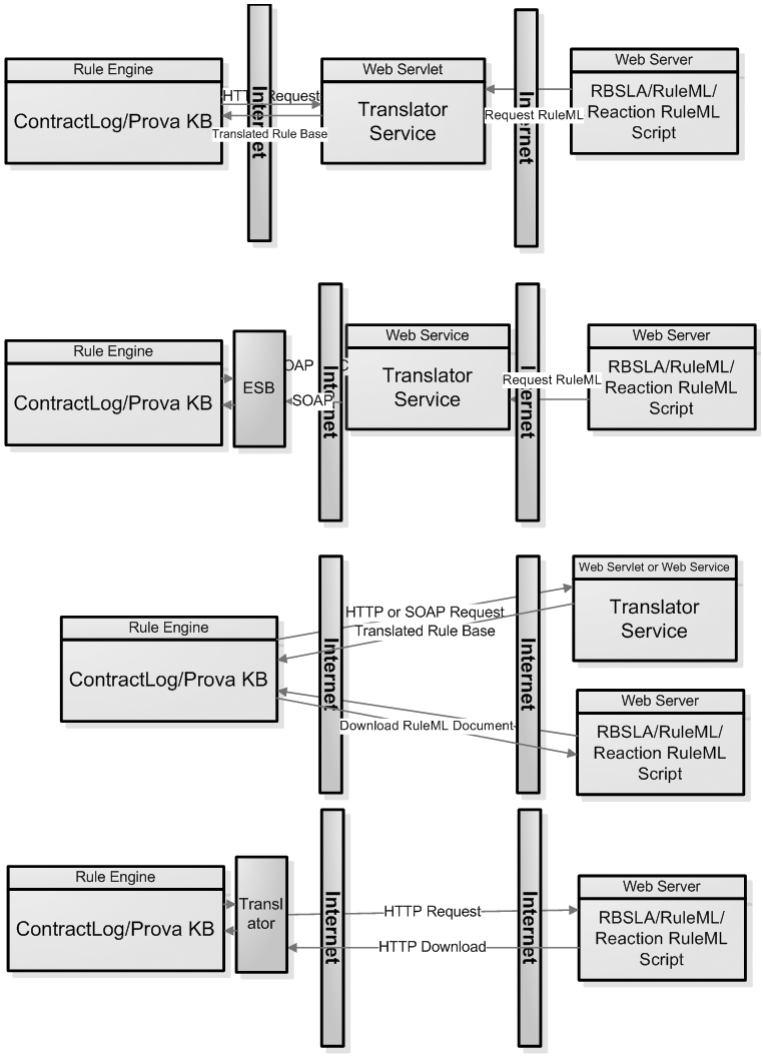

Figure 6.9: Class Diagram of the OWL2Prova API

to parse and transform Prova scripts into an abstract syntax tree (AST) which then is transformed into the XML representation.

The bi-directional translators are provided as internal built-ins integrated into ContractLog's update functions, but also as online services for automated consumption (HTTP requests, SOAP RPCs) or human consumption via web forms. Figure 6.9 illustrates four different distribution scenarios.

6.3 RBSLM Tool

The RBSLA tool implements an UI and project management system for the collaborative engineering, project-specific management and runtime execution

of distributed (web-based) SLA contracts. The tool is provided in the RBSLA project as a stand-alone version and as a web-based version (download by Java Web Start). It splits into two basic components, the contract manager (CM) and the service dashboard (SD), which are coupled by a mediator implementation. The mediator is responsible for the management of RSLA projects.

The implementation follows a Model View Controller (MVC) pattern which is additionally extended with an Observer pattern to monitor changes in the resources of the model. The View is implemented by the CM UI and SD UI. The mediator implements the Controller and provides the API to load and save the project-specific Model.

6.3.1 Contract Manager

The CM provides a graphical editor and management UI for RBSLA projects. Its main features are:

- provides a complete engineering, management and testing UI for centralized (local) or decentralized RBSLA projects
- wizard for writing free-hand rules, XML-tree wizard or template-driven wizard
- project-centered management of different project-specific rule languages, domain vocabularies and knowledge repositories
- abstraction from the platform-specific execution syntaxes into platform-independent rule interchange and serialization syntaxes
- support for different roles such as domain experts, rule engineers and programmers, business practitioners
- visual support for engineering, testing, maintaining, updating and executing/querying the rule base
- means to define a domain-specific, human-oriented natural rule language (by named templates implemented by domain experts and rule programmers) which allows non-technical business practitioners to easily define rule-based SLA specifications
- decoupling from the underlying execution environment, such that different possibly web-based inference services (rule engines) can be used
- a local persistence layer (provided by the mediator) storing the project components or at least web references to it
- import and export features to import and extract parts of RBSLA projects and deploy them on the Internet

Each RBSLA project consists of:

- project meta data

- the domain specific platform independent XML rule interchange syntax (e.g. a XML syntax such as RuleML, Reaction RuleML, RBSLA) which is modelled as an abstract syntax
- domain vocabularies / ontologies / type systems
- translators to transform the platform independent rule syntax into the platform specific execution syntax and vice versa (e.g. XSLT-based, programmatic translator, ANTLR)
- API references to an underlying execution environment / inference service for querying the rule base
- a rule base consisting of several possibly further nested named and meta-annotated rule modules (representing e.g. contracts) which include rules and facts of the various kinds together with queries.
- a repository of predefined knowledge templates, i.e. partially filled and reusable rule structures which have been given human-oriented name in natural-language
- test suites which consists of sets of test cases for testing certain functionalities and properties of the rule base and its modules
- runtime views (used in the runtime environment)

That is, a RBSLA project stores all domain-specific knowledge including, e.g., the underlying PIM rule interchange syntax, the template based natural language syntax and the domain-specific vocabularies. This qualifies the Contract Manager to be a project-specific management and engineering tool and to be a general graphical rule editor for various rule languages and rule engines by specifying:

1. an abstract syntax model for the used XML rule interchange and serialization language
2. translators from and into the execution rule syntax
3. a wrapper implementation wrapping the API of an underlying rule engine or a web client for remote procedure calls on a web-based inference service

An user-defined natural language representation of the rule language can be implemented by named templates which describe partially pre-defined knowledge pieces such as rule templates, fact templates, built-in templates. These templates form the basic language constructs for a human-oriented rule language.

Example 51

```
"if status of customer is __
    then customer gets discount of ___ ."
```

```
<Implies>
    <head>
        <Atom>
            <Rel>discount</Rel>
            <Var>Customer</Var>
            <Ind>     </Ind>
        </Atom>
    </head>
    <body>
        <Atom>
            <Rel>status</Rel>
            <Var>Customer</Var>
            <Ind>    </Ind>
        </Atom>
    </body>
</Implies>


"status of __ is __."

<Atom>
    <Rel>status</Rel>
    <Ind>     </Ind>
    <Ind>     <Ind>
</Atom>
```

The example first shows the template and then the XML description. The first template defines a discount rule which has an empty filler for the discount value and the customer status. That is, business practitioners might customize the rule, possibly several times with different discount value and customer status levels. The second template denotes facts about the customer status. The binary "status" template can be completed to different atoms which can be used as facts or within rules.

This template-driven repository approach supports different types of users: the business practitioner, the rule and system engineers/programers and the domain experts.

- The domain experts together with the rule programmers which have a background in logic are responsible for the implementation of the domain and SLA project models. That is, they fill the repository with domain specific measurement, monitoring and computing functions, interface implementations to existing databases or system tools, references to existing business objects as well as rule templates, test cases and other domain specific concepts (e.g. vocabularies). Additionally, they specify test cases together with certain test data to be used for verification and validation

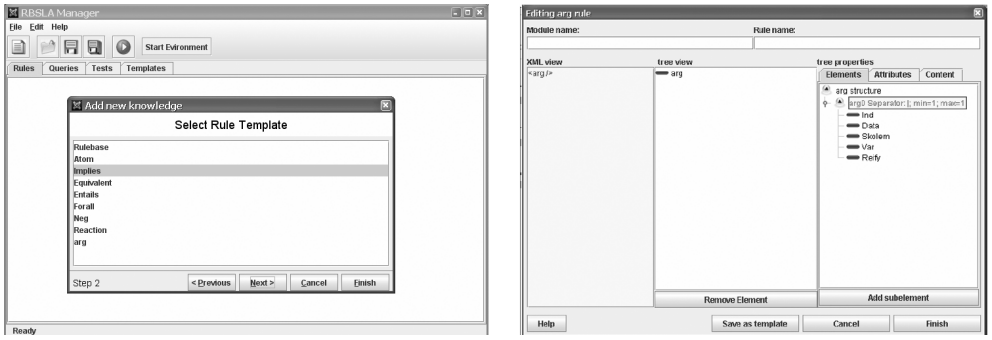

Figure 6.10: Contract Manager User Interface

of contract rules in order to ensure the correct usage and a high-quality of the SLA rule sets.

- The business practitioners make use of the predefined templates to write and maintain the contract rules. They do not need to know any implementation details of functions, objects or interfaces nor do they need to have a complete overview of all the rules in the contract system; meaning they do not know what their effect is on existing rules when a rule is added or changed. The test cases safeguard the authoring of rules and allow validation of complete rule sets and contracts to detect anomalies like inconsistencies, incompleteness or redundancies referring to the intended goals.

The left picture in figure 6.10 shows the rule wizard to add new knowledge to the KB. The right picture shows the XML tree views to create and edit a rule. The graphical implementation uses Java Swing.

6.3.2 Mediator

The mediator is responsible for the internal management of RBSLA projects and for the control and communication with the inference and translator services. Although the control flow will be initiated by the UIs, the mediator implements the Controller for managing a project, e.g., loading and saving the project resources. Since the complete model is project-specific new models might be loaded by the mediator. The complete project management is executed by the mediator. Observed changes in the project are communicated to the registered observers of the UI views, which are then updated accordingly. To manage the project states, such as "project loaded/saved/changed", the mediator applies a State pattern. Figure 6.11 describes the UML state diagram for the four mediator states. This workflow is used to control the user workflow, e.g., the user is requested to save the project before it is closed in case the project has been changed.

Figure 6.12 shows the class diagram of the mediator implementation.

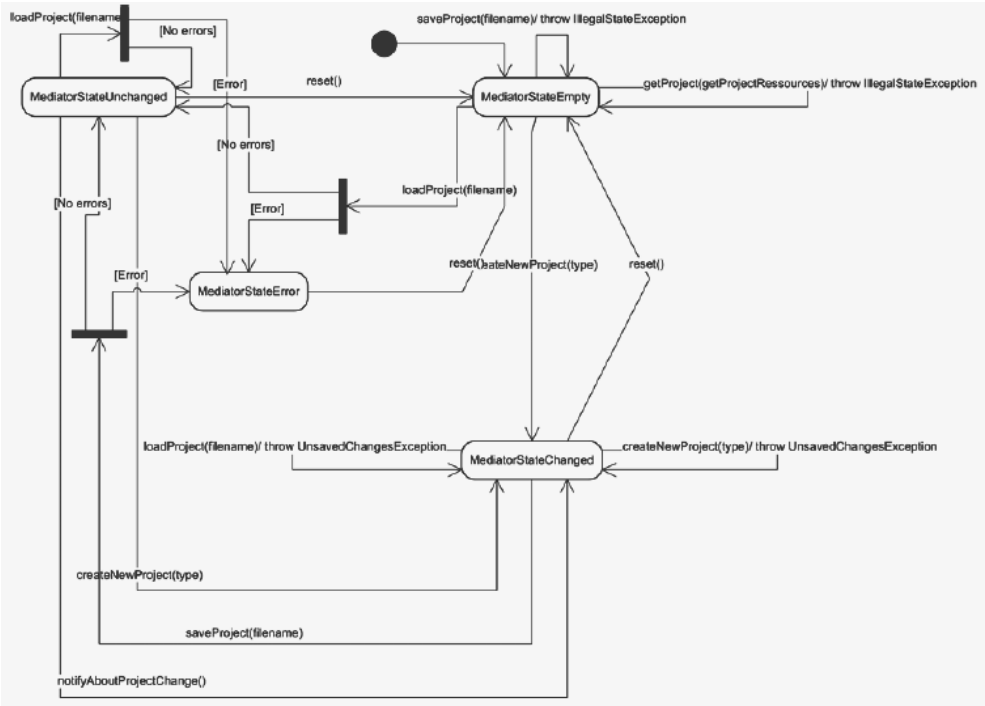

Figure 6.11: UML State Diagram for the Mediator

- Mediator: provides the API methods to manage a project
- Project: abstract class for a project which manages the project resources
- ProjectRBSLA: abstract RBSLA project class; (currently) implemented either as semi-structured XML documents (ProjectXML) or as Prova scripts (ProjectProva)
- Resource: abstract class for object project resources (ResourceObject) or text resources (ResourceText); All resources implement load and save methods and an Observer interface which informs the project observer in case of changes in the resource model
- MediatorState: abstract class for the State pattern to control the project states

6.3.3 Service Dashboard

The Service Dashboard acts as runtime environment for the SLA. It visualizes the monitoring information and SLA metrics. Different and adaptable visualization views are provided and new user-defined views can be easily plugged into the framework in order to satisfy the needs of different users. For instance, a service administrator needs more detailed technical and textual views showing, e.g., low-level QoS metrics of the monitored services, whereas the SLA quality

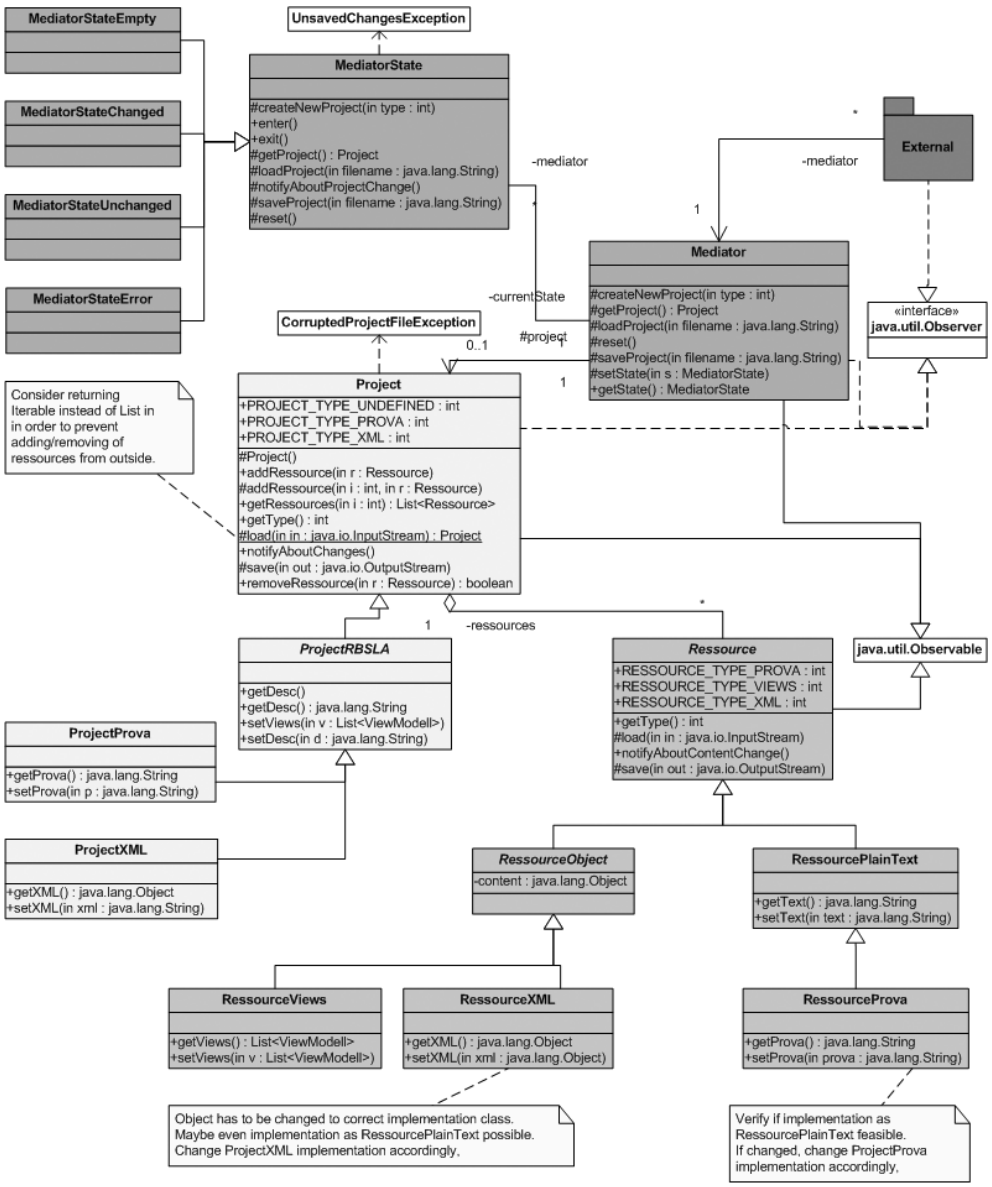

Figure 6.12: Class Diagram of the Mediator

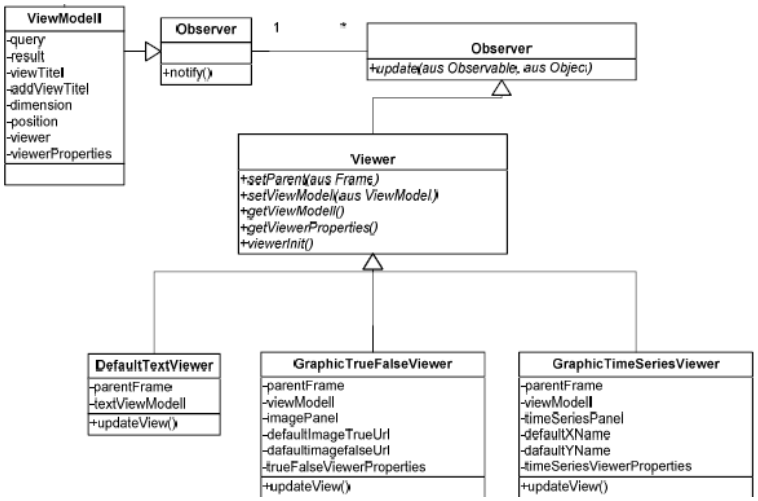

Figure 6.13: Class Diagram of the Service Dashboard Views

manager or the CEO will need more high-level business-oriented views for BAM and BPM such as graphical SLA score cards or incident reports. Figure 6.13 shows the core class diagram of the extensible view implementation.

- Viewer: interface for view implementations following the Observer pattern
- DefaultTextViewer, GraphicTrueFalseViewer, GraphicTimeSeriesViewer ...: Implemented views using e.g. the jFree chart library

Figure 6.14 gives an impression of different views and their configuration properties in the SD.

Beside these tracking, monitoring and reporting views the SD visualizes and explains the inference results such as the derivation trees and allows to explore the active project and its resources. This gives a better understanding of the derived results and the dynamic changes of the SLA project during execution and enforcement phase, hence increasing trust and helps, in addition to the test-case based approach, to find failures in the rule-based SLA specification by visually debugging the execution traces. Figure 6.15 shows some example views of the SD project explorer interface.

The SD provides also specialized user interfaces for the particular formalisms of the ContractLog KR such as the defeasible logic and the integrity test logic, e.g., in order to find integrity violations by performing integrity tests and automatically solve conflicts through transformation of conflicting rules into defeasible prioritized rules. Figure 6.16 shows the class diagram of the defeasible test interface of the SD.

Figure 6.17 describes the sequence diagram for performing integrity tests and solving conflicts via defeasible transformation and prioritization.

Figure 6.14: Different Views in the Service Dashboard

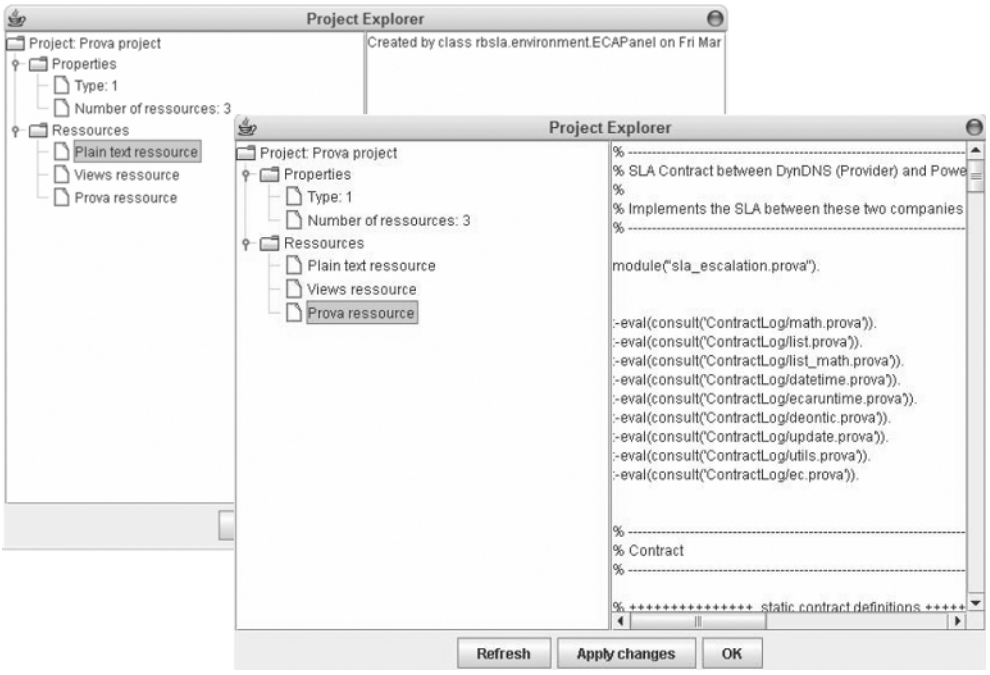

Figure 6.15: Service Dashboard Project Explorer

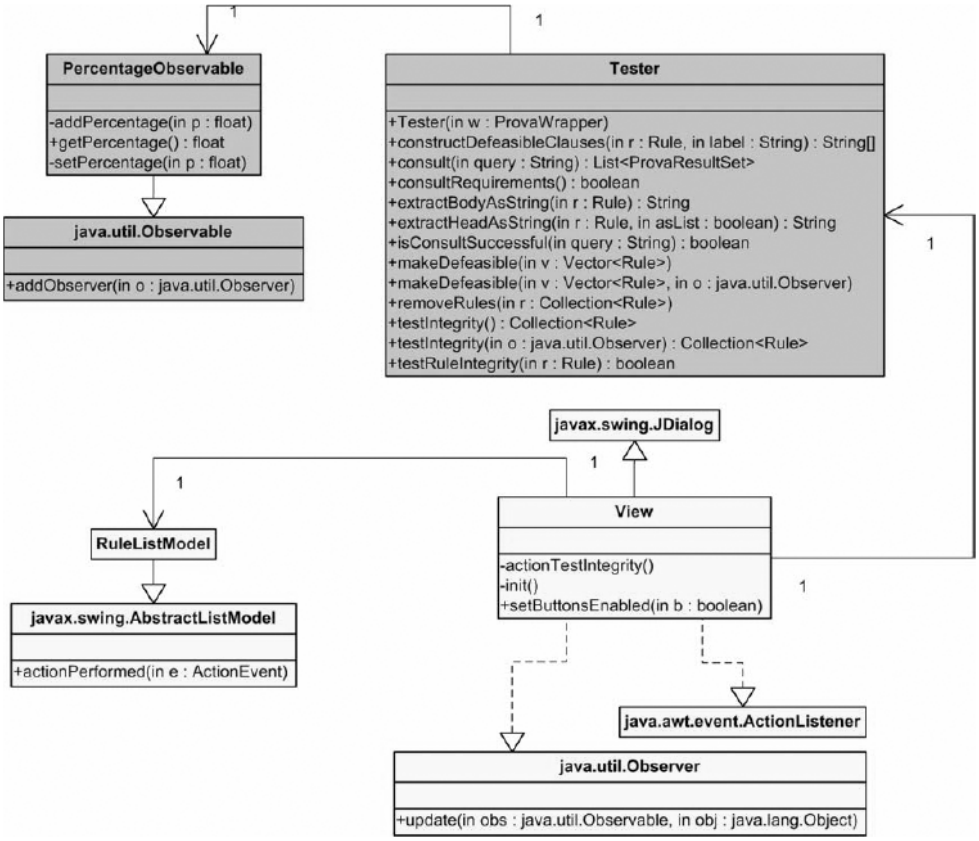

Figure 6.16: Class Diagram of the Defeasible Test Interface

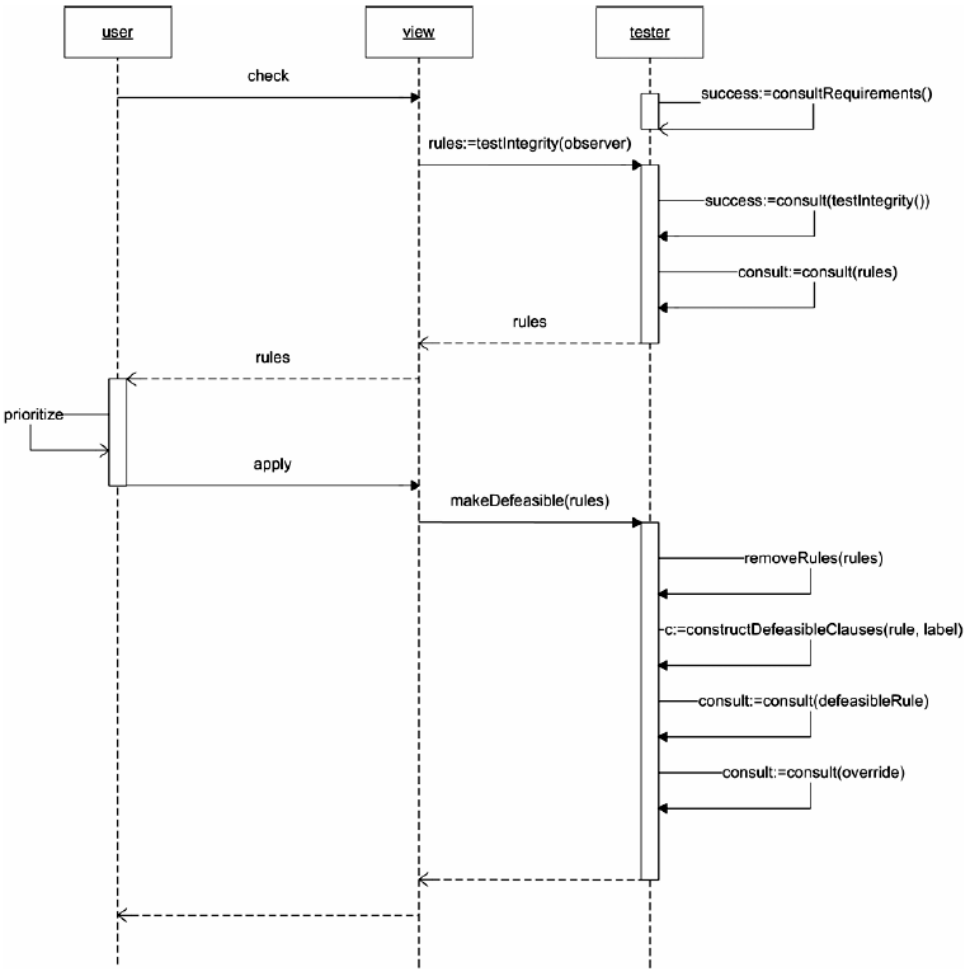

Figure 6.17: Sequence Diagram for Integrity Testing and Defeasible Conflict Resolution

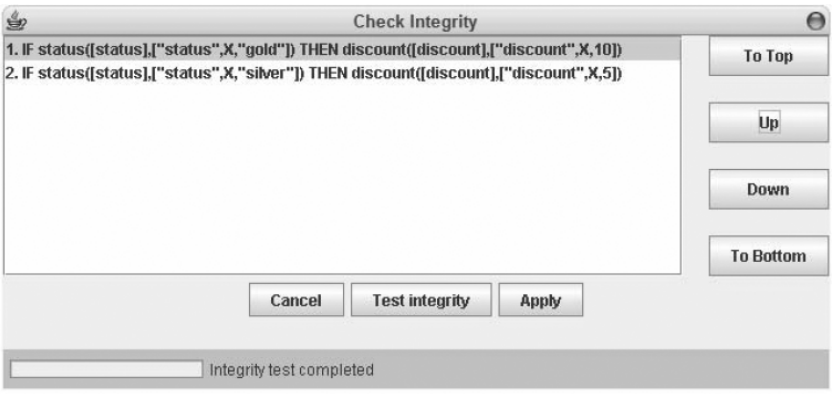

Figure 6.18: Integrity Test Interface

Figure 6.18 shows the user interface for the integrity tests and defeasible prioritization.

6.4 Enterprise Service Bus

The RBSLM tool is build on top of Prova's support for Mule [Mul06] as an ESB in order to seamlessly handle interactions with other applications and systems using disparate CEP technologies, transports and protocols. The ESB also allows to deploy several Prova/ContractLog rule engines as a highly distributable inference services installed as web-based endpoints in the Mule object broker and supports the Reaction RuleML based communication between them. That is, beside the direct ad-hoc communication with external business component via their (web) service APIs which are called by procedural attachments from the ContractLog rules, the ESB provides a highly scalable and flexible application messaging framework to communicate synchronously but also asynchronously with external systems and internal RBSLA components. In this section I describe the integration of Mule into the RBSLM tool and its usage to deploy RBSLA/RBSLM/ContractLog as basis for enterprise application techniques such as SOA, SCA, CEP, EDA or BAM.

Mule is a messaging platform based on ideas from ESB architectures, but goes beyond the typical definition of an ESB as a transit system for carrying data between applications by providing a distributable object broker to manage all sorts of service components. Figure 6.19 shows the architecture of the mule manager which is the central component in Mule.

The model encapsulates and manages the runtime behavior of a Mule server instance. The transport provider enables Mule components to send and receive information over a particular protocol, repository messaging or other technology. Mule supports a great variety of transport protocols such as JMS, HTTP, SOAP, TCP. Autonomous components such as JavaBeans or components from

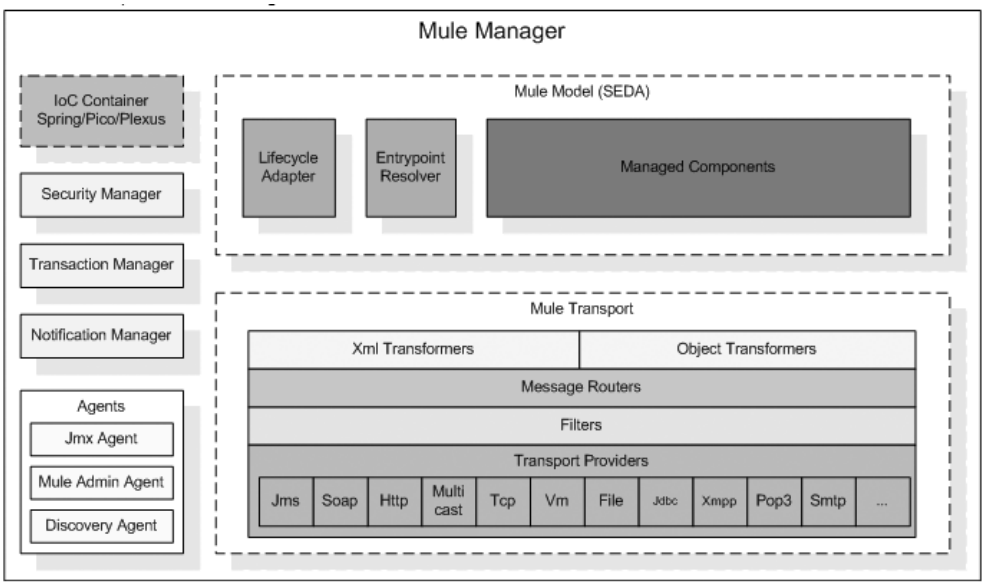

Figure 6.19: Mule Manager Architecture [Mul06]

other frameworks are managed within the object broker and configured to exchange inbound and outbound event messages through registered routers to the components' endpoint addresses. The properties and payload of the event messages will be used and/or manipulated by one or more components. The three processing modes of Mule are [Mul06]:

- Asynchronously: many events can be processed by the same component at a time in different threads. When the Mule server is running asynchronously instances of a component run in different threads all accepting incoming events, though the event will only be processed by one instance of the component.
- Synchronously: when a UMO Component receives an event in this mode the whole request is executed in a single thread
- Request-Response: this allows for a UMO Component to make a specific request for an event and wait for a specified time to get a response back

The object broker follows the Staged Event Driven Architecture (SEDA) pattern [WCB01]. The basic approach of SEDA is to decomposes a complex, event-driven application into a set of stages connected by queues. This design decouples event and thread scheduling from application logic and avoids the high overhead associated with thread-based concurrency models. That is, SEDA supports massive concurrency demands on web-based services and provides a highly scalable approach for asynchronous communication.

The integration of Mule into RBSLM extends Provas' AA capabilities to send and receive messages [KPS06]. Figure 6.20 shows a simplified breakdown of the integration of Mule and RBSLM.

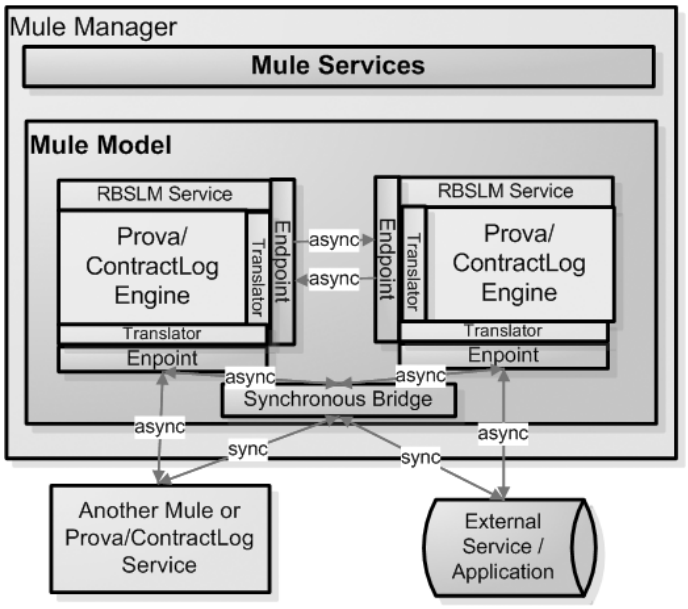

Figure 6.20: Integration of Mule into RBSLM

Several instances of RBSLM services which at their core run a Prova/ContractLog rule engine are installed as Mule components which listen at configured endpoints, e.g., JMS message endpoints, HTTP ports, SOAP server/client addresses or JDBC addresses. RBSLA/Reaction RuleML is used as a common platform independent rule interchange format between the Prova/ContractLog rule engine (and possible other rule execution environments/engines). The RBSLA translator services (see figure 6.4) are used to translate inbound and outbound event messages from RBSLA/Reaction RuleML into Prova/ContractLog syntax and vice versa. Similar translation services can be installed for the communication with other rule engines and external systems. The payload of a transported event message reaches from queries and answers to complete rule and fact bases which are interchanged between the services and applications. That is, possible distributed scenarios for RBSLM might follow conversational query-answer paradigm (including semantic and pragmatic context information) up to interchange of executable rule code which is uploaded to available web-based inference services. The example below shows an *outbound* Reaction RuleML message for the ESB from $client1@labbichler1$ with a *query* $ping(service1@labbichler1)$ as payload.

Example 52

```
<RuleML xmlns="http://www.ruleml.org/0.91/xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ruleml.org/0.91/xsd
http://ibis.in.tum.de/research/ReactionRuleML/0.1/rr.xsd">
```

```
<Message mode='outbound' directive='query'>
    <oid>
        <Ind>converstation1234</Ind>
    </oid>
    <protocol>
        <Ind>esb</Ind>
    </protocol>
    <sender>
        <Ind>client1@labbichler1</Ind>
    </sender>
    <content>
        <Atom>
            <Rel>ping</Rel>
            <Ind>service1@labbichler1</Ind>
        </Atom>
    </content>
</Message> </RuleML>
```

The large variety of transport protocols provided by Mule can be used to transport the RBSLA/Reaction RuleML messages to the registered endpoints or external objects. Usually JMS is used for the internal communication between distributed RBSLM service instances, while HTTP and SOAP is used to access external web services. The usual processing style is asynchronous using SEDA event queues. However, sometimes synchronous communication is needed. For instance, to handle communication with external synchronous HTTP clients such as Web browsers where requests, e.g. by a Web from, are send through a synchronous channel. In this case the implemented synchronous bridge component dispatches the requests into the asynchronous messaging framework and collects all answers from the internal service nodes, while keeping the synchronous channel with the external application open. After all asynchronous answers have been collected they are send back to the still connected external application.

Diagram 6.21 shows the core implementation classes of the Mule integration.

- ESBMangers: wraps the Mule Manager and provides methods to start and stop the ESB service
- ProvaUMOImpl: implementation for ContractLog/Prova instances as Mule UMO components
- SyncUMOImpl: implementation of the synchronous bridge between synchronous and asynchronous messaging framework
- RuleML2ProvaTranslator: translator implementation for translating RBSLA/RuleML into Prova execution language (uses XSLT)

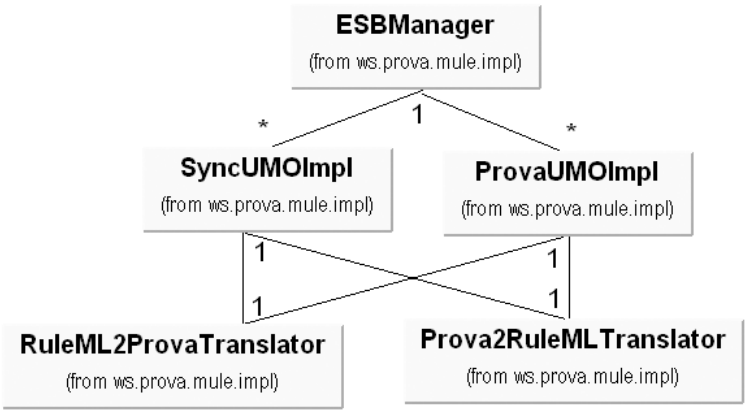

Figure 6.21: Classes for Integration of Mule into RBSLM

- Prova2RuleMLTranslator: translator implementation for translating Prova into RBSLA/RuleML (based on ANTLR)

6.5 Discussion and Conclusion

In this chapter I have described the RBSLM reference implementation of the RBSLA project, which serves as a proof-of-concept implementation on the one hand, but also addresses industrial requirements to integrate the RBSLA approach into the existing enterprise service landscape and common industrial technologies and make it really usable for humans and machines.

The RBSLM user interface, consisting of the contract manager and the service dashboard, is mainly intended for the engineering, management, maintenance and execution of RBSLA projects . The tool is project-centered which allows configuring it to particular IT service management domains and provides adaptable support for different roles which are involved during the life-cycle of a RBSLA project. I loosely followed the Logical User-Centered Interactive Design (LUCID) methodology developed by Cognetics, Corporation [Cog05] which is a methodology for user-centered UI design with the central components "ease-of-understanding" and "ease-of-use". For instance, the adaptable and highly configurable views of the SD which are stored as project resources and the support for different roles (e.g. experts implement easy to use and understand template language and aggregating views for practitioners and decision makers) are intended to fulfil these two goals.

The RBSLM translator services allow to transform the rule-based SLA specifications from the PSM format (Prova/ContractLog) into an interchangeable XML-based PIM format (RBSLA/RuleML). The integration of these webized XML RBSLA documents into existing service description languages such as WSDL is straightforward by simply referencing their URI/URL. That is, RBSLA templates or completed specifications can be applied to describe the con-

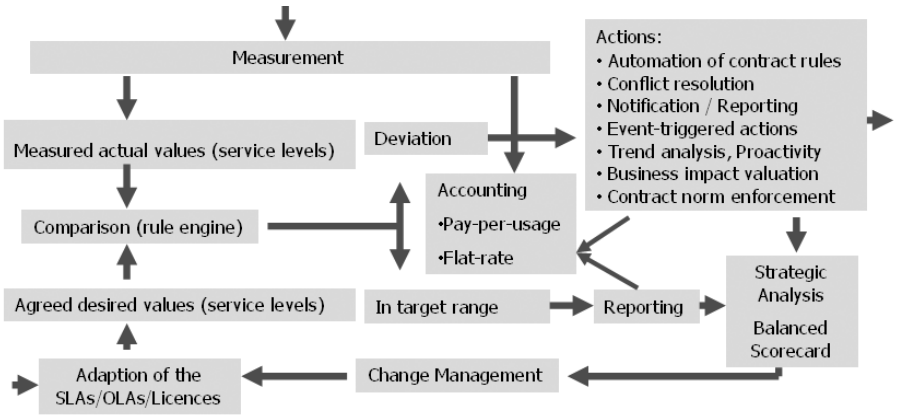

Figure 6.22: Employing RBSLM for IT Service Level Management

tractual rights and obligations and non-functional quality properties of e.g. web services, extending them beside their syntactic description with a precise and clear semantic description (which leads to Semantic Web Services).

Finally, the integration of an ESB messaging middleware enhances RBSLM to a highly distributable service-oriented and event-driven architecture. As such, RBSLM services and ContractLog/Prova inference services can be integrated and interoperate with the existing enterprise service tools such systems and network management tools, data warehouses and workflow/BPM tools. By the capability of ContractLog to perform rule-based complex event processing and complex rule reactions typical workflow and process patterns can be implemented, which qualifies it to work in combination or as an alternative to languages such as BPEL.

In summary, RBSLM provides the technical infrastructure and tools for rule-based IT SLM processes as specified in ITIL and BS1500 and underpins enterprise management models such as business activity management. Figure 6.22 outlines the processes of using RBSLM for IT SLM.

7 Evaluation

ContractLog provides compact, declarative knowledge representation for contractual agreements based on logic programming. This enables high levels of flexibility and easy maintenance as compared to traditional SLA approaches. Generic rule engines allow for an efficient execution of SLAs. In the following, I evaluate the KR approach by means of theoretical analysis, experiments and on an example derived from common industry use cases. Since the ContractLog KR is a huge logical framework incorporating many logical formalisms with many expressive variants, which can be used in combination with each other, a pure theoretical analysis would be not sufficient in order to meet the practical design goal and the real-world application of rules in the SLA domain. The use of real test programs and use cases with real data provides insights that are not to be found through theoretical analysis. Accordingly, the main focus in this chapter is on the experimental evaluation and the demonstration of adequacy by means of a use case.

7.1 Theoretical Worst Case Complexity and Expressiveness

I first introduce the necessary terminology: A relation $r(t)$ on the set of ground terms of a LP language is definable under $\models$ (sceptical) semantics if there exists a program P and predicate symbol $p(t)$ in the language such that for every ground term t, $P \models r(t)$ or $P \models \neg r(t)$. As usual in Datalog, I further distinguish a database of facts (EDB) and a set of input rules (IDB) for inferring additional information. More precisely, the predicate symbols in a LP language are divided into a set of extensional relations and a set of intensional relations. The facts EDB are formed from the extensional predicates and ground terms, while the rule heads are formed from the intensional predicates.

Definition 141 *(**Data and Program Complexity**) The worst case complexity is given by the complexity of checking whether $EDB \cup IDB \models \overline{A}$ for a variable rule base (IDB), ground atoms $\overline{A}$, and extensional facts (EDB). Fixing the intensional relations leads to data complexity which is the complexity of checking whether the ground atoms $\overline{A}$ are entailed by $IDB \cup EDB$ for a fixed set of intensional predicates and a variable set of extensional predicates, i.e., a fixed rule base IDB and a variable input fact base EDB. Data complexity can be viewed as a function of the size of EDB. The program complexity is the*

complexity of checking $IDB \cup EDB \models \overline{A}$ for variable rule base IDB and ground atoms $\overline{A}$ over a fixed extensional database EDB.

Complexity of Logic Programs As discussed in section 3.3.2 there are multiple different semantics for logic programs. In general, the formalisms implemented within the ContractLog KR are designed to be usable in different LP environment having different semantics such as extended well-founded semantics or stable model semantics resp. answer set semantics. Accordingly, the general complexity depends on the LP class (e.g., LP without variables (i.e., propositional) LP without functions (i.e., Datalog), LP with functions, LP with strong negation (i.e., extended LP), LP with disjunction (i.e., disjunctive LP)) and its semantics (e.g., completion semantics, stratified semantics, well-founded semantics, stable model semantics, answer set) used to interpret the ContractLog KR. Logic programming has been proven to have efficient complexity results under certain expressive restrictions such as for normal LPs with finite functions, Datalog restriction, bounded number of variables, which are commonly met in practical SLA formalizations. The following theorem 9 extracted from [DV97] gives a brief review of complexity results derived for major LP classes and semantics.

Theorem 9 *The decision problem for propositional LPs (with negation) is:*

(a) P-complete ($O(n)$) under stratified semantics
(b) co-NP complete for COMP
(c) co-NP complete for STABLE
(d) P-complete ($O(n2)$ resp. $O(n)$ for acyclic LPs) under WFS

Datalog is:

(e) data complete in co-NP under STABLE and program complete in co-NEXPTIME
(f)data complete in P under WFS and program complete in DEXPTIME under WFS
(g) data-complete in P and program complete in DEXPTIME for stratified Datalog

Normal LP with functions are:

(h) r.e.-complete for full LP but NEXPTIME complete for non-recursive LPs
(i) $\prod_1^1$-complete for full LPs under WFS and STABLE

Proof 2 *see (a) [DG84], (b) [KP87], (c) [Mar91], (d) [VGRS91, VG89], (e) [Mar91, Sch95], (f) [VGRS91, VG89], (g) [ABW88], (h) [Tae77, DV97], (i) [Sch95]*

The formalisms and meta programs in the ContractLog KR are in general interpretable by different LP semantics. Since the semantics of choice in the ContractLog inference engine is WFS complexity is polynomial [VGRS91]. Note that for typical mathematical functions and equalities the hybrid ContractLog KR uses external procedural implementations (Java code) which are called by procedural attachments. The resulting values (objects) are bound to variables and used in the further refutation attempt based on resolution and unification. For a survey of unification algorithms see [BS01].

Theorem 10 *The general unification problem is P-complete under logspace reductions.*

Proof 3 *see e.g., [DKM84]*

The typed logic with a typed unification in ContractLog extends the decision problem of deciding whether two terms are unifiable to sub-type checking and computing the greatest lower bound under the type order. The resolution algorithm in ContractLog computes the most general unifier of two terms and the greatest lower bound of the types of these terms. I use an efficient representation of terms within the KB data structures with extended key indexing which improves the basic Robinson algorithm to almost linear time (see section 4.1.3 and section 4.2.4.2). For a survey of complexity results for description logics, in case of external Semantic Web type systems, see [Zol05].

ECA Processor Algorithm

Algorithm 1 *Operational semantics of ECA processor:*

1. *The ECA processor collects all reaction rules (represented as facts) from the KB via a query* $eca(T, E, C, A, P, EL)$?
2. *Then it processes each constituent term in each reaction rule according to the forward directed ECA paradigm.*

Theorem 11 *The forward directed execution of reaction rules within the ECA processor takes linear time* $O(n)$ *where* n *is the number of reaction rules.*

Proof 4 *(sketch)*

- *step (1) is* $P-$ *complete under logspace reductions: All reaction rules are represented as ground positive facts in the KB. Hence, the complexity* $EDB \cup IDB \models eca(T, E, C, A, P, EL)$ *is equivalent to the data complexity for a variable input database* EDB *and a fixed normal program* P*, where the set of reaction rules* $\overline{ECA} = \{eca_1, ..eca_n\} \subseteq EDB$*. The problem of query answering with ground facts (without rules) reduces to the problem of unifying the query with the facts (here the reaction rules) which is P-complete under logspace reductions (see proof 3) and* $O(n)$ *in ContractLog for all facts in* EDB*.*

- *step (2) takes $O(n)$ time since operationally selecting an empty thread and processing the constituent parts takes the same amount of time for each rule.*

Note the time to evaluate the reaction rule parts is neglected here since the evaluation time of the reaction rules' is dependent on their underlying implementation, i.e., depends on the semantics of the underlying LP used to define functionalities in terms of derivation rules.

Event Calculus Algorithm

Algorithm 2 *Query answering in the Event Calculus (holdsAt)*

1. *first all occurred events have to queried by $happens(Ei, T1)$?*
2. *then all events Ei which happened before the queried time T need to be selected: $T1 < T$*
3. *then all events Ei which initiate the queried fluent need to be selected: $initiates(Ei, F, T1)$*
4. *and finally it needs to be checked whether there exists another event Et which terminates the fluent between $T1$ and T: $notclipped(T1, Fluent, T)$*

Theorem 12 *In the case of absolute times and total ordering of events the worst case data complexity of query answering whether a fluent holds or not at a given time point has linear complexity $O(n)$ where n is the number of events.*

Proof 5 *(sketch) The general complexity of the EC is the complexity of checking whether a certain fluent holds $holdsAt(Fluent, t)$ based on the happened events and the defined EC inference rules initiating respectively terminating this fluent. Hence, the cost is measured by data complexity for a fixed set of initiates/terminates rules and a variable set of happens facts as the number of accesses to the knowledge base to unify the happens facts during the refutation attempt. The complexity is given as a function of the number n of occurred events.*

- *step (1) takes linear time $O(n)$ and has $O(n)$ output size. The query unifies directly with the happens facts and succeeds n times as much events are recorded in the KB. Hence, the complexity reduces to the unification problem and has cost $O(n)$ (see proof 3).*
- *since I impose absolute times and total orderings of events in the ContractLog meta program step (2) is a simple date comparison function (solved by procedural code) which has constant cost.*
- *step (3) takes $O(n)$, since the initiates resp. terminates are typically represented as facts which a queried by a bound query (i.e., reduces to the unification problem - see proof 3.*

- *step (4) again needs to query all events n which cost $O(n)$ to derive Et and compare the occurrence dates and the termination rules with constant costs.*

Accordingly, the overall complexity is linear $O(n)$ by the number of events n in the EDB.

Note: Performance of query answering in the EC can be significantly increased by reducing the number of events. One way to achieve this is to use typed events so that a query on a particular fluent only needs to consider events of a particular type and not all recorded events (based on typed unification - see complexity of unification). Another way is to use external databases which populate the KB with the recorded events at query time. Here pre-selections can be applied, using highly optimized query languages such as SQL (via where clause) to select a limited subset of all occurred events, e.g., all event which occurred in the last month, the last hour, before the time t etc., so that reasoning is done over a much smaller number of events.

Defeasible Logic

Theorem 13 *Propositional Defeasible Logic has been proven to have linear complexity.*

Proof 6 *[Mah01]*

In ContractLog I do not constrain on propositional defeasible logic but allow variables. The defeasible rules and supporting auxiliary rules are meta interpreted, i.e., a defeasible theory Φ_{Defl} is translated into a $LP(\Phi_{Defl})$ meta program as described in section 4.4.4. A defeasible rule is translated into a derivation rule testing all body literals defeasibly and testing the strict integrity as well as the defeasible integrity (not defeated). Furthermore two auxiliary rules are built which are used in the defeasible meta program to decide whether a rule holds, i.e., all body literals can be derived defeasible, and the rule is not definitely blocked. In summary the transformation of a defeasible theory into a LP increases the size of rules by 3 per defeasible rule and the size of literals per rule by $6 + 3k$ where k is the number of body literals in a defeasible rule. The transformation is done in a pre-processing step during translation from RB-SLA to ContractLog. The deduction of strict rules in extended LPs, which are represented in the classical sense as "plain" derivation rules without any meta predicates, needs no proofs of opposers, in contrast to defeasible provability. The inference rules to compute conclusions of the input theory and show that a query is provable defeasibly is done by the meta program described in section 4.4.4. The basic steps are:

Algorithm 3 *Defeasible Meta Program*

1. *check whether the conclusion H, the head literal of a defeasible rule, will violate the integrity of any strict knowledge ($testIntegerity(H)$) (hypothetical test). The meta program iterates over all integrity constraints in the KB and tries to derive any mutual exclusively strict rule which strictly overturns the defeasible conclusion L.*

2. *Each body literal L_b in the body of each defeasible rule with head H needs to be checked defeasibly by using it as a subgoal ($defeasible(L_b)$).*

3. *Defeasible "attacker" must be considered ($not(defeated(r, H))$).*

   - *The meta program proves that $defeated(r, H)$ can not be derived, i.e., similar to strict integrity test as described in step 1, it iterates over all integrity constraints where H is a member and tries to find any not blocked defeasible opposer (i.e., strongly negated) using the supporting auxiliary rules ($neg(blocked(defeasible(Opposer) :- ...)$. If no opposer can be found the conclusion H is not defeated.*

   - *If there is an unblocked opposer it proves whether the opposer is overruled or not using the priority definitions for modules and single rules.*

Discussion:

1. Step (1) tries to derive strictly (i.e., in the classical sense) all other integrity goals (queries on the rule heads of other strict rules / facts) in the defined integrity constraints where H is a member. Hence, the cost of this step is dependent on the complexity of deriving all other strict goals. Accordingly the complexity is given by theorem 9, that is data-complete in P under WFS with or data complete in $co - NP$ for STABLE with Datalog restriction.

2. Step (2) has the complexity of defeasible provability for each L_b as discussed for H, i.e., again step 1 to step 3.

3. Step (3) first starts a proof attempt on all mutual exclusive defeasible opposers by

   - (a) It uses the supporting auxiliary meta rules ($neg(blocked(defeasible(Opposer) :- ...$). which starts a strict integrity test for the opposer as defined in step 1 (i.e., cost of step 1) and proves all opposers' body literals defeasibly, i.e., has cost of defeasible provability for each body goal (again step 1 to step 3). If an opposer can be derived, i.e., the mutual exclusive literal is $not(neg(blocked()))$, the meta program tests whether the opposer is overridden by H.

   - (b) The meta program tries to derive and overrides facts where H, the label of the rule with head H or the module id where H is a member overrides the opposer resp. the opposers' rule label or module id. This is solved by query on the facts with unification complexity, i.e., $O(n)$.

In summary, the described meta algorithm to defeasibly prove a goal includes several queries on the KB which are strongly dependent on the complexity of deriving strict knowledge as defined in theorem 9.

7.2 Experimental Results

To experimentally analyze the performance of the ContractLogs' formalisms wrt query answering in different logic classes (e.g., propositional, Datalog, normal) I adapt a benchmark test suite for defeasible theories [ABMR00] to the ContractLog KR and extend it to evaluate different inference properties, rule types and logic program classes. The experiments are designed to test different performance/scalability aspects of the inference engine and the logical formalisms. I run the performance tests on an Intel Pentium 1.2 GHz PC with 512 MB RAM using Windows XP. I use different benchmark tests to measure the time required for proofs, i.e., I test performance (time in CPU seconds to answer a query) and scalability (size of test in number of literals). Various metrics such as number of facts, number of rules, overall size of literals, indicating the size of complexity of a particular benchmark test might be used to estimate the time for query answering and memory consumption.

The first group of benchmarks evaluates different inference aspects of the ContractLog KR such as rule chaining, recursion, and unification. In the *chains test* a chain of n rules and one fact at its end is queried. In the *dag test* a directed-acyclic tree of depth n is spanned by the rules in which every literal occurs recursively k times. In the *tree test* a fact is at the root of a k-branching tree of depth n in which every literal occurs once. Each test class is performed for propositional LPs without variables and Datalog LPs with one variable. The tests further distinguish strict LPs, i.e., normal LPs with only strict (normal) derivation rules and defeasible LPs with defeasible derivation rules and compare inference for both classes with goal memoization and without goal memoization. I use the total number of literals as a measure of problem size, which is much larger for the defeasible theories due to the meta program transformations, as described in section 4.4.4. The second and third group of experiments test the ECA processor for (re)active rule processing and complex event processing based on Event Calculus formulations. For more detailed description of the benchmark test suite see appendix C. Figure 7.1 shows a summary of selected performance results in CPU seconds.

Each experiment was performed several times (10 experiments per benchmark test) and figure 7.1 shows the average results with a range of $\pm 10\%$ for the measured values in the series of tests. The table shows the computation time to find an answer for a query.

Due to the needed variable unifications and variable substitutions in the derivation trees, the Datalog tests are in general more expensive than the propositional tests and goal memoization adds small extra costs to rule derivations.

Test	Size		No Memoization		Memoization	
	Strict	Defeasible	Strict (Propos. / Datalog)	Defeasible (Propositional / Datalog)	Strict (Propositional / Datalog)	Defeasible (Propos. / Datalog)
chains(n)	2001	11001	0.01 / 0.07	4 / 7.6	0.05 / 0.17	5,7 / 7,8
	5001	27501	0.03 / 0.17	12.8 / 25	0,15 / 0.47	18 / 24,3
	10001	55001	0.07 / 0.3	40 / 70	0,4 / 1.05	59 / 75
	20001	110001	0.15 / 0.62	127 / 250	1,25 / 2,62	170 / 200
dag(n,k)						
n=3 k=3	39	156	0.01 / 0.06	0.54 / 0.89	0.005/ 0.01	0.05/ 0.05
n=4 k=4	84	324	2.2 / 7.7	81 / 120	0.01 / 0.03	0.06/ 0.07
n=10k=10	1110	3810	- / -	- / -	0.05 / 0.16	0.2 / 0.32
tree(n,k)						
n=3 k=3	79	248	0.01 / 0.02	0.04/0.04	0.001/0.001	0.04/ 0.05
n=4 k=3	281	761	0.015/ 0.03	0.09/0.1	0.005/0.006	0.08/ 0.11
n=8 k=3	19681	62321	0.17 / 0.5	- / -	0.02/0.04	0.09/ 0.14
			Update Time		Execution Time	
$eca_{plain}(n)$	1000		0.4		0.005	
	2500		1.1		0.01	
	5000		2.5		0.015	
	10000		4.3		0.02	
$ec_{HoldsAt}(n)$	1002		3.3			
	2502		6.8			
	5002		14.6			
	10002		28.7			

Figure 7.1: Performance Evaluation

In the chains test, where subgoals are never reused, the experiments with memoization are slower than without memoization due to the caching overhead. However, the advantages of goal memoization can be seen in the tree and dag tests which (recursively) reuse subgoals. Here goal memoization leads to much higher performance and scalability (large problem sizes can be solved). As expected, the defeasible experiments are slower, due to the much larger problem sizes and the meta program interpretations which need several KB subgoal queries. Goal memoization reduces duplication of work, e.g., to test strict integrity of defeasible rules. The reaction rule experiments distinguish between update time for querying the KB for ECA rules and processing time for executing the ECA rules. The experiments reveal an increase in time linear in the problem size. The event calculus tests also show linear time increase in the problem size, which here is the number of occurred events stated as happens facts which initiate resp. terminate a fluent.

In summary, the experiments reveal high performance of the ContractLog formalisms even for larger problem sizes with thousands of rules and more than 10,000 literals, which suggests the approach also for industrial-size applications. I have formalized typical real-world SLAs from different industries in ContractLog within several dozens up to hundreds of rules and much smaller literal sizes (see for example the RIF / RuleML use cases) which can be efficiently executed and monitored within milliseconds. Moreover, the hybrid approach in ContractLog allows outsourcing lower-level computations and operational functionalities to procedural code and specialized external systems.

7.3 Use Case Revisited - Adequacy / Expressiveness

In this section, I illustrate the adequacy of the rule-based SLA representation approach, in particular with respect to expressiveness of the ContractLog KR, by means of a use case example derived from common industry SLAs. I revive the example SLA described in section 2.4 and present a formalization of a selected subset in ContractLog, namely the monitoring schedules, the escalation levels and the associated roles, as well as the following SLA rules:

Example 53 *"The service availability will be measured every $t_{schedule}$ according to the actual schedule by a ping on the service. If the service is unavailable and is not maintained then escalation level 1 is triggered and the process manager is informed. Between 0-4 a.m. the process manager is permitted to start servicing which terminates any escalation level. The process manager is obliged to restart the service within time-to-repair, if the service is unavailable. If the process manager fails to restore the service in time-to-repair (violation of obligation), escalation level 2 is triggered and the chief quality manager is informed. The chief quality manager is permitted to extend the time-to-repair interval up to a defined maximum value in order to enable the process manager to restart the service within this new time-to-repair. If the process manager fails to restart the service within a maximum time-to-repair escalation level 3 is triggered and the control committee is informed. In escalation level 3 the service consumer is permitted to cancel the contract."*

For the reason of understandability I present an untyped formalization, but appropriate external domain vocabularies/ontologies might be used as type systems for term typing of the rules in order to give them an additional semantic meaning. Use cases and examples demonstrating this integration of external domain vocabularies can be found in the RBSLA distribution.

The formalization in ContractLog is as follows:

```
% service definition
service(http://ibis.in.tum.de/staff/paschke/rbsla/index.htm).

% role model and escalation levels
initially(escl_lvl(0)). % initially escalation level 0
% if escalation level 1 then process_manager
role(process_manager) :- holdsAt(escl_lvl(1),T).
% if escalation level 2 then chief quality manager
role(chief_quality_manager) :-  holdsAt(escl_lvl(2),T).
% if escalation level 3 then control committee
role(control_committee) :- holdsAt(escl_lvl(3),T).

% time schedules standard, prime, maintenance and monitoring intervals

% before 8 and after 18 every minute
schedule(standard, Service):- systime(datetime(Y,M,D,H,Min,S)),
```

```
        less(datetime(Y,M,D,H,Min,S), datetime(Y,M,D,8,0,0)),
        interval(timespan(0,0,1,0), datetime(Y,M,D,H,Min,S)),
        service(Service), not(maintenance(Service)).  % not maintenance
schedule(standard, Service):- systime(datetime(Y,M,D,H,Min,S)),
        more(datetime(Y,M,D,H,Min,S), datetime(Y,M,D,18,0,0)),
        interval(timespan(0,0,1,0), datetime(Y,M,D,H,Min,S)),
        service(Service), not(maintenance(Service)).  % not maintenance

% between 8 and 18 every 10 seconds
schedule(prime, Service):- sysTime(datetime(Y,M,D,H,Min,S)),
        lessequ(datetime(Y,M,D,H,Min,S),datetime(Y,M,D,18,0,0)),
        moreequ(datetime(Y,M,D,H,Min,S),datetime(Y,M,D,8,0,0)),
        interval(timespan(0,0,0,10), datetime(Y,M,D,H,Min,S)) ,
        service(Service).

% between 0 and 4 if maintenance every 10 minutes
schedule(maintenance, Service) :-
        sysTime(datetime(Y,M,D,H,Min,S)),
        lessequ(datetime(Y,M,D,H,Min,S),datetime(Y,M,D,4,0,0)),
        interval(timespan(0,0,10,0), datetime(Y,M,D,H,Min,S)) ,
        service(Service), maintenance(Service). % servicing

% initiate maintenance if permitted
initiates(startServicing(S),maintenance(S),T).
terminates(stopServicing(S), maintenance(S),T). % terminate maintenance
happens(startServicing(Service),T):-
        happens(requestServicing(Role,Service),T),
        holdsAt(permit(Role,Service, startServicing(Service)),T).

% ECA rule: "If the ping on the service fails and not maintenance then
% trigger escalation level 1 and notify process manager, else if
% ping succeeds and service is down then update with restart
% information and inform responsible role about restart".
eca(
    schedule(T,S),
    not(available(S)),
    not(maintenance(S)),
    escalate(S),_,
    restart(S)). % ECA rule

available(S) :- WebService.ping(S).     % ping service
maintenance(S) :- sysTime(T), holdsAt(maintenance(S),T).
escalate(S) :-  sysTime(T), not(holdsAt(unavailable(S),T)),
    add("outages","happens(outage(_0),_1).",[S,T]),% add event
    role(R), notify (R, unavailable(S)).  % notify
restart(S) :- sysTime(T), holdsAt(unavailable(S),T),
    add("outages","happens(restart(_0),_1).",[S,T]),% add event
    role(R), notify(R,restart(S)). % update + notify

% initiate unavailable state if outage event happens
initiates(outage(S),unavailable(S),T).
terminates(restart(S),unavailable(S),T).
```

```
% initiate escalation level 1 if outage event happens
terminates(outage(S),escl_lvl(0),T).
initiates(outage(S),escl_lvl(1),T).

% terminate escalation level 1/2/3 if restart event happens
initiates(restart(S),escl_lvl(0),T).
terminates(restart(S),escl_lvl(1),T).
terminates(restart(S),escl_lvl(2),T).
terminates(restart(S),escl_lvl(3), T).

% terminate escalation level 1/2/3 if servicing is started
initiates(startServicing(S),escl_lvl(0),T).
terminates(startServicing(S), escl_lvl(1),T).
terminates(startServicing(S), escl_lvl(2),T).
terminates(startServicing(S),escl_lvl(3),T).

% permit process manager to start servicing between 0-4 a.m.
holdsAt(permit(process_manager,Service, startServicing(Service)),
datetime(Y,M,D,H,Min,S)):-
    lessequ(datetime(Y,M,D,H,Min,S),datetime(Y,M,D,4,0,0)).
% else forbid process manager to start servicing.
holdsAt(forbid(process_manager,Service, startServicing(Service)),
datetime(Y,M,D,H,Min,S)):-
    more(datetime(Y,M,D,H,Min,S),datetime(Y,M,D,4,0,0))..

% derive obligation to start the service if service unavailable
% oblige process manager
derived(oblige(process_manager, Service , restart(Service))).
holdsAt(oblige(process_manager, Service , restart(Service)), T) :-
    holdsAt(unavailable(Service),T).

% define time-to-repair deadline and trigger escalation level 2
% if deadline is elapsed
time_to_repair(tdeadline). % relative time to repair value
trajectory(escl_lvl(1),T1,deadline,T2,(T2 - T1)) .  % deadline function
derivedEvent(elapsed).
happens(elapsed,T) :- time_to_repair(TTR),
    valueAt(deadline,T, TTR).
terminates(elapsed, escl_lvl(1),T).% terminate escalation level 1
initiates(elapsed, escl_lvl(2),T). % initiate escalation level 2

% trigger escalation level 3 if (updated)
%time-to-repair is > max time-to-repair
happens(exceeded,T) :- happens(elapsed,T1), T=T1+  ttrmax.
terminates(exceeded,escl_lvl(2),T). initiates(exceeded,
escl_lvl(3),T).

% service consumer is permitted to cancel the contract in escl_lvl3
derived(permit(service_consumer, contract , cancel)).
holdsAt(permit(service_consumer, contract , cancel), T) :-
    holdsAt(escl_lvl(3),T).
```

The following example demonstrates the integration of external domain ontologies.

Example 54

```
schedule(standard, Service:wsla_ServiceObject):-
    systime(datetime(Y:owlTime_Year,M:owlTime_Month,D:owlTime_Day,
            H:owlTime_Hour,Min:owlTime_Minute,S:owlTime_Second)),
    less(
        datetime(Y:owlTime_Year,M:owlTime_Month,D:owlTime_Day,
            H:owlTime_Hour,Min:owlTime_Minute,S:owlTime_Second),
        datetime(Y:owlTime_Year,M:owlTime_Month,D:owlTime_Day,
            owlTime_Hour:8,owlTime_Minute:0,owlTime_Second:0)),
    interval(timespan(
      owlTime_Day:0,owlTime_Hour:0,owlTime_Minute:1,owlTime_Second:0),
      datetime(datetime(Y:owlTime_Year,M:owlTime_Month,D:owlTime_Day,
      H:owlTime_Hour,Min:owlTime_Minute,S:owlTime_Second)),
    service(Service:wsla_ServiceObject),
    not(maintenance(Service:wsla_ServiceObject)).  % not maintenance
```

The respective typed RBSLA serialization of this rule is as follows:

Example 55

```
<Implies>
  <head>
    <Atom>
      <op><Rel>schedule</Rel></op>
      <Ind>standard</Ind>
      <Var type="wsla:ServiceObject">Service</Var>
    </Atom>
  </head>
  <body>
    <And>
    <Atom>
      <op><Rel>systime</Rel></op>
      <Expr>
        <Fun in="no">datetime</Var>
        <Var type="owlTime:Year">Y</Var>
        <Var type="owlTime:Month">M</Var>
        <Var type="owlTime:Day">D</Var>
        <Var type="owlTime:Hour">H</Var>
        <Var type="owlTime:Minute">Min</Var>
        <Var type="owlTime:Second">S</Var>
      </Expr>
    </Atom>
```

```
    ...

    <Atom>
        <op><Rel>service</Rel></op>
        <Var type="wsla:ServiceObject">Service</Var>
    <Atom>
    <Naf>
        <Atom>
            <op><Rel>maintenance</Rel></op>
            <Var type="wsla:ServiceObject">Service</Var>
        </Atom>
    </Naf>
    </And>
  </body>
</Implies>
```

By simple queries, the actual escalation level and the rights and obligations each role has in a particular state can be derived from the rule base. And, the maximum validity interval (MVI) for each contract state, e.g., the maximum outage time, can be computed. The MVIs can be used to compute service levels such as the average availability. The ECA processor of the ContractLog framework actively monitors the reaction rules. Every t_{check} according to the actual schedule it pings the service via a procedural attachment, triggers the next escalation level if the service is unavailable and informs the corresponding role. To illustrate this process, I assume that the service becomes unavailable at time $t1$. Accordingly, escalation level 1 is triggered and the process manager has time-to-repair $t2$. After $t2$ escalation level 2 is triggered and the chief quality manager adapts the time-to-repair to $t3$ and then to $t4$ until the maximum threshold max. time-to-repair is reached at time point $t4$. After $t4$ the SLA is violated and escalation level 3 is initiated which permits the service consumer to terminate the contract. By querying the rule engine, this status information can be dynamically derived at each point in time and used to feed periodical reports, enforce rights and obligations or visualize monitoring results on quality aspects in the Service Dashboard. Figure 7.2 shows this process in a dashboard view.

7.4 Discussion

Since this dissertation is on the interplay of several programming paradigms and research disciplines I discuss adequacy of my RBSLA approach using established criteria in these fields.

Knowledge Representation Adequacy Criteria The presented rule based SLA representation approach and the implemented logical formalisms in the ContractLog framework fulfil typical KR adequacy criteria as stated in section 1.2:

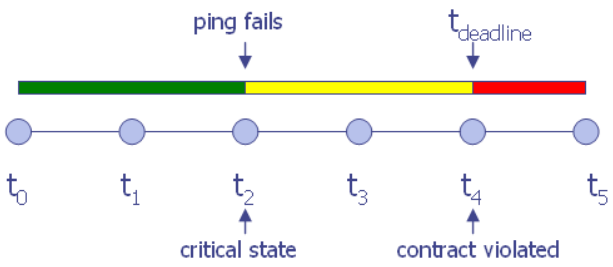

Figure 7.2: Contract tracking

- A proof for epistemological adequacy wrt expressiveness is given by the examples in this dissertation. In particular the use case example in section 7.3, and the various examples and use cases provided by the RBLSA/-ContractLog distribution demonstrate adequate expressiveness for SLA representation.
- Due to the hybrid approach in ContractLog which calls highly optimized external object-oriented code and query languages such as SQL to solve certain tasks the benefits of both worlds can be combined and the inferences can run with limited resources. Scalability for large problem sizes has been experimentally demonstrated in section 7.2.
- The formalisms of the ContractLog KR have been carefully selected and implemented to fulfil algorithmic and logical-formal adequacy in real-world settings of the SLA domain. The worst-case complexity of the main formalisms has been discussed in section 7.1 and an experimental proof has been given in section 7.2.
- The complete approach has been designed and developed for the practical application in the SLA domain. It addresses machine-interpretability and automated execution by means of a precise formal semantics and addresses human-consumption wrt psychological adequacy and ergonomically adequacy by means of successful methodologies such as test-driven development from SE, Semantic Web, XML and Object-Oriented technologies and tool support.

Moreover the ContractLog language as well as the RBSLA language fulfil criteria of good language design ([Cod71]) such as minimality (lean set of needed constructs), symmetry (the same language construct always expresses the same semantics regardless of the context it is used) and orthogonality (permits every meaningful combination of a language constructs to be applicable). Hence they can be considered as clear, precise and easy adaptable SLA representation languages which haven been proven to be adequate for the formalization and declarative programming of SLA specifications. The extensible design which allows using arbitrary external Semantic Web ontologies ensures syntactical expressiveness based on a clear formal semantics.

Software Engineering Criteria ContractLogs adopts several SE methodologies and techniques. The hybrid integration approach facilitates interoperability with existing external tools, data sources and object-oriented functionalities. The declarative rule-based representation supports a separation of concerns and aims at easy implementations and modifications. The support for unitized modular rule sets in ContractLog facilitates large SLA specifications to be put together from components which can be developed, compiled and tested separately and managed and communicated within a distributed environment. The approach enforces the "principle of information hiding" and facilitates typical SE related non-functional requirements such as correctness or robustness via meta rules, i.e., rules which monitor the QoS properties of the RBSLM tool.

The RBSLA XML layer supports interchangeability in open environments. Moreover, on the XML level it becomes straightforward to combine the SLA formalizations with standard Web service technologies such as WSDL and Semantic Web technologies. The test logic adopts from SE the test-driven approach for verification and validation of rule bases.

Remarkably, I followed the ideas of agile SE and agile IT project management in the development and management of the RBSLA project. In particular, I applied SCRUM project management techniques and XP techniques such as test-driven development. The RBSLA distribution comes with a set of different test suites for verification, validation and integrity testing of the project implementations reaching from regression tests, functional tests to performance/scalability and inference tests, test coverage analysis and dependency management through Maven2 Java project management as well as Ant and Maven2 project life cycle management including automated deployment tasks for web deployment and SNAPSHOT / distribution releases. The code implementation follows a clear object-oriented approach and applies well-known SE Design Patterns [GHJV94].

Information Systems Design Science Perspective Contrary to numerous empirically oriented studies in IS research of the last years and pure theoretical works in logics, the presented approach follows a constructivist, software engineering-oriented methodology and presents a proof-of-concept implementation. It adopts the Design Science Research approach as described in Hevner et al. [HMPR04] and propose a new design artifact which defines an expressive, declarative, logic-based KR for representation and automation of SLAs based on logic programming techniques and further adequate logical formalisms. The implemented and evaluated IT artifact provides solutions for identified problems in the SLA domain. With my approach I provide new levels of flexibility and automation which are not available within the current procedural techniques and tools. My rule-based approach overcome real-world problems which are of high relevance and importance in SLA representation, management and enforcement such as rapidly changing business environment, slow IT-change cycles and highly-distributed loosely coupled service oriented environments. I base my work on the key developments in research on KR and LP which have been mostly of the more theoretical sort. I draw on these results to develop my prac-

tical ContractLog KR. With the RBSLA markup language which stays close to the emerging Semantic Web rule standard RuleML I address technological real-world business needs concerning the use of my approach in practical Web-based applications. As a result the work presented in this dissertation fulfils both primary demands of Design Science, namely relevance and rigor.

Usability Study During the analysis phase of the RBSLA project, where I have spoken with several IT service providers I have collected users' needs and usability requirements for an SLM tool. Feedback has been also provided from the open-source community to which my RBSLA approach has contributed (Mandarax, Prova, RBSLA), standardization initiatives (RuleML, Reaction RuleML, W3C RIF) and through my work as a technical consultant for industrial partners. However, to quantify the usability of the developed front-end in a more structured way, I have conducted an usability study for the RBSLM tool with 12 participants from the above mentioned groups. The goal was to determine the usability of the tool from a users' perspective and answer questions relating in particular to the two primary goals of the LUCID methodology (see section 6.5), namely "easy-to-use" and "easy-to-understand". The questionnaire had the following categories

1. Related Experience: experience of the user with rule languages in general and RuleML in particular
2. Overall Reactions: on the tool
3. View: understandability and usefulness of the provided menus, wizards and views
4. Learning: learning to operate and apply the tool
5. Capabilities: assess the capabilities
6. Features: assess the features

The complete questionnaire can be found in appendix F. The averaged overall reactions of the participants were:

terrible/wonderful	5.2
frustrating/satisfying	3.7
dull/stimulating	5.8
difficult/easy	4.7
inadequate power / adequate power	5.4
rigid/flexible	6.8

In a nutshell, the RBSLM tool was quantified as useful and adequate. However, the primary LUCID goals are only partially fulfilled. In particular users with less experience in RuleML found it difficult to use and learn the tool. The reason might be, that all users regardless of their related experience where asked to use the tool without any specific task and starting with an empty project. As a result users without experience in RuleML where forced to use features and

fulfill tasks which are normally intended for experienced domain experts and rule programmers. That is, a more fine grained usability study with appropriate preloaded projects and different tasks for different user groups needs to be done as a future step.

SLA-specific Implications As it can be seen from the use case example in section 7.3, the declarative rule based approach allows a compact representation of modularized SLA rules, which would not be possible in standard imperative programming languages. To encapsulate the QoS monitoring and decision making logic of the use case in Java, a large object oriented program with several Java classes and multiple methods to model all subtle nuances would be needed. The entire control flow must be specified, i.e., the exact order and number of steps and decisions needs to be translated in the procedural code. Obviously, this does not always scale and maintenance and management of the knowledge structures becomes increasingly difficult, especially when the SLA logic is likely to change frequently.

This is backed up by the high Capers Jones language levels [Jon96] for declarative LP languages. The Capers Jones level identifies the number of source lines of code which are necessary in a given language to implement a single function point. The higher the language level, the fewer lines of code it takes to implement a function point, and thus presumably it is an indicator of the productivity levels achievable using the language. In contrast to if-then structures in Java, which form a hierarchical tree, logical rules assemble something more like a complex net, with dozens or even thousands of interconnected global rule nodes. Hence, the declarative approach naturally supports reuse of rules by global visibility. While in Java the if-then statements need to be processed from the beginning to the end exactly in the predefined order, in the declarative approach the derivation net can be queried at any node and the inference engine solves the task of finding all specific knowledge out of the general rules which assert general information about a SLA decision making problem.

The declarative approach provides much flexibility when adapting SLA rules to changes, for example, when new discounting policies are added. This ability to dynamically and quickly alter the SLA logic at runtime without any extensions to the generic inference engine is a key advantage of the declarative approach, which otherwise would require reimplementation of the procedural application code resp. database schemas and perhaps larger service execution outages for redeployment. Hence, by representing the SLA monitoring and enforcement logic on a more abstract rule-based level and separating it form the low-level procedural aspects of service runtime environment, much more powerful SLA specifications can be written and maintenance and management of large numbers of complex individualized SLAs becomes much easier. The clear mathematical basis of the selected logical formalisms ensures correctness and traceability of derived results, which is crucial in the SLA domain in order to provide reliable and provable reactions and results, e.g., computed penalties used in accounting. Furthermore, it enables easier validation and verification

of the SLA specifications and therefore ensures consistency (due to sound and complete logical semantics) and integrity (integrity constraints / test cases) as well automated conflict resolution (via defeasible refutation).

8 Conclusion

Logic programming has been a very popular paradigm in the late 1980's and one of the most successful representatives of declarative programming in general. Although, logic programming is based on solid and well-understood theoretical concepts and has been proven to be very useful for rapid prototyping and describing problems on a high abstraction level, its application in commercial software has been limited throughout the past years. However, service level management and electronic contract management appear to be particularly suitable to logic programming. IT service providers need to manage large amounts of SLAs with complex contractual rules. These rules describe various decision and business logic and reach from deeply nested conditional clauses, reactive or even proactive behavior to normative statements and integrity definitions. They are typically not of static nature and have to be continuously adapted to changing needs. Furthermore, the conclusions and results need to be highly reliable and traceable to count even in the legal sense of a contract. There is demand for a declarative knowledge representation language which is

- efficient even for larger SLA specifications
- reliable and traceable even in case of incomplete or contradicting knowledge
- flexible in a way that allows to quickly alter the behavior of the SLA system
- supports declarative programming of arbitrary functionalities and decision procedures

Logic programs have several advantages over imperative languages such as Java, or database solutions. However, given the linguistic richness of real-world IT contracts and the dynamics of the SLA domain, it is clear that any specific pure logical account of SLA rules, such as pure classical definite Horn clauses, are too limited and not able to capture the entire contractual logic with e.g., a normative account. General logic programs need to be extended by multiple knowledge representation concepts and commercial system management tools to allow to formalize the inherent complexity of SLAs and provide the necessary basis for service level management.

8.1 Thesis Summary

In this dissertation, I have implemented a coherent framework of integrated rule-based knowledge representation concepts to axiomatize and automatically enforce large amounts of SLAs based on generic derivation rule engines. In contrast to conventional procedural implementations as in commercial SLM tools the declarative, rule-based (programming) approach in ContractLog provides high levels of extensibility and allows for a greater degree of flexibility in defining contractual agreements. My aim was to select and investigate adequate knowledge representation concepts from the area of AI and KR which satisfy the practical requirements of this domain and combine them in an useful way with conventional techniques from the object-oriented and relational programing domain. A particular interest was the investigation of expressive logic programming techniques and logical formalisms such as defeasible logic, deontic logic, temporal event/action logics, transaction and update logics, description logics. Following a design science research approach I have tackled the prescriptive design problem which is in engineering an assemblage of adequate components which improve SLA representation, monitoring and enforcement. I presented the ContractLog KR as logical core, the RBSLA markup language as serialization, and interchange language and the RBSLM tool as engineering and service dashboard tool, which provide a more effective, flexible and more efficient solution to service level management than existing approaches. To defend and demonstrate the quality of this rule-based solution I have applied established methodologies in SE, KR and LP. A particular focus was the experimental analysis of the comprehensive ContractLog KR in simulations and with test case-based black-box tests and coverage analysis of the derivation and execution chains. The evaluation was in particular with respect to complexity and expressiveness and illustrates that this particular combination of KR concepts allows an efficient and scalable implementation of rule-based service level management tools. Multiple contributions from ContractLog and RBSLA have been made to open source projects (RBSLA, Mandarax, Prova), the new W3C Rule Interchange Format and the Reaction RuleML initiative, giving evidence for the usability of the rule-based approach. Recapitulating, the thematic structure and main line of arguments in this dissertation were:

Chapter 1 described the research problem addressed in this dissertation. It discussed the requirements for a practical IT service level management (SLM) tool and a declarative SLA representation language. The research problem was on the interplay of software engineering (SE) and logical KR formalisms. Based on these requirements a rule-based solution was introduced and design science research was chosen as a set of analytical techniques and perspectives for performing research in IT service level management. The advantages of the proposed rule-based approach were discussed.

Chapter 2 introduces relevant concepts, technologies and standards in IT SLM. A detailed use case example which was extracted from real-world SLAs illustrated the specific needs of the domain. Related work reaching from com-

mercial tools to Semantic Web and logic based approaches was discussed.

Chapter 3 focused on relevant basics in knowledge representation, in particular in the domain of logic programming, Semantic Web technologies and reactive systems. With this background the requirements for a rule based SLA language stated in the introduction were further refined.

Chapter 4 described the ContractLog KR as logical core and introduced the logical formalisms and rule types needed for the adequate representation and automated execution of SLAs / policies. The syntax and semantics were discussed, including descriptive model-theoretic and procedural proof-theoretic definitions and implementations of the formalisms. Examples were used to illustrate their features and usage. The approach relies on meta programming and allows interpretation with different semantics and rule languages (by syntax transformations via RBSLA and XSLT). Moreover, the formalisms and variants are encapsulated as stand-alone scripts which can be imported as modules. This approach allows a flexible management of the well-known trade-off between expressiveness and computational efficiency. Therefore, the ContractLog KR is an useful tool in many domains, in particular in the context of the Semantic Web and Web-based distributed and service-oriented systems. Nevertheless, to optimize ContractLog for the practical requirements of the specific tasks in the SLA/policy domain, some features such as the hybrid typed logic or the labelled meta-data annotated module concept have been implemented as extensions of the logical inference mechanisms. Although these features have no direct formalization in first order logic, I argue that the benefits for a practical rule-based system, which needs to cope with large problem sizes and which needs to efficiently interoperate with existing systems and data sources, prevail. The hybrid design which allows the integration of external vocabulary types, methods and data into rule execution combines the benefits of declarative and procedural (object-oriented) programming and helps to overcome typical problems of declarative programming, e.g., wrt to computational efficiency of certain tasks. This is the basis for a tight integration of the rule-based approach into the existing applications and management tools in the IT service domain, e.g., databases and data warehouses, communication middleware, system and network management tools, ERP systems. The rich capabilities for declarative programming of complex contract decision and behavioral logic ensure epistemologically adequacy, wrt flexibility and extensibility. The ability to integrate external Semantic Web vocabularies, data types and object oriented class hierarchies as type systems provides syntactical expressiveness and facilities rule interchange between domain boundaries.

Chapter 5 addressed requirements of the distributed, Web-based IT service domain, i.e., serialization and interchange of distributed contracts and policies as XML based rule bases. The developed RBSLA language is build on top of the emerging rule markup language (RuleML). Several contributions have been made to RuleML and the W3C RIF standard. The reactive event/action part of the RBSLA language has become the basis for the Reaction RuleML sublanguage of RuleML. The language follows the design principle of RuleML and defines new constructs within separated modules which are added to the RuleML

family as additional layers. The layers are not organized around complexity but add different modelling expressiveness to the RuleML core for the representation of behavioral (re)action and KR event/action logic. The language fulfils typical criteria for good language design such as minimality, symmetry and orthogonality. It is as a full declarative programming and interchange format for contract specifications. The formal semantics is based on the ContractLog KR. Syntactic expressive power as in existing markup proposals such as WS-Agreement or WSLA is provided by the typed logic that allows to reuse arbitrary webized vocabularies (ontologies) as external order-sorted type systems for term typing of rules. Due to this syntactic and semantic heritage the RBSLA approach also qualifies for other phases of the service life cycle such as semantic service discovery by means of SLA offerings written in RBSLA.

Chapter 6 described the superimposed RBSLM prototype. This prototype serves as a proof-of-concept implementation. The RBSLM tool provides support for engineering and managing service contracts including test-suites and template-based repositories to safeguard the authoring process of SLA specifications. It also provides a service dashboard for runtime execution of SLAs, which can be used as tool for further SLM processes. As messaging middleware and distributed object broker the Mule ESB is integrated into the RBSLM implementation.

Chapter 7 evaluated the proposed approach. Since the ContractLog KR is comprehensive including many different formalisms which are used in combination with each other a pure formal analysis of the complete KR is not feasible and misses important practical aspects. Therefore, I focused the theoretical analysis on the core formalisms. They were analyzed in isolation in order to asymptotically quantify the complexity of reasoning with ContractLog. The major analysis was done by experimental simulations measuring the time to derive answers and the scalability for different problem sizes and different settings of expressiveness. The results give an insight into the computational obstacles of the KR framework and allow quantifying the quality and feasibility of the implementation of the selected logical formalisms for SLA representation. The results proved scalability and efficiency for large rule bases in practical real-world settings. Expressiveness and practical applicability for the task of SLA representation are demonstrate by a comprehensive use case. More use case implementations are provided in the ContractLog distribution and by submissions to open source projects such as Mandarax, Prova and the W3C RIF initiative.

Among others, I have shown the following results:

- Automated SLA and policy representation, monitoring and enforcement is a business problem that is relevant and important. The proposed declarative rule-based approach is one possible technology-based solution which qualifies for the frequently changing requirements of contract management in open, distributed, service-oriented environments
- Several requirements for an adequate rule-based representation language and rule-based service level management tools, which come from different

domains, have been identified by a thorough analysis of the problem domain. Among the critical goals are flexibility, extensibility by means of (declarative) programming and scalability/efficiency. Logic programming has been selected as the basis and common "denominator" to solve these core problems.

- A hybrid approach which tightly combines declarative and object-oriented programming is proposed in order to reach an optimal design which provides adequate solutions for the representation, inference and search problem and which fits nicely into the existing techniques, methods and system environments in the SLA domain. The homogenous language design which includes practical language constructs, which do not have a formal semantics based on classical first-order logic, such as Java object qualifications and calls to external functions, operational systems, data sources and terminological descriptions, are vital to produce feasible results in the automated execution, monitoring and enforcement phase and deal with practical problems. In particular, shifting computational intensive tasks such as mathematical computations, aggregations etc. to highly optimized imperative languages such as SQL or Java further increases the efficiency of our ContractLog KR. I have paid special attention to find the right trade-off between efficient practical language constructs without a clear formal semantics and logical formalisms with a declarative model-theoretic semantics.

- Syntax and semantics of RBSLA, ContractLog and its constituent formalisms are presented as part of the system design. A formal characterization of the formalisms wrt model-theory and proof-theory is given. The underlying inference infrastructure, e.g., the ContractLog inference engine and the ECA interpreter, as well as the superimposed RBSLM tool follows a design pattern based architectural style.

- The ContractLog KR and the RBSLA language follow a modular design where the logical formalisms and the language constructs are implemented as separated semi-independent components. That is, the meta programs are implemented as stand-alone LP scripts which can be imported as modules and the RBSLA XML Schema design bundles constructs which belong together to modules which are organized as layers. As a result, the expressive power needed to represent a particular SLA which is in tradeoff to the complexity can be configured on a per-need-basis and integrated under the intellectual control of a human engineer. This decomposition also facilitates extension (with further modules) and verification and validation via test suites provided for each component. An abstraction from the operational level of the inference system, the formal ContractLog KR, the interchangeable RBSLA markup language to the RBSLM user-interface which provides a RBSLA editor and a repository with predefined templates and test cases, provides a mechanism to abstract from the technical details and make the approach more user-friendly.

- The implementation and engineering methodology follows a test-driven ap-

proach stemming from XP with clear iterations between implementation and evaluation. The developed test suites comprise functional-, regression-, performance- and meta test cases for the verification and validation of the inference implementations, semantics and meta programs of the ContractLog KR. The implementation of the RBSLM approach in a business environment has been shown with proof-of-concept scenarios, open-source projects and contributions to standardization initiatives. The quality and efficacy of the approach has been evaluated analytically, experimentally by simulations and by functional testing by means of test cases. The developed test logic also provides means for declarative coverage testing and the semantics allows analysis of the derivation trees.

- The RBSLA approach integrates and interoperates with standard enterprise application architectures and distributed web architectures

In summary, the essential advantages of my approach are:

- Contract rules are separated from the service management application. This allows easier maintenance and management and facilitates contract arrangements which are adaptable to meet changes to service requirements dynamically with the indispensable minimum of service execution disruption at runtime, even possibly permitting coexistence of differentiated contract variants in a distributed environment.
- Rules can be automatically linked (=rule chaining), bundled to modules and executed by rule engines in order to enforce complex business policies and individual contractual agreements.
- Declarative semantics provide highly reliable, traceable and verifiable results, enable shared understanding and interoperability of rules. The rules are easily modifiable, exchangeable and executable between partners allowing definition of rule sets on different levels such as general laws, compliance rules, business rules, event processing rules, normative rules.
- External domain vocabularies can be integrated into rule descriptions providing rich and extensible syntax to the declarative programming power of the RBSLA language
- Test-driven validation and verification methods can be applied to determine the correctness and completeness of contract specifications. Large rule sets can be automatically checked for consistency. Additionally, explanatory reasoning chains provide means for debugging and explanation.
- Contract norms (i.e. rights and obligations can be tracked and enforced on a state-oriented event/action logic and contract/norm violations and exceptions can be (proactively) detected and treated via automated monitoring processes in reactive rules.
- Existing tools and applications, secondary data storages and (business) object implementations might be (re)used by an intelligent combination

of declarative logic based and procedural imperative programming, e.g., reusing EJBs, system and management tools or data warehouses.

Apart from this specific contribution to SLA representation and management, I have made several contributions to the used KR formalisms and LP techniques and described several solutions and extensions to problems. As a result, the ContractLog KR also qualifies as an useful tool in other rule-based application domains, in particular in the area of the Semantic Web.

8.2 Practical Implications and Future Work

The research described in this dissertation was stimulated by the need to design and implement an industrially usable system which fulfils the various real-world requirements of the SLA domain, e.g.,: a flexible and extensible architectural design and implementation, needs of human users and machines, needs of specific SLM applications and interaction with other components. In this dissertation I have restricted myself mainly to the technical design and implementation of the KR formalisms and the inference and KB operations on the platform-specific execution level and the development of the RBSLA language and RBSLM prototype on the platform-independent level. I have done significant work relating to software system development, to move such a large KR design artifact, as the ContractLog KR, into real use.

Even if the setting is not a commercial one, as soon as a system is put to use in practical real-world settings and released to open source communities and standardization initiatives, a difference in the worlds between specialized and small examples and theoretical considerations in research papers and the details of practical and sizable rule-based application systems becomes evident. Although a formal specification including algorithmic specification and computational asymptotic worst-case analysis of the inferences of a KR system is important to precisely clarify the consequences and the properties of the logical formalisms, it is only one part of the work. Because of the complexities and hidden details of real-world problems the presence of a theoretical formal account is no guarantee for the adequacy in a practical system. For instance, a formal semantics published in a theoretical KR paper may have all the properties one may desire, be formally tractable and yet not provide the intended answers. Or it may be too difficult to efficiently implement the inference algorithms or use them in a practical system. Even if a theoretical formalism at the first glance appears to meet the needs, it might not be useful in the interplay with other formalisms or might not provide means to extend it, e.g., in order to overcome flaws and gaps which are subtly hidden in the original logic and become only apparent in the practical use.

My experience gained with implementing and using all the logical formalisms and inference algorithms of the ContractLog KR in the context of a real-world application was that practical implementation and real use is a vital complement to the often too idealistic and delimitated theoretical work on KR theory.

The practical effort can substantially clarify the impact of a logical formalism in real-world settings and can help to detect problems in the formal specification. The resulting concrete implementation clearly reveals its true value for the application domain, it is clearer and more elegant, substantial parts of it have been validated by use and the results are not only theoretically interesting, but are important since they arose out of a real problem. Moreover, the feedback from practice to theory, in particular in the combination of logical formalisms, can lead to new perspectives, new lines of research and new cognitions on the expressiveness of language constructs, the formal semantics and the complexity of inference algorithms. Many of the contributions of the ContractLog KR to the core logical formalisms as described in chapter 4 arose from this fruitful interplay.

However, designing and implementing a formal KR system practically right is an extremely difficult task which is not just a small matter of programming. Significant research is still necessary even after the basic theory is in place and many compromises wrt trade-offs and changes of the original logics are needed, e.g., in adopting practical language constructs without a standard formal semantics (but with a non-standard one). While there is a risk that these concessions to procedural implementation, which allows externalizing tasks to highly optimized built-ins, query languages or object-oriented code, might destroy the benefits of the formal semantics of the overall KR, it turned out to be a crucial means to get around limitations in the logical formalisms which otherwise need to be extended to the expressive power of full first-order logic or even second-order logic and hence would no longer be computationally efficient for the application domain. The reality of a rule-based SLM system which certainly never runs in isolation, but interacts with various external components demands for functionalities such as efficient object-oriented and relational SQL-style retrieval methods that are common in modern information systems.

In fact, in my opinion the hybrid representation approach in ContractLog which exploits declarative rule-based programming in combination with object-oriented procedural programming and support for advanced ontological constructs achieves a much more intuitive and reasonable formal semantics for the core parts of the KR than approaches which strictly follow classical first-order logic and ignore non-classical inference features and practical rule concepts. For instance, typical objects used in computer science are usually composite structures involving several different structures whose elements have a hierarchical inner composition, as opposed to model-theoretic first order structures whose elements have no further composition and are treated uniformly on the basis of truth valuation. By assuming not just a single universe of discourse, but multi-sorted domains in a single structure and by isomorphically abstracting from the inner composition of the elements of the universe, it becomes possible to treat complex objects such as Java objects or ontological individuals as constants in the combined signature of the language(s). As a result, it becomes possible to provide a model-theoretic logical account for transactional KB updates actions, active event-based reactions or external functions (side-effects are neglected). The resulting KR accommodates various formalisms based on

non-standard logics with forms of, e.g., temporal, defeasible and event based reasoning, and combines them with specific features for, e.g., constructing open and closed views and managing and updating the KB in terms of modules including imports of external LP scripts and transactional updates which might be rolled back.

My substantial system development effort made it clear that usability and efficacy of the proposed approach not only depends on the expressiveness, formal account and theoretical clarity of the used logical formalisms, but also strongly depends on how simply the new KR artifact can be integrated into an existing environment, how efficiently it can interact with external components and react to changes in the external environment, how easy it is for the user to learn, apply and understand it and how much support for safely engineering and maintaining a rule base are given. I have addressed this issues on various levels:

(1) At the platform-specific level comprehension and usability of the ContractLog/Prova execution syntax is ensured by the high recognition level due to its close relations to the well-known ISO Prolog standard, the homogeneous representation of different rule types based on a combined syntax and the prescriptive typing approach which allows the natural qualifications of order-sorted distributed KBs and ontological conceptualizations by their URIs and external Java class, object and method/field representations by their class path names. The rich library of built-in and meta functions provided by Prova and ContractLog, e.g., to process mathematical aggregations, process lists or data structures such as date/time values, query relational databases by SQL, process XML data sources or to communicate between distributed knowledge system in multi-agent settings via communication protocols such as Jade or JMS simplifies the integration, management and maintenance of the rule-based systems in distributed Web-based environments. In particular, it becomes straightforward to combine the rule-based approach with existing web service technologies such as XML-based SOAP communication or WSDL service descriptions by calling the respective Java APIs such as Apache Axis during rule execution (e.g., in reactive rules). Moreover, Prova makes it easy to handle errors produced during the generation and reasoning with complex Java objects by providing reasonable built-ins exploiting Java's error-reporting and error-handling capabilities. Although I have done a lot of work to implement useful libraries and functionalities in ContractLog e.g., for processing lists, date/time/interval values or mathematical computations and aggregations, due to the reason of space I could not address them explicitly in this dissertation. Moreover, many of these development and operational execution features are relatively low-level and relate to implementation details, which are not necessarily interesting from a research perspective, although these specific features might be absolutely vital for the acceptance and usability of the approach in practice.

(2) At the platform-independent level I have developed the RBSLA language, which addresses serialization into XML and rule interchange and management in distributed web-based environments. In contrast to other SLA markup or Semantic Web Services proposals, my central goal for RBSLA was to produce a

compact declarative rule-based programming language with an extensible and open design. Minimality, orthogonality and symmetry of the language constructs and a modular layered architecture were the primary design principles. This small and minimal design of the RBSLA core has an important advantage in practical settings, such as maintainability, portability and comprehensibility. On the other hand, the rich programming capabilities of the rule-based approach allows to declaratively implement arbitrary functionalities and the open design of RBSLA allows the integration of external ontological Semantic Web domain models. As a result, the core RBSLA language stays clear and compact, while arbitrary syntactical expressiveness is provided by external domain vocabularies which can be developed and maintained in separation. Moreover, the serialization in XML and the ability to reference external sources by their URIs makes it easy to combine the rule based SLA specification with typical web service descriptions in WSDL.

(3) Finally, on the problem domain and user interface level the RBSLM tool serves as development and runtime environment which provides visual rendering and GUI based management and visual performance and execution monitoring by means of adaptable service dashboard views. My intention was to support different roles which are involved during the life cycle of SLAs and policies such as domain experts and rule engineers which design and provide predefined rule templates, interface functions, test suites or domain conceptualizations in a repository and business practitioners and quality managers which reuse the predefined templates and models to build their SLA specifications and maintain and monitor the contracts during runtime by graphical dashboard views which can be adapted to the specific needs of the user, e.g., the customer or the system administrator. From a pragmatic, practical point of view this user-oriented tool support is very important to improve usability and it is clear that users will not adopt and stick to the rule-based approach if the abstraction of the user interface from the logical formalisms and the programming interfaces is not understandable and easy to use for the different (end) users.

In sum, when designing a KR artifact paying only attention to theoretical issues such as computational complexity, decidability or compactness is no guarantor for the success and adequacy of the approach and practical factors ranging from implementation tradeoffs and system engineering features like testing, modular management and maintenance or interchangeability and interoperability to concern of different users including learnability and efficiently usability of the KR language(s) can not be ignored. In my work I have considered and contributed to both sides.

In this dissertation I have mainly concentrated on the monitoring and enforcement phase of service contracts, since obviously this phase has the most elaborated requirements on the knowledge representation, whereas other phases such as the discovery or analysis phase only need smaller subsets of the full expressive power. Future work in the RBSLA project might consider other phases. SLA offerings specified in RBSLA in addition to the technical WSDL description of a web service need no extra language constructs and the formal

semantics provided by the ContractLog KR and the description logic/Semantic Web domain vocabularies qualifies my approach directly as a highly expressive Semantic Web Service and policy language.

Another dimension of future work is the further integration of the rule-based SLA/SLM component into ITSM frameworks. Basically an integration is possible on three different levels: operationally, tactically and strategically. As discussed the technical interfaces to communicate and integrate external functionalities as well as to derive and query results from the rule system or actively trigger reactions in external systems exists and are already demonstrated and used in the use cases and examples provided in the RBSLA project. This forms the technical basis to close the loop between the different components of a service level management solution, which includes e.g.,

- complete SLA life cycle management: all elements in the life cycle from definition, configuration, enforcement and assessment are connected, integrated and working together.
- coordinated approach: all involved management tools work together in a coordinated manner.
- integration: flexibility and scalability to continually grow and adapt to changes in the dynamic service-delivery and business policies
- decision on different levels: individual infrastructure elements are linked to the business perspective enabling e.g., business impact analysis or aggregated balanced score cards.

8.3 Closing Remarks

In this dissertation I tackled the manifold challenges in IT SLM and SLA management which are posed by the highly complex, dynamic, scalable and open enterprise service architectures and environments such as the Semantic Web. The essence of my work is on the practical combination and on the interplay of successful methodologies, techniques and technologies from different fields. I have formalized, implemented and evaluated the proposed concepts within open-source projects (e.g. Prova, Mandarax, RBSLA), research collaborations (e.g. Rewerse R2ML) and industrial use case studies. Multiple contributions from my RBSLA project have been made to the standardization initiatives RuleML, the new W3C Rule Interchange Format and Reaction RuleML which I co-chair. Beside the further standardization of Reaction RuleML with ContractLog/Prova as a reference implementation, further investigations will address the combination of RBSLA/RBSLM with the macro-level of IT service management, Business Activity Management and Business Process Management. Here, the developed enterprise integration patterns for complex event processing and rule-based workflows in the RBSLM prototype serve as a technical foundation.

A Glossary

AA	agent architecture
ACL	agent communication language
ACTL	action computation tree logic
AI	artificial intelligence
ASP	application service providing
ASS	answer set semantics
AST	abstract syntax tree
BAM	business activity management
BPM	business performance management
BPEL	business process execution language
BSI	british standards institutes
CBE	common base event
CEP	complex event processing
CTDO	contrary to duty obligations
CLP	constraint logic programming
CM	contract manager
COMP	completion semantics
DC	Dublin Core
DCA	domain closure assumption
DefL	defeasible logic
DL	description logic
DLP	description logic programs
DNL	deontic logic
EAI	enterprize application integration
EC	event calculus
ECA	event condition action
EDA	event-driven architecture
EDB	extensional database
EIS	event instance sequence
ELP	extended LP
ESB	enterprise service bus
FOL	first-order logic
GCLP	generalized courteous logic programs
GUI	graphical user interface
HIS	heterogenous information systems
IC	integrity constraint
IDB	intensional database
ILP	inductive logic programming
IT	information technology
ITIL	information technology infrastructure library
ITIM	IT infrastructure management
ITSM	IT service management
JMS	java messaging service
KB	knowledge base
KBS	knowledge base system
KR	knowledge representation

LFP	least fix point
LP	logic program
LLP	labelled logic program
LTL	labelled transition logic
LUCID	Logical User-Centered Interactive Design
MAS	multi-agent system
MGU	most general unifier
MVC	model view controller
MVI	maximum validity interval
NAF	negation as failure
OASIS	organization for the advancement of structured information standards
OCL	object constraint language
OID	object identifier
OLA	operation level agreement
OLP	ordered logic program
OO	object oriented
OWA	open world assumption
OWL	web ontology language
PIM	platform independent model
PLP	prioritized logic program
PSM	platform specific model
QoS	quality of service
RBSLA	rule based service level agreement
RBSLM	rule based service level management
RDF	resource description framework
RDFS	RDF schema
RIF	rule interchange format
RTE	real-time enterprise
RuleML	rule markup language
SDL	standard deontic logic
SE	software engineering
SEM	semantics
SLA	service level agreement
SLE	linear resolution with selection function for extended WFS
SLI	service level indicators
SLM	service level management
SLO	service level objectives (SLOs)
SMI	structure of management information
SCA	service component architecture
SD	service dashboard
SEDA	staged event driven architecture
SOA	service-oriented architecture
SOAP	simple object access protocol
SOC	service oriented computing (SOC)
STABLE	stable model semantics
SWS	semantic web services (SWS)
TAL	temporal action logic

TC	test case
TR	transaction logics
TS	test suite
UC	underpinning contract
UI	user interface
UNA	unique name assumption
USDL	universal service-semantics description language
V&V	validation and verification
V&V&I	validation, verification and integrity testing
WFS	well-founded semantics
WFSX	WFS for extended LPs
WSDL	web service description language
XP	extreme programming

B Variables and Functions

A	atom
Ac	action
$arity$	arity function
As	assertion
$\overline{As}$	set of assertions
B	body of rule
B_P	Herbrand base
$\overline{B}(P)$	extended Herbrand base
c	constant
$\overline{c}$	finite or infinite sequence of constant symbols
C	constraint
$\overline{C}$	set of constraints or scope
cl	clause
Cl	closure
coh	coherence operator
Co	condition
D	domain description
D^{EC}	domain description
E	expression
Eq	set of equations
Ex	extension
$\overline{E}$	finite set of expressions
Ev	event
$\overline{Ev}$	set of events
f	function
$f_{\cong}$	isomorphic function
F	formula
$\overline{F}$	finite sequence of function symbols
Fa	fact
$\overline{Fa}$	set of facts
Fl	fluent
$\overline{Fl}$	set of fluents
G	goal
$ground(P)$	Herbrand instantiation / grounding
H	head of rule
I	Interpretation
I^{Herb}	Herbrand Interpretation
IC	integrity constraint
$\overline{IC}$	set of integrity constraints
KB	knowledge base
KB_i	knowledge base state
$\overline{KB}$	combined knowledge base
l	label
L	literal

L^t	tagged literal
L_b	body literal
L_h	head literal
lfp	least fix point
lgg	least general generalization
m	meta data annotation label
M	(classical) model
$\overline{M}$	set of all (classical) models
M^{Herb}	Herbrand model
M^{Stable}	Stable model
$\overline{M}^{Herb}(P)$	set of all Herbrand models of a program P
M_P^{Herb}	minimal/least Herbrand model
$\|M\|$	domain or universe of an interpretation
N	deontic norm
O	obligation
p	predicate symbole
$\overline{P}$	finite sequence of predicate symbols
P	logic program
P^{part}	partial program / module
P^M	Gelfond-Lifschitz transformation / reduct
Po	post condition
Q	query
r	rule
$\overline{R}$	set of rules
$\overline{R}_s$	set of strict rules
$\overline{R}_d$	set of defeasible rules
$\overline{R}_{sd}$	set of defeasible and strict rules
S	signature
$\overline{S}$	combined signature
SR	set of sort restrictions
t	term
$\bar{t}$	finite sequence of terms
T	sort / type
$\overline{T}$	finite sequence of sorts / types respectively type alphabet
TC	test case
Te	test
$\overline{Te}$	set of tests
Ti	time point
$\overline{Ti}$	set of time points
TS	type system
U_{oid}	update
U_{oid}^{pos}	positive update
U_{oid}^{neg}	negative update
U_P	Herbrand universe
$\overline{U}(P)$	combined universe
W_P^*	well-founded (partial) model

X	variable
$\overline{X}$	finite set of variables
θ	substitution
SEM	semantics
σ	variable/term assignment
Σ	alphabet/language
Σ^{DL}	DL language
Σ^{EC}	EC language
Φ	knowledge base or theory
Ψ	DL knowledge base
π	execution path

C Rule Benchmarks

Scalable Derivation Rules Benchmark Tests (adapted from [ABMR00])

Rule Chaining (chains): In chains(n,p,v) $p * a_0(x_1, .., x_v)$ *facts are at the end of a chain of* n *rules* $a_i(X_1, .., X_v) : -a_{i-1}(X_1, .., X_v)$ *with* v *variables in each atom, where* $p > 0$, $v >= 0$ *and* $n > 0$. *A query* $a_n(X_1, .., X_v)$? *will use all* n *rules and all* p *facts. If* $v = 0$ *the test is propositional, i.e. it has no variables, if* $v > 0$, *i.e. it has* v *variables, the test is a (function-free) datalog program. Further variants with functions and negations (default / explicit) might be defined.*

$$chains(n,p,v) = \left\{ \begin{array}{c} a_n(X_1, .., X_v) : -a_{n-1}(X_1, .., X_v) \\ ... \\ a_1(X_1, .., X_v) : -a_1(X_1, .., X_v) \\ \\ a_0(x_1, .., x_v)....a_0(x_1, .., x_v).\%facts \end{array} \right.$$

Loop (loop): loop(n,p,v) consists of n *rules* $a_{(i+1)modn}(X_1, .., X_v) : -a_i(X_1, .., X_v).$ *with* v *variables and* p *facts.*

$$loop(n,p,v) = \left\{ \begin{array}{c} a_n(X_1, .., X_v) : -a_{n-1}(X_1, .., X_v) \\ ... \\ a_1(X_1, .., X_v) : -a_0(X_1, .., X_v) \\ \\ a_0(x_1, .., x_v) : -a_n(X_1, .., X_v).\%loop \\ \\ a_0(x_1, .., x_v).\%facts \end{array} \right.$$

Recursion (dag): In Directed Acyclic Graph dag(n,k,v) $a_0(X_1, .., X_v)$ *with* v *variables* X_v *is at the root of a* k*-branching graph of rules of depth* n *in which every literal occurs* k *times and has* k *facts at the ground. A query* $a_0(X_1, .., X_v)$? *will (recursively) use every rule and all facts.*

$$dag(n,k,v) = \left\{ \begin{array}{c} a_0(X_1, .., X_v) : -a_1(X_1, .., X_v), a_2(X_1, .., X_v), ..., a_k(X_1, .., X_v). \\ a_1(X_1, .., X_v) : -a_2(X_1, .., X_v), a_3(X_1, .., X_v), ..., a_{k+1}(X_1, .., X_v). \\ ... \\ a_{nk}(X_1, .., X_v) : -a_{nk+1}(X_1, .., X_v), a_{nk+2}(X_1, .., X_v), ..., a_{nk+k}(X_1, .., X_v). \\ a_{nk+1}(x_1, .., x_v).a_{nk+2}(x_1, .., x_v)....a_{nk+k}(x_1, .., x_v). \end{array} \right.$$

Tree (tree): In tree(n,k,v) $a_0(X_1, .., X_v)$ *is at the root of a* k*-branching tree of depth* n *in which every literal occurs once and which has* v *variables.* $tree(n, k, v) = rule(a_0, n, k, v)$ *where* $v >= 0$ *is the number of variables,* a_0 *is the start literal,* $n > 0$ *and* $k > 0$ *and* $a_1, a_2, ..., a_k$ *are new unique literals in each step* $n-1$ *until* $n = 0$. *A query* $a_0(X_1, .., X_v)$? *will use every rule and all facts.*

$$rules(a,n,k,v) = \begin{cases} a(X_1,..,X_v) :- a_1(X1,..,Xv), a_2(X_1,..,X_v),..,a_k(X_1,..,X_v). \\ rules(a_1,n-1,k,v) \\ rules(a2,n-1,k,v) \\ ... \\ rules(ak,n-1,k,v) \end{cases}$$

$$and : rules(a,0,k) = a(X_1,..,X_v).\%facts$$

Table C.1: Size of Derivation Rules Benchmarks)

Theory	Facts	Rules	Size (Literals)
chains(n,p,v)			
strict	p	n	$2n+p$
defeasible	$p+2n$	$3n$	$11n+p$
loop(n,p,v)			
strict	p	$n+1$	$2(n+1)+p$
defeasible	$p+2(n+1)$	$3(n+1)$	$11(n+1)+p$
dag(n,k,v)			
strict	k	$nk+1$	nk^2+nk+k
defeasible	$k+2n$	$3n+3$	$3nk^2+8nk+k$
tree(n,k,v)			
strict	k^n	$\sum_{i=0}^{n-1}$	$(k+1)\sum_{i=0}^{n-1}+k^n$
defeasible	k^n	$3\sum_{i=0}^{n-1}$	$(3k+8)\sum_{i=0}^{n-1}+k^n$

Scalable Event Calculus Tests

HoldsAt: $ec_{holdsAt}(n)$ *there is a pair of initates/terminates rules initiating / terminating a property (fluent)* f *by means of* n *alternating events* e_1, e_2 *(happens facts). A query* $holdsAt(f,t_n)$*? will use the two EC rules and all happens facts.*

$$ec_{holdsAt}(n) = holdsAt(f,t_n) = \begin{cases} initiates(e_1,f,T). \\ terminates(e_2,f,T). \\ happens(e_1,t_0). \\ ... \\ happens(e_2,t_n). \end{cases}$$

ValueAt: $ec_{valueAt}(n)$ *there is a pair of initates/terminates rules initiating / terminating a property (fluent)* f *by means of* n *alternating events* e_1, e_2 *(happens facts). The event* e_1 *starts a parameter* p *which computes the validity interval (trajectory counter) of the fluent* f *until* e_2 *occurs. A query* $valueAt(f,t_n)$*? will use the two EC rules, all happens facts and the trajectory to compute the validity interval* p *of the fluent* f*.*

$$ec_{holdsAt}(n) = holdsAt(f,t_n) = \begin{cases} trajectory(f,T_1,p,T_2,(T_2-T_1)). \\ initiates(e_1,f,T). \\ terminates(e_2,f,T). \\ happens(e_1,t_0). \\ ... \\ happens(e_2,t_n). \end{cases}$$

Scalable ECA Tests

Plain ECA: $eca_{plain}(n)$ *consists of n ECA rules* $eca(\_,\_,\_,\_,\_,\_)$*, where the ECA parts have no functionality, i.e. are empty and always true, which leads to a full execution of the ECA rule. The rules are processed sequentially by the ECA processor. A variant* $eca^*_{plain}(n)$ *uses multi-threading and processes the rules in parallel. The ECA daemon first queries the KB for new ECA rules via a query* $eca(T,E,C,A,P,El)?$ *and populates the active KB with them. Then it evaluates the ECA rule one after another. Accordingly, update time for querying the KB and collecting new/updated ECA rules and processing time for executing all ECA rules must be distinguished.*

$$eca_{plain}(n) = \begin{cases} eca_1 : eca(\_,\_,\_,\_,\_,\_). \\ eca_2 : eca(\_,\_,\_,\_,\_,\_). \\ \quad\quad ... \\ eca_n : eca(\_,\_,\_,\_,\_,\_). \end{cases}$$

LP ECA: $eca_{LP}(n,v)$ *consists of n ECA rules (TECAP rules)* $eca(t(X_1,..,X_v)$ $,e(X_1,..,X_v),\ c(X_1,..,X_v),\ a(X_1,..,X_v),\ p(X_1,..,X_v))$*, where the ECA parts are complex terms with v variables. Each part is evaluated with a query on a LP fact* $t(x_1,..,x_v).$*,* $e(x_1,..,x_v),...,p(x_1,..,x_v)$*. The test distinguishes update time and processing time. A variant* $eca^*_{LP}(n,v)$ *uses multi-threading and processes the rules in parallel.*

$$eca_{LP}(n,v) = \begin{cases} eca_1 : eca(t(X_1,..,X_v),e(X_1,..,X_v),c(X_1,..,X_v),a(X_1,..,X_v),p(X_1,..,X_v)). \\ eca_n : eca(t(X_1,..,X_v),e(X_1,..,X_v),c(X_1,..,X_v),a(X_1,..,X_v),p(X_1,..,X_v)). \end{cases}$$
$t(x_1,..,x_v).\ e(x_1,..,x_v).\ c(x_1,..,x_v).\ a(x_1,..,x_v).\ p(x_1,..,x_v).$

Procedural ECA: $eca_{Procedural}(n)$ *consists of n ECA rules*

```
eca(
    rbsla.utils.Test.test(),
    rbsla.utils.Test.test(),
    rbsla.utils.Test.test(),
    rbsla.utils.Test.test(),
    rbsla.utils.Test.test(),
    rbsla.utils.Test.test( ))
```

where each ECA part is a procedural attachment on a boolean valued test of the class rbsla.utils.Test which returns true, i.e. the complete ECA rule is executed. The test distinguishes update time and processing time. A variant $eca^*_{Procedural}(n,v)$ *uses multi-threading and processes the rules in parallel.*

Table C.2: Size of Event Calculus Benchmarks)

Theory	Facts	Rules	Size (Literals)
$ec_{holdsAt}(n)$	$n+2$	0	$n+2$
$ec_{valueAt}(n)$	$n+3$	0	$n+3$

CA Rules Benchmark Programs

Table C.3: Size of ECA Benchmarks)

Theory	Facts	Rules	Size (Literals)
$eca_{plain}(n)$	n	0	n
$eca_{LP}(n, v)$	$n + 5$	0	$n + 5$
$eca_{Procedural}(n)$	n	0	n

Table C.4: Benchmark Test Suite)

Name	Description	Size (Rules)
Manners	Finds a seating arrangement for dinner guests	8
Waltz	Waltz line labelling for simple scenes	33
ARP	Route planner for a robotic air vehicle	118

D Appendix RuleML

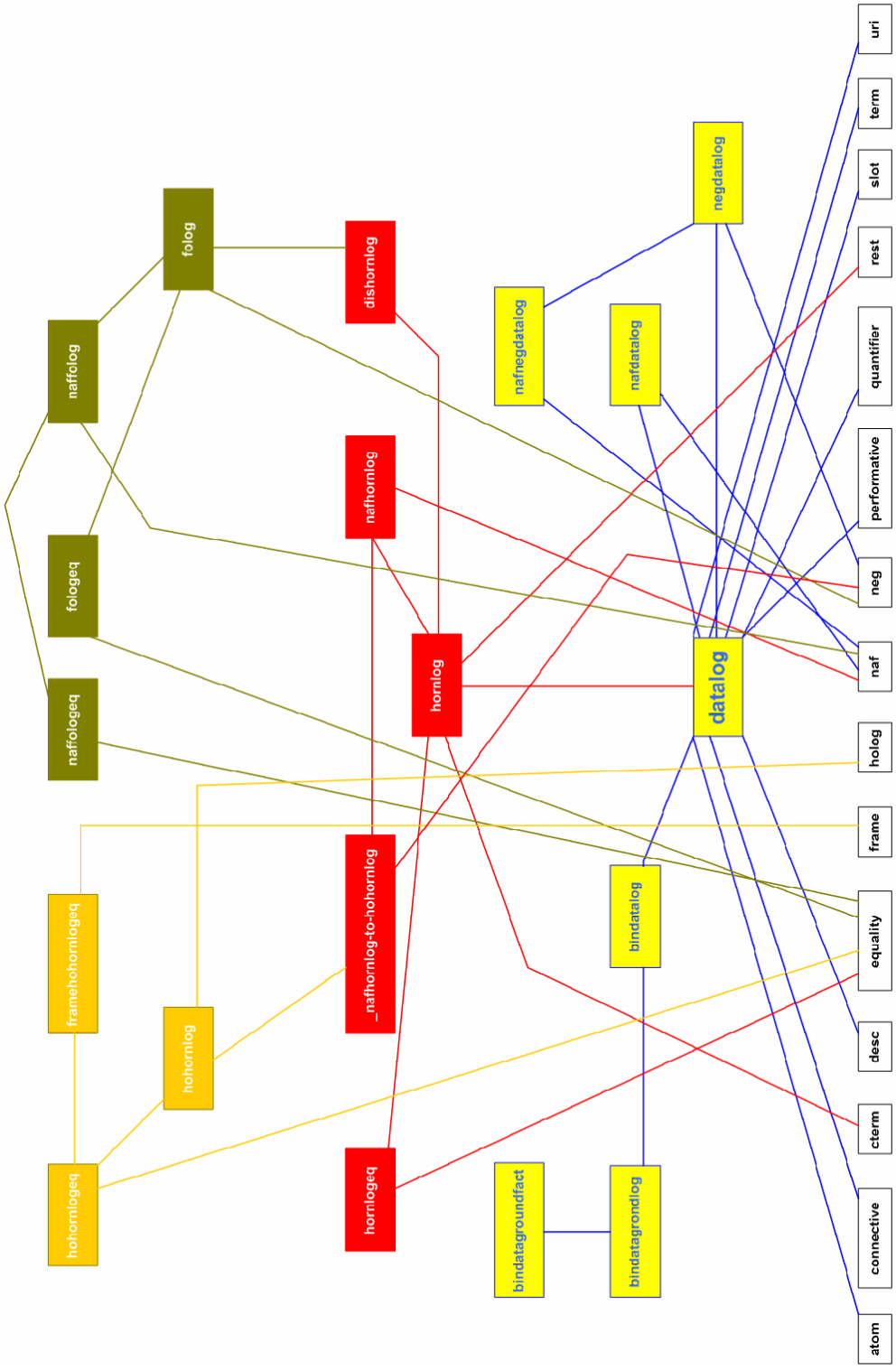

Figure D.1: RuleML Family

E Appendix Categorization of SLA Metrics

No	Description	Object	Unit
1	Availability	Hardware	Time hour, percent
2	Maximum down-time	Hardware	Hours or percent
3	Failure frequency	Hardware	Number
4	Response time	Hardware	Duration in minutes/seconds
5	Periods of operation	Hardware	Time
6	Service times	Hardware	Time
7	Accessibility in case of problems	Hardware	Yes/no
8	Backup	Hardware	Time
9	Processor time	Hardware	Seconds
10	Instructions per second	Hardware	Number per second
11	Number of workstations	Hardware	Number

Figure E.1: Hardware Performance Metrics

No	Description	Object	Unit
1	Service times	Software	Time
2	Response times	Software	Minutes
3	Availability	Software	Time
4	Solution times	Software	Minutes
5	Number of licences	Software	Number

Figure E.2: Software Performance Metrics

No	Description	Object	Unit
1	WAN period of operation	Network	Time
2	WAN Service times	Network	Time
3	LAN period of operation	Network	Time
4	LAN Service times	Network	Time
5	Solution times	Network	Minutes
6	Availability WAN	Network	Percent
7	Availability LAN	Network	Percent
8	Access Internet across Firewall	Network	Yes/no
9	Access RAS	Network	Yes/no
10	Latency times	Network	Ms

Figure E.3: Network Performance Metrics

No	Description	Object	Unit
1	Availability	Storage	Time hour, percent
2	Maximum down-time	Storage	Hours or percent
3	Failure frequency	Storage	Number
4	Response time	Storage	Duration in minutes/seconds
5	Periods of operation	Storage	Time
6	Service times	Storage	Time
7	Accessibility in the case of problem	Storage	Yes/no
8	Backup	Storage	Time
9	Bytes per second	Storage	Number per second
10	Memory size	Storage	Number in bytes

Figure E.4: Storage Performance Metrics

No	Description	Object	Unit
1	Self solution rate (not with 2nd level support)	User Help Desk	Percent
2	Service times	User Help Desk	Time
3	Availability	User Help Desk	Time
4	Failure forwarding degree	User Help Desk	Percent
5	Failure categorization degree	User Help Desk	Percent
6	Availability with phone	User Help Desk	Hours
7	Availability with Email	User Help Desk	Hours
8	Response time	User Help Desk	Hours, minutes
9	Language variety	User Help Desk	Number of languages

Figure E.5: Help Desk Performance Metrics

ITIL Process	Service Metrics
Service Desk	Customer satisfaction with the Help Desk
Incident Management	Time between loss and replacement
Problem Management	Number of repeated disturbances
Configuration Management	Time between adding configuration items to Configuration Management Data Base (CMDB)
Change Management	Number of untreated changes
Service-Level Management	Number of SLAs
Release Management	Time between releases
Capacity Management	Completion of the capacity plan at a fixed time
Availability Management	Completion of the availability plan at a fixed time
IT-Service-Continuity-Management	Completion of the contingency plan at a fixed time
Financial Management	Cost overview to the deadline

Figure E.6: Categorization according to ITIL Process Metrics

Description	Examples
Measurable	Availabilities, Response Times
Limited Measurability	Customer Satisfaction
Not Measurable	Qualification of employees

Figure E.7: Categorization according to Measurability

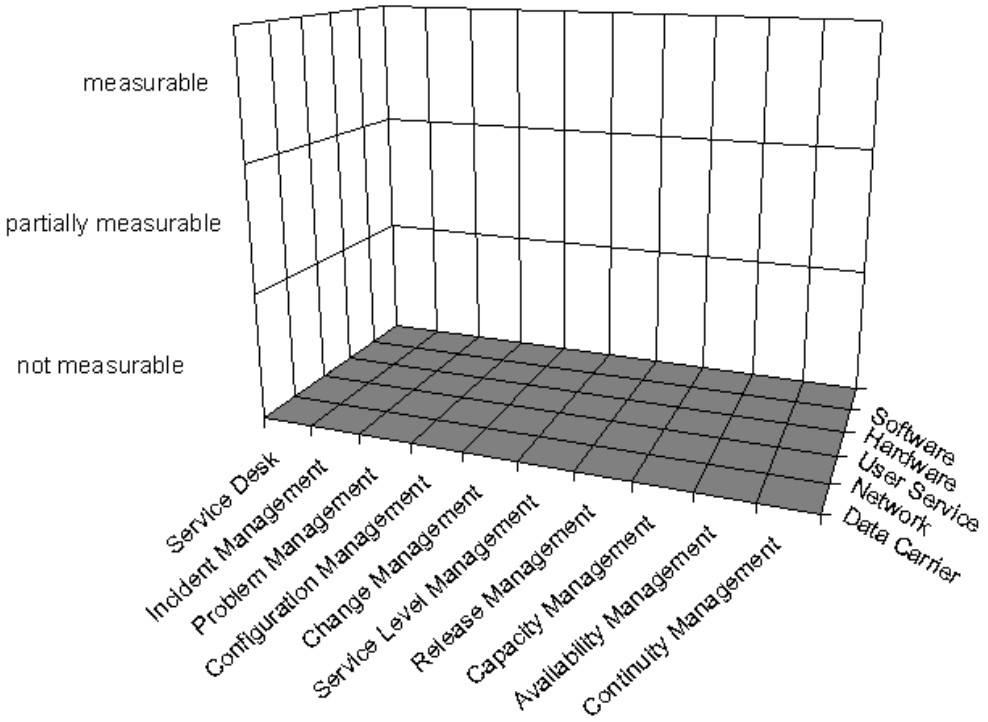

Figure E.8: Three-dimensional categorization scheme for SLA metrics

F RBSLM Usability Testing Questionnaire

RBSLM Usability Testing Questionnaire

I. Personal Information (Optional) Name:

Sex: Female Male

Age:

II. Related Experience

	no experience						extensive use
SLA / SLM	1	2	3	4	5	6	7
XML	1	2	3	4	5	6	7
Rule Languages and/or Rule Engines	1	2	3	4	5	6	7
RuleML	1	2	3	4	5	6	7

III. Overall Reactions:

terrible wonderful
1 2 3 4 5 6 7

frustrating satisfying
1 2 3 4 5 6 7

dull stimulating
1 2 3 4 5 6 7

difficult easy
1 2 3 4 5 6 7

```
inadequate power    adequate power
 1  2  3  4  5  6  7

rigid   flexible
 1  2  3  4  5  6  7


IV. View

1.) Characters on the computer screen

    hard to read    easy to read
     1  2  3  4  5  ; 6  7

   1.1) Image of the characters

        fuzzy   sharp
     1  2  3  4  5  ; 6  7

   1.2) Character font

   barely legible  very legible
     1  2  3  4  5  ; 6  7

2.)Flexible/Adaptable views were helpful

        never   always
     1  2  3  4  5  ; 6  7

   2.1) Amount of information displayed on screen

     inadequate  adequate
     1  2  3  4  5  ; 6  7

   2.2) Arrangement of information that can be displayed

      illogical   logical
     1  2  3  4  5  ; 6  7

   2.3) Size of Input and output text areas.

    too small   large enough
     1  2  3  4  5  ; 6  7

3.) Menus and Wizards where easy to use and understand

        difficult   easy
```

```
     1  2  3  4  5  ; 6  7


V. Learning

1.) Learning to operate the system

        difficult   easy
     1  2  3  4  5  ; 6  7

2.) Exploration of features by trial and error

    discouraging    encouraging
     1  2  3  4  5  ; 6  7

   2.1) Exploration of features

        risky   safe
     1  2  3  4  5  ; 6  7

   2.2) Discovering new features

        difficult   easy
     1  2  3  4  5  ; 6  7

3.) Tasks can be performed in a straightforward manner

       never   always
     1  2  3  4  5  ; 6  7

   3.1) Number of steps per task

     too many    just right
     1  2  3  4  5  ; 6  7

   3.2) Steps to complete a task follow a logical sequence

        never   always
     1  2  3  4  5  ; 6  7

   3.3) Needed sequence of steps for completion

        unclear clear
     1  2  3  4  5  ; 6  7


VI. Capabilities
```

```
1.) Tool performance

     too slow    too fast
     1  2  3  4  5  ; 6  7

   1.1) Response time for most operations

    too slow    fast enough
     1  2  3  4  5  ; 6  7

   1.2) Answers / results are displayed

    too slow    fast enough
     1  2  3  4  5  ; 6  7

2.) The Tool is reliable

       never   always
     1  2  3  4  5  ; 6  7

   2.1) Systems failures occur

        frequently  seldom
     1  2  3  4  5  ; 6  7

   2.2) System warns you about potential problems

        never   always
     1  2  3  4  5  ; 6  7

3.) Correcting your mistakes

        difficult   easy
     1  2  3  4  5  ; 6  7

   3.1) Correcting errors

        complex simple
     1  2  3  4  5  ; 6  7

   3.2) Ability to undo operations

        inadequate  adequate
     1  2  3  4  5  ; 6  7

4.) Ease of operation depends on your level of experience
```

```
        never   always
     1  2  3  4  5  ; 6  7

   4.1) You can accomplish tasks knowing very little

    with difficulty easily
     1  2  3  4  5  ; 6  7

   4.2) You can use features

    with difficulty easily
     1  2  3  4  5  ; 6  7


VII. Features

1.) Project Management Features

    hard to use easy to use
     1  2  3  4  5  ; 6  7

    hard to understand  easy to understand
     1  2  3  4  5  ; 6  7

2.) Contract Manager Query Tab

    hard to use easy to use
     1  2  3  4  5  ; 6  7

    hard to understand  easy to understand
     1  2  3  4  5  ; 6  7

3.) Contract Manager Test Suite Tab

    hard to use easy to use
     1  2  3  4  5  ; 6  7

    hard to understand  easy to understand
     1  2  3  4  5  ; 6  7

 4.) Knowledge Base Explorer

    hard to use easy to use
     1  2  3  4  5  ; 6  7

    hard to understand  easy to understand
```

```
     1  2  3  4  5  ; 6  7

5.) View Creator

    hard to use easy to use
     1  2  3  4  5  ; 6  7

    hard to understand  easy to understand
     1  2  3  4  5  ; 6  7

6. )Integrity Test and Defeasible Converter
        hard to use easy to use
         1  2  3  4  5  ; 6  7

    hard to understand  easy to understand
         1  2  3  4  5  ; 6  7

VIII. Other comments
```

G Bibliography

[AADW05] A. Analyti, G. Antoniou, C. Damasio, and G. Wagner. Stable model theory for extended rdf ontologies. In *Proceedings of Int. Semantic Web Conf.*, 2005.

[AB94] K. Apt and H. Blair. Logic programming and negation: A survey. *J. of Logic Programming*, 19(20):9–71, 1994.

[AB01] A. S. Abrahams and J. M. Bacon. Representing and enforcing e-commerce contracts using occurrences. In *4th International Conference on Electronic Commerce Research (ICECR4)*, Dallas Texas, 2001.

[ABGM00] G. Antoniou, D. Billington, G. Governatori, and M. J. Maher. A flexible framework for defeasible logic. In *AAAI-2000*, Menlo Park, CA, 2000. MIT Press.

[ABGM01] G. Antoniou, D. Billington, G. Governatori, and M.J. Maher. Representation results for defeasible logic. *ACM Transactions on Computational Logic*, 2:255–287, 2001.

[ABLP02] J.J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA'02*, 2002.

[ABM+02] S. Abiteboul, O. Benjelloun, I. Manolsecu, T. Milo, and R. Weber. Active xml: Peer-to-peer data and web services integration. In *VLDB*, 2002.

[ABMR00] G. Antoniou, D. Billington, M.J. Maher, and A. Rock. Efficient defeasible reasoning systems. In *Australian Workshop on Computational Logic*, 2000.

[ABW88] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases*, pages 89–148. Morgan Kaufmann, 1988.

[ACD+05] A. Andrieux, C. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Webservices agreement specification (ws-agreement), http://www.gridforum.org/, accessed nov. 2005, 2005.

[ADP94] J. Alferes, C. Damasio, and L. M. Pereira. Slx: a top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *International Logic Programming Symp*, pages 424–439, 1994.

[ADP95] J. J. Alferes, C. Damasio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *J. of Automated Reasoning*, 14(1):93–147, 1995.

[AE82] K. Apt and M. H. Emden. Contributions to the theory of logic programming. *J. of ACM*, 29(3):841–862, 1982.

[AE93] K. Apt and S. Etalle. On the unification free prolog programs. In *Mathematical Foundations of Computer Science*, pages 1–19. 1993.

[AH93] A. Abecker and P. Hanschke. Taxlog: A flexible architecture for logic programming with structured types and constraints. In *Constraint Processing: Proceedings of the International Workshop at CSAM'93*, St. Petersburg, 1993.

[AHPV98] G. Antoniou, F.v. Harmelen, R. Plant, and J. Vanthienen. Verification and validation of knowledge-based systems - report on two 1997 events. *AI Magazine*, 19(3):123–126, 1998.

[AJ98] G. Antoniou and O. Jack. Testing production system programs. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, 1998.

[AKP91] H. Ait-Kaci and A. Podelski. Towards the meaning of life. In *Int. Symposium on Programming Language Implementation and Logic Programming*, pages 255–274. Springer, 1991.

[Alf93] J. Alferes. *Semantics of Logic Programs with Explict Negation*. PhD Thesis. Univ. Nova de Lisboa, 1993.

[AM02] G. Antoniou and M.J. Maher. Embedding defeasible logic into logic programs. In *ICLP 2002*, 2002.

[AMB00] G. Antoniou, M. J. Maher, and D. Billington. Defeasible logic versus logic programming without negation as failure. *Journal of Logic Programming*, 41(1):45–57, 2000.

[Ant97] G. Antoniou. Verification and correctness issues for nonmonotonic knowledge bases. *International Journal of Intelligent Systems*, 12(10):725–738, 1997.

[Ant02] G. Antoniou. Relating defeasible logic to extended logic programs. In *SETN '02: Proceedings of the Second Hellenic Conference on AI*, pages 54–64. Springer, 2002.

[AO04] A. Adi and E. Opher. Amit - the situation manager. *VLDB Journal*, 13(2), 2004.

[Apt90] K. Apt. Logic programming. In J.v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume volume B, chapter 10, pages 493–574. Elsevier Science, 1990.

[Aqu87] L. Aquivst. *Introduction to Deontic Logic and Theory of Normative Systems*. Bibliopolis, Napoli, 1987.

[AS96] J. Andrew and M. Sergot. A formal characterisation of institutionalised power. *Journal of the IGPL*, 4(3):427–443, 1996.

[AV91] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Science*, 43:62–124, 1991.

[AWH94] A. Aiken, J. Widom, and J.M. Hellerstein. Behaviour of database production rules: termination, confluence and observable determinism. In *Int. Conf. on Management of Data*, pages 59–68. ACM, 1994.

[Bar88] H. P. Barendregt. Introduction to lambda calculus. In *Proceedings of Workshop on Implementation of Functional Languages*, Goeteborg, 1988. Programming Methodology Group.

[BB89] C. Beierle and S. Boettcher. Protos-l: Towards a knowledge base programming language. IWBS Report 89, 1989.

[BBCC02] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active xquery. In *Int. Conf. on Data Engineering (ICDE)*, pages 403–418, 2002.

[BCP01] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to xml repositories using active rules. In *WWW 2001*, 2001.

[BCRS97] J. Baley, L. Crnogorac, K. Ramamoharanao, and H. Sondergaard. Abstract interpretation of active rules and its use in termination analysis. In *Int. Conf. on Database Theory*, pages 188–202, 1997.

[BCV03] R. Baldoni, M. Contenti, and A. Virgillito. The evolution of publish/subscribe communication systems. In *Future Directions in Distributed Computing*, pages 137–141, 2003.

[BD93] R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *Intl. Logic Programming Symposium*, 1993.

[BD97] S. Brass and J. Dix. Characterizations of the disjunctive well-founded semantics: Confluent calculi and iterated gcwa. *Journal of Automated Reasoning*, 1997.

[BD98] S. Brass and J. Dix. Characterizations of the disjunctive well-founded semantics. *Journal of Logic Programming*, 34(2):67–109, 1998.

[BDF+04] M. Bichler, N. Diernhofer, F. Fay, C. Koenig, A. MacWilliams, A. Paschke, T. Setzer, and G. Voelk. Dynamic value webs for it-services - it-service technologies and management,. Siemens sbs / tum research study 10/2004, IBIS, Technical University Munich, 2004.

[BDS93] M. Buchheit, F.M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. of Artificial Intelligence Research*, pages 109–138, 1993.

[BDZ01] S. Brass, J. Dix, and U. Zukowski. Transformation based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, 1(5):497–538, 2001.

[Beh95] H. Behrends. A description of event based activities in database related information systems, report 3/1995. Technical report, Univ. of Oldenburg, 1995.

[Bel87] M. Belzer. Legal reasoning in 3-d. In *1st Int. Conf. on Artificial Intelligence and Law*, New York, 1987. ACM Press.

[Ben97] D. Bennet. *Designing Hard Software: The Essential Tasks.* Prentice-Hall, 1997.

[BG94] C. Baral and M. Gelfond. Logic programming and knowledge representation. *J. of Logic Programming*, 19, 20:73–148, 1994.

[BG04a] B. Bennett and A. P. Galton. A unifying semantics for time and events. *Artificial Intelligence*, 153(1-2):13–48, 2004.

[BG04b] D. Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema, http://www.w3.org/tr/rdf-schema/, accessed june 2005, 2004.

[BGL85] R.J. Brachman, P.V. Gilbert, and H.J. Levesque. An essential hybrid reasoning system: Knowledge and symbol level accounts for krypton. In *Int. Conf. on Artificial Inelligence*, 1985.

[BHS03] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. lecture notes in artificial intelligence. In *Lecture Notes in Artificial Intelligence*. Springer, 2003.

[Bil90] D. Billington. Logic is stable. *J. of Experimental and Theoretical Artificial Intelligence*, 2(1990):151–177, 1990.

[Bil93] D. Billington. Defeasible logic is stable. *Journal of Logic and Computation*, 3:370–400, 1993.

[Bin01] U. Binder. Ehevertrag fuer it dienstleistungen. *Infoweek*, 34(4), 2001.

[BK82] K.A. Bowen and R.A. Kowalski. Amalgamating language and meta-language in logic programming. *Journal of Logic programming*, pages 153–172, 1982.

[BK95] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical report csri-323, University of Toronto, Nov. 1995 1995.

[BL90] N. Bidoit and P. Legay. Well!: An evaluation procedure for all logic programs. In *Int. Conf. on Database Theory*, pages 335–348, 1990.

[BL96] C. Baral and J. Lobo. Formal characterization of active databases. In *Int. Workshop on Logic in Databases*, pages 175–195, 1996.

[BL99] T. Berners-Lee. Weaving the web, 1999.

[BL03] T. Berners-Lee. Www past and future. http://www.w3.org/2003/talks/0922-rsoc-tbl/, accessed jan. 2005, 2003.

[BL05] T. Berners-Lee. Web for real people, http://www.w3.org/2005/talks/0511-keynote-tbl, accessed jan. 2006, 2005.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[BLM90] C. Baral, J. Lobo, and J. Minker. Generalized well-founded semantics for logic programs. In M. E. Stickel, editor, *International Conference on Automated Deduction*. Springer, 1990.

[BLM92] C. Baral, J. Lobo, and J. Minker. Generalized disjunctive well-founded semantics for logic programs. *Annals of Math and Artificial Intelligence*, 11(5):89–132, 1992.

[BLR96] F. Buccafurri, N. Leone, and P. Rullo. Stable models and their computation for logic programming with inheritance and true negation. *The Journal of Logic Programming*, 27(1):5–43, 1996.

[BLR97] F. Bouali, S. Loiseau, and M-C. Rousset. Verification and revision of rule bases. *Proceedings EUROVAV 97 Katholieke Universiteit Leuven, Leuven, Belgium*, pages 253–264, 1997.

[Bol98] R. Bol. Tabulated resolution for the well-founded semantics. *Journal of Logic Programming*, 34(2):67–109, 1998.

[Bol03] H. Boley. Object-oriented ruleml: User-level roles, uri-grounded clauses, and order-sorted terms. In *International workshop on rules and rule markup languages for the semantic web*, Sanibel Island FL, USA, 2003.

[BP05] F. Bry and P.L. Patranjan. Reactivity on the web: Paradigms and applications of the language xchange. In *ACM Symp. Applied Computing*, 2005.

[BPW02] J. Bailey, A. Poulovassilis, and P.T. Wood. An event-condition-action language for xml. In *WWW 2002*, 2002.

[BR91] C. Beeri and R. Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10:255–299, 1991.

[Bra04] G. Bracha. Generics in the java programming language; http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, accessed jan. 2005, 2004.

[Bre96] G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research*, 4:19–36, 1996.

[Bre01] G. Brewka. On the relationship between defeasible logic and well-founded semantics. In *Proc. LPNMR-2001*, pages 121–132, 2001.

[BRM02] BRML. Ibm brml: Business rules markup language, http://xml.coverpages.org/brml.html, accessed jan. 2004, 2002.

[Bry90a] F. Bry. Negation in logic programming: A formalization in constructive logic. In *Information Systems and Artificial Intelligence: Integration Aspects*, volume LNCS Nr. 474, pages 30–46. Springer, 1990.

[Bry90b] F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowlege Engineering*, 5:289–312, 1990.

[BS91] C. Baral and V. S. Subrahmanian. Dualities between alternative semantics for logic programming and non-monotonic reasoning. In *Int. Workshop of Logic Programming and Non-Monotonic Reasoning*, pages 69–86. MIT Press, 1991.

[BS01] F. Baader and W. Snyder. Unification theory. handbook of automated reasoning. pages 445–532. 2001.

[BSC98] P. Bhoj, S. Singal, and S. Chutani. Sla management in federate environments. Technical Report HPL-98-203, HP Internet Systems and Applications Laboratory, December 1998 1998.

[BSI02] BSI. *IT Service Management, Part 1: Specification for service management.* 2002.

[BT00] H. Boley and S. Tabet. Ruleml: The ruleml standardization initiative, http://www.ruleml.org/, 2000.

[BW94] E. Baralis and J. Widom. An algebraic approach to rule analysis by means of triggering and activation graphs. In *VLDB 94*, pages 475–486, 1994.

[Byr80] L. Byrd. Understanding the control flow of prolog programs. In *Proceedings of Workshop on Logic Programming*, 1980.

[Car97] L. Cardelli. *Type Systems.* The Computer Science and Engineering Handbook. CRC Press, 1997.

[Cas81] H. N. Castaneda. The paradoxes of deontic logic: the solution to all of them in one fell swoop. In R. Hilpinen, editor, *New Studies in Deontic Logic*, pages 37–85. Reidl, Dordrecht, 1981.

[CBC93] F.P. Coenen and T. Bench-Capon. *Maintenance of Knowledge-Based Systems: Theory, Techniques and Tools.* Academic Press, London, 1993.

[CC96] C. Collet and T. Coupaye. Composite events in naos. In *Proceedings of Dexa'06*, Zuerich, Switzerland, 1996.

[CCDA+05] A. Cortes-Calabuig, M. Denecker, O. Arieli, B. Van Nuffelen, and M. Bruynooghe. On the local closed-world assumption of data-sources. In *LPNMR 2005*, volume LNAI 3662, pages 145–157. Springer, 2005.

[CCM96] I. Cervesato, L. Chittaro, and A. Montanari. A general modal framework for the event calculus and its skeptical and credulous variants. In *ECAI'96*, 1996.

[CCS90] C.L. Chang, J.B. Combs, and R.A. Stachowitz. A report on the expert systems validation associate (eva). *Journal of Expert Systems with Applications*, 1(3):219–230, 1990.

[CDGL02] D. Calvanese, G. De Giacomo, and M. Lenzerini. Description logics for information integration. In *Computational Logic: Logic Programming and Beyond - Essays in Honour of Robert A. Kowalski (Part II)*, volume LNCS 2408. 2002.

[CGL98] D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *PODS'98*, pages 149–158, 1998.

[CJ02] J. Carmo and A. J.I. Jones. Deontic logic and contrary to duties. In *Handbook of Philosophical Logic*, volume 2nd Edition, volume 8, pages 265–343. Kluwer, 2002.

[CK91] J. Chen and S. Kundu. The strong semantics for logic programs. In *Proceedings of the 6th Int. Symp. on Methodologies for Intelligent Systems*, Charlotte, NC, 1991.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics contexts and detection. In *VLDB 94*, pages 606–617, 1994.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data-Bases*, pages 293–322. New York, 1978.

[CLN98] D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modelling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 229–264. Kluwer Academic Press, 1998.

[CLN00] J. Chomicki, J. Lobo, and S. Naqvi. A logic programming approach to conflict resolution in policy management. In *Principles of Knowledge Representation and Reasoning*, San Francisco, 2000. Morgan Kaufmann.

[CLTSBS03] Z. Chen, C. Liang-Tien, B. Silverajan, and L. Bu-Sung. Ux - an architecture providing qos-aware and federated support for uddi.

In *Int. Conf. onf Web Services (ICWS'03)*, Las Vegas, Nevada, USA, 2003.

[CNF98] G. Cugola, E. D. Nitto, and A. Fuggeta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Int. Conf. on Software Engineering*, 1998.

[Cod71] E. Codd. Alpha: A data base sublanguage founded on the relational calculus of the database relational model. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, San Diego, CA., 1971.

[Cog05] Cognetics. The lucid framework, http://www.cognetics.com/lucid/index.html, accessed oct. 2005. 2005.

[CS90] S. Craw and D. Sleeman. Automating the refinement of kbs. *Proceedings ECAI'90*, 1990.

[CSM+04] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Journal of Web Semantics*, 1:281–308, 2004.

[CSW95] W. Chen, T. Swift, and D.S. Warren. Efficient top-down computation of queries under the well-founded semantics. *J. of Logic Programming*, 24(3):161–199, 1995.

[CW92] W. Chen and D.S. Warren. A goal-oriented approach to computing well-founded semantics. In *Intl. Conf. and Symposium on Logic Programming*, 1992.

[CW93] W. Chen and D.S. Warren. Query evaluation under the well-founded semantics. *Proceedings of Symp. on the Principles of Database Systems*, 1993.

[DAAW06] C. Damasio, A. Analyti, G. Antoniou, and G. Wagner. Supporting open and closed world reasoning on the web. 2006.

[Das00] A. Daskalopulu. Modelling legal contracts as processes. In *11th Int. Conf. and Work. on Databases and Expert Systems Applications*, 2000.

[Dav03] D. Davison. On-demand outsourcing - outsourcing and service provider strategies, service management strategies. Technical Report Delta 2416, Meta Group Delta, August 2003 2003.

[DBC96] U. Dayal, A. Buchmann, and S. Chakravarty. The hipac project. In J. Widom and S. Ceri, editors, *Active Database Systems*. Morgan Kaufmann, 1996.

[DDK+01] A. Dan, D. Dias, R. Kearney, T. Lau, T. Nguyen, F. Parr, M. Sachs, and H. Shaikh. Business-to-business integration with tpaml and a b2b protocol framework. *IBM Systems Journal of Artificial Intelligence*, 40(1), 2001.

[DDK+04] A. Dan, D. Davis, R. Kearney, R. King, A. Keller, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM Systems Journal, Special Issue on Utility Computing*, 43(1):136–158, 2004.

[DDLS01] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Work. on Policies for Distributed Systems and Networks (Policy'01)*, Bristol, UK, 2001.

[DE88] L. M. L. Declambre and J. N. Etheredge. A self-controlling interpreter for the relational production language. In *ACM Sigmod Int. Conf. on the Management of Data*, 1988.

[Den91] R. Denney. Test-case generation from prolog-based specifications. *IEEE Software*, 8(2):49–57, 1991.

[DG84] W. Dowling and H. Gallier. Linear time algorithms for testing the satisfiabiltiy of propositional horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[DGdlBS99] B. Demoen, M. Garcia de la Banda, and P. J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *In Procs. of the 22nd Australian Comp. Sci. Conf.*, pages 217–228, 1999.

[DGKK98] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnstroem. Tal: Temporal action logics language specification and tutorial. *Linkoeping Electronic Articles in Computer and Information Science*, 3(015), 1998.

[DH88] R. Dietrich and A. F. Hagl. A polymorphic type system with subtypes for prolog. In *Proceedings of the 2nd European Symposium on Programming*, pages 79–93. Springer-Verlag, 1988.

[DH04] J. Dietrich and J. Hiller. Mandarax 3.1, http://mandarax.sourceforge.net/, accessed oct. 2004, 2004.

[Die04] J. Dietrich. A rule-based system for ecommerce applications. In *Proceedings of KES 2004*, volume LNAI 3213 p 455, 2004.

[Dix92] J. Dix. A framework for representing and characterizing semantics of logic programs. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, pages 591–602, San Mateo, CA, 1992. Morgan Kaufmann.

[Dix95a] J. Dix. A classification-theory of semantics of normal logic programs: Ii. weak properties. *Fundamenta Informaticae*, XXII(3):257–288, 1995.

[Dix95b] J. Dix. Semantics of logic programs: Their intuitions and formal properties. an overview. In A. Fuhrmann and H. Rott, editors,

Essays on Logic in Philosophy and Artificial Intelligence, pages 241–327. DeGruyter, 1995.

[DK89] P. M. Dung and K. Kanchansut. A natural semantics of logic programs with negation. In *9th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 70–80, 1989.

[DKM84] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

[DKSW03] J. Dietrich, A. Kozlenkov, M. Schroeder, and G. Wagner. Rule-based agents for the semantic web. *Journal on Electronic Commerce Research Applications*, 2003.

[DLNS91] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. A hybrid system with datalog and concept languages. *Trends in Artificial Intelligence*, LNAI 549:88–97, 1991.

[DLS00] P. Doherty, T. Lukasiewicz, and A. Szalas. Efficient reasoning using the local closed-world assumption. In *Proceedings of 9th AIMSA*, volume LNCS 2407, pages 49–58, 2000.

[DM94] J. Dix and M. Mueller. Partial evaluation and relevance for approximations of the stable semantics. In *8th Int. Symp. on Methodologies for Intelligent Systems*, pages 511–520, Charlotte, NC, 1994. Springer.

[DM02] P. M. Dung and P. Mancaralle. Production systems with negation as failure. *IEEE Transactions onf Knowledge and Data Engineering*, 14(2), 2002.

[DMB92] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *ECAI'92*, Vienna, Austria, 1992.

[Doe94] K. Doets. *From Logic to Logic Programming.* MIT Press, Camebridge, MA, USA, 1994.

[DP05] J. Dietrich and A. Paschke. On the test-driven development and validation of business rules. In *ISTA05*, Massey, New Zealand, 2005.

[DSGF03] V. Deora, J. Shao, W. A. Gray, and J. Fiddian. A qualitiy of service management framework based on user expectations. In *Service-Oriented Computing (ICSOC'03).* Springer, 2003.

[Dun91] P. M. Dung. Negation as hypotheses: An abductive foundation for logic programming. In *8th Int. Conf. on Logic Programming*, Paris, 1991. MIT Press.

[Dun93] P. M. Dung. An argumentation semantics for logic programming with explicit negation. In *10th Logic Programming Conf.* MIT Press, 1993.

[DV97] E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex data. In *Proceedings of LFCS'97*. Springer LNCS, 1997.

[DZ92a] P. W. Dart and J. Zobel. Efficient run-time type checking of typed logic programs. *J. Log. Program.*, 14(1-2):31–69, 1992.

[DZ92b] P. W. Dart and J. Zobel. A regular type language for logic programs. In *Types in Logic Programming*, pages 157–187. 1992.

[EFST01] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specification in logic programs. In *IJCAI*, 2001.

[EGW97] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artifical Intelligence*, 89(1-2):113–148, 1997.

[EK76] M. H. Emden and R. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23:733–742, 1976.

[ELST04] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *KR-2004*, 2004.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.

[Eri93] J. Erikson. *CEDE: Composite Event Detector in An Active Database*. PhD Thesis. University of Skoevde, 1993.

[FB02] D. Fensel and C. Bussler. The web service modeling framework (wsmf). White paper, Virje Universiteit Amsterdam, 2002.

[FG98] S. Flesca and S. Greco. Declarative semantics for active rules. In *Int. Conf. on Database and Expert Systems Applications*, volume 1460, pages 871 – 880. Springer LNCS, 1998.

[FH71] D. Follesdal and R. Hilpinen. *Deontic Logic: An introduction*. Deontic Logic: Introductory and Systematic Readings. D. Reidel Publishing, 1971.

[Fit85] M. Fitting. A kripke-kleene semantics of logic programs. *Journal of Logic Programming*, 4:295–312, 1985.

[Fit90] M. Fitting. Well-founded semantics, generalized. In *Int. Symposium of Logic Programming*, pages 71–84, San Diego, 1990. MIT Press.

[Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving*, volume Second Edition. Springer, 1996.

[FN71] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, (2):189–208, 1971.

[FSS+04] A. Farrell, M. Sergot, M. Salle, C. Bartolini, D. Trastour, and A. Christodoulou. Performance monitoring of service level agreements for utility computing using event calculus. In *First IEEE Int. Workshop on Electronic Contracting*, San Diego, CA, 2004.

[FWHH02] D. Fensel, W. Wahlster, Liebermanm H., and J. Hendler. *Spinning the Semantic Web: Bridging the World Wide Web to Its Full Potential*. MIT Press, 2002.

[GB00] A.J. Gonzales and V. Barr. Validation and verification of intelligent systems. *Journal of Experimental and Theoretical AI*, 2000.

[GD93] S. Gatziu and K. Dittrich. Event in an active object-oriented database system. In *Int. Conf. on Rules in Database Systems*, Edinburgh, 1993.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[GHVD03] B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *International World Wide Web Conference*. ACM, 2003.

[GJS92] N. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Int. Conf. on Management of Data*, pages 81–90, San Diego, 1992.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *ICLP'90*, pages 579–597. MIT Press, 1990.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[GL93] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2-4):301–321, 1993.

[GL96] G. De Giacomo and M. Lenzerini. Tbox and abox reasoning in expressive description logics. In *KR'96: Principles of Knowledge Represenation and Reasoning*, pages 316–327, San Francisco, California, 1996. Morgan Kaufmann.

[GL98] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Conf. on Innovative Applications of Artificial Intelligence*, pages 623–630, Menlo Park, CA, 1998. AAAI Press.

[GL99] E. Giunchiglia and V. Lifschitz. Action languages, temporal action logics and the situation calculus. *Linkoeping Electronic Articles in Computer and Information Science*, 4(040), 1999.

[GLC99] B.N. Grosof, Y. Labrou, and H.Y. Chan. A declarative approach to business rules in contracts: Courteous logic programms in xml. In M.P. Wellman, editor, *Conf. on Electronic Commerce (EC-99)*, Denver UK, 1999. ACM Press.

[GLV91] D. Gabbay, E. Laenens, and D. Vermeir. Creulous vs. sceptical semantics for ordered logic programs. In *Int. Conf. on Principles of Knowledge Representation and Reasioning*, pages 208–218, 1991.

[GM86] J. A. Goguen and J. Meseguer. Eqlog: Equality, types and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 295–263. Prentice Hall, 1986.

[GM87] J. A. Goguen and J. Meseguer. Order-sorted algebra i: Partial and overloaded operators, errors and inheritance. Technical report, Computer Science Laboratory, SRI International, 1987.

[Gog86] M. Gogolla. *Ueber partiell geordnete Sortenmengen und deren Anwendung zur Fehlerbehandlung in abstrakten Datentypen.* PhD Thesis. Technische Universitaet Braunschweig, Germany, 1986.

[Gra02] G. Grahne. Information integration and incomplete information. *IEEE Data Engineering Bulleting*, 25(3):46–52, 2002.

[Gre04] W.Van Grembergen. Strategies for information technology governance„ 2004.

[Gro99] B. N. Grosof. A courteous compiler from generalized courteous logic programs to ordinary logic progams. Supplementary update follow-on to ibm research report rc 21472., IBM T.J. Watson Research Center, 1999.

[Gru93] T.R. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5:199–220, 1993.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. W. Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.*, volume Vol. IV of *Current Trends on Programming Methodology.* Prentice-Hall Int.„ 1978.

[HA02] J. Heflin and H. M. Avila. Lcw-based agent planning for the semantic web. *Ontologies and the Semantic Web*, WS-02-11:63–70, 2002.

[Hag01] J. Hage. Contrary to duty obligations - a study in legal ontology. In *Jurix 2001*, Amsterdam, 2001.

[Han65] W. H. Hanson. Semantics for deontic logic. *Loguique et Analyse*, 31:177–190, 1965.

[Han91] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theor. Comput. Sci.*, 89(1):63–106, 1991.

[Han92] M. Hanus. Logic programming with type specifications. In *Types in Logic Programming*, pages 91–140. 1992.

[Hay04] P. Hayes. Rdf semantics, http://www.w3.org/tr/2004/rec-rdf-mt-20040210, accessed dec. 2005, 2004.

[HC02] L. Hendriks and M. Carr. Itil: best practice in it service management. In J. Bon, editor, *The guide to IT Service Management*, pages 131–150. Addison-Wesley, London, 2002.

[HCC94] P. Hentenryck, A. Cortesi, and B. Charlier. Type analysis of prolog using type graphs. In *Conference on Programming Language Design and Implementation*, pages 337–348, 1994.

[HCYY99] X. He, W.C. Chu, H. Yang, and S.J.H. Yang. A new approach to verify rule based systems using petri nets. In *Int. Conf. on Computer Software and Applications*, pages 462–467, Alamitos, CA, USA, 1999.

[HFGL01] Y. Hoffner, S. Field, P. Grefen, and H. Ludwig. Contract-driven creation and operation of virtual enterprises. *Computer Networks, Elsevier Science*, 37:111 – 136, 2001.

[HJ92] N. Heintze and J. Jaffar. Semantic types for logic programs. In *Types in Logic Programming*, pages 141–155. 1992.

[HJW99] H. Herre, J. Jaspars, and G. Wagner. *Partial Logics with Two Kinds of Negation as a Foundation of Knowledge-Based Reasoning.* What is Negation? Kluwer Academic Publishers, 1999.

[HK03] M. Hondo and C. Kaler. Web services policy framework (wspolicy), ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf, 2003.

[HL94] P. M. Hill and J.W. Lloyd. *The Goedel Programming Language.* MIT Press, 1994.

[HMPR04] A. Hevner, S. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–101, 2004.

[Hoa85] C. A. R. Hoare. *Communication and Concurrency.* Prentice Hall, 1985.

[Hor93] J. F. Horty. Nonmonotonic techniques in the formalization of commonsense normative reasoning. In *Proc. Workshop on Nonmonotonic Reasoning*, pages 74–84, 1993.

[HP03] HP. Hp it service management (itsm) - transforming it organizations into service providers. White paper, 2003.

[HPPS+05] I. Horrocks, B. Parsia, P. Patel-Schneider, J. Hendler, F. Francois, and S. Soliman. Semantic web architecture : Stack or two towers? In *PPSWR 2005 : principles and practice of semantic web reasoning*, Dagstuhl Castle , Germany, 2005.

[HPS04] I. Horrocks and P. Patel-Schneider. Reducing owl entailment to description logic satisfiability. *J. Web Sem.*, 1(4):345–357, 2004.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml, http://www.w3.org/submission/swrl/, accessed jan. 2006, 2004.

[HR85] F. Hayes-Roth. Rule based systems. *ACM Computing Surveys*, 28(9), 1985.

[HS88] M. Hohfeld and G. Smolka. *Definite Relations over Constraint Languages, LILOG Report, LR-53.* IWBS, IBM Deutschland, 1988.

[HS90] S. Hoelldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8(3):225–244, 1990.

[HT92] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In *Types in Logic Programming*, pages 1–62. 1992.

[HTT87] J. F. Horty, R. H. Thomason, and D. S. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic networks. In *AAAI'87*, Menlo Park, CA, 1987. AAAI Press.

[HV87] M. Huber and I. Varsek. Extended prolog for order-sorted resolution. In *Symposium of Logic Programming*, pages 34–45, San Francisco, 1987.

[HVNV05] S. Heymans, D. Van Niewenborgh, and D. Vermeier. Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs. In *European Semantic Web Conference (ESWC2005)*, Heraklion, Greece, 2005. Springer.

[HY91] Y. Hu and L. Y. Yuan. Extended well-founded model semantics for general logic programs. in koichi furukawa, editor,. In *Int. Conf. on Logic Programming*, pages 412–425, Paris, 1991.

[ISO99] ISO. Iso/iec jtc 1/sc 7. information technology - open distributed processing - reference model -enterprise language: Iso/iec 15414 | itu-t ecommendation x.911,. committee draft. 8. july 1999. Technical report, 1999.

[ITG98] ITGI. *Control Objectives for Information and related Technology (CobiT)*, volume 3rd edition. 1998.

[itS04a] itSMF. *IT Service Management, an introduction.* 2004.

[ITS04b] ITSMF. It service management forum: What is the itsmf?, http://www.itsmf.com/about/whatis.asp, accessed jan. 2005, 2004.

[Jac96] O. Jack. *Software Testing for Conventional and Logic Programming.* vol. 10 of Programming Complex Systems. Walter de Gruyter und Co., Berlin, New York, 1996.

[JCJ+03] M. Jonson, P. Chang, R. Jeffers, J. Bradshaw, and et al. Kaos semantic policy and domain services: An application of daml to web services-based grid architectures. In *AAMAS'03*, Melbourne, Australia, 2003.

[Jon96] C. Jones. Capers jones language level, http://www.theadvisors.com/langcomparison.htm, accessed nov. 2004, 1996.

[KBBF05] M. Kifer, J. de Bruijn, H. Boley, and D. Fensel. A realistic architecture for the semantic web. In *Proc. of RuleML 2005*, pages 17–29, 2005.

[KBGH06] S Kona, A. Bansal, G. Gupta, and T. Hite. Efficient web service discovery and composition using constraint logic programming. In *ICLP'06 Workshop Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS2006)*, Seattle, WA, 2006.

[KC04] G. Klyne and J. Caroll. Resource description framework (rdf): Concepts and abstract syntax, http://www.w3.org/tr/2004/rec-rdf-concepts-20040210, accessed nov. 2005, 2004.

[Ker98] G. M. Kern. A framework for service management of information systems. *Mid. American journal of business*, 13(1):49–57, 1998.

[KFJ05] L. Kagal, T. Finin, and A. Joshi. Declarative policies for describing web service capabilities and constraints. In *W3C Workshop on Constraints and Capabilities for Web Services*, 2005.

[KK71] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artifical Intelligence*, 2:227–260, 1971.

[KLM90] S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence 11(1980)*, 44(1-2):167–207, 1990.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4), 1995.

[KM97a] A. Kakas and R. Miller. A simple declarative language for describing narratives with actions. *JLP (special issue)*, 31(1-3):157–200, 1997.

[KM97b] M. A. Khamsi and D. Misane. Fixed point theorems in logic programming. *Ann. Math. Artif. Intell.*, 21(2-4):231–243, 1997.

[KMT99] A. Kakas, R. Miller, and F. Toni. An argumentation framework for reasoning about actions and change. In *LPNMR'99*, 1999.

[KNW93] K. Kwon, G. Nadathur, and D. S. Wilson. Implementing polymorphic typing in a logic programming language. Duke–tr–1993–24, 1993.

[Kow92] R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146, 1992.

[KP87] P. Kolaitis and C. Papadimitriou. Why not negation by fixpoint? In *Prod. of PODS-87*, pages 231–239, 1987.

[KP06] A. Kozlenkov and A. Paschke. Prova 2.0 user documentation, http://www.prova.ws/, accessed oct. 2006. Technical report, 2006.

[KPS06] A. Kozlenkov, A. Paschke, and M. Schroeder. Prova, http://prova.ws, accessed jan. 2006. 2006.

[KS86] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KS04] A. Kozlenkov and M. Schroeder. Prova: Rule-based java-scripting for a bioinformatics semantic web. *Proceedings International Workshop on Data Integration in the Life Sciences*, 2004.

[KSS95] D. B. Kemp, D. Srivastava, and P. J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theor. Comput. Sci.*, 146:145–184, 1995.

[Kun87] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Lan51] E. Landau. Foundations of analysis. pages 1–18. Chelsea, 1951.

[LAP01] J.A. Leite, J.J. Alferes, and L.M. Pereira. On the use of multi-dimensional dynamic logic programming to represent societal agents' viewpoints. In *EPIA01*, volume LNAI 2258, pages 276–289, 2001.

[LB87] H. J. Levesque and R. J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence journal*, 3:78–93, 1987.

[LBSB92] G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In *Int. Symp. on Software Reliability Enginnering*, pages 104–113, 1992.

[LDa02] G. Libera-Dell and et. al. Web services security policy language (ws-securitypolicy), 2002.

[LDK04] H. Ludwig, A. Dan, and R. Kearney. Cremona: An architecture and library for creation and monitoring of ws-agreements. In *ICSOC'04*, New York, 2004.

[Lei97] A. Leitsch. *The Resolution Calculus*. Springer, 1997.

[Lei03] J. A. Leite. Evolving knowledge bases. *Frontiers in Artificial Intelligence and Applications*, 81, 2003.

[Lew74] D. Lewis. Semantic analyses for dydic deontic logic. In *Logical Theory and Semantic Analysis*, pages 59–104. Reidel, Dordrecht, 1974.

[LHL95] B. Ludaescher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Int. Conf. on Management of Data*, Pune, India, 1995.

[Lif96] V. Lifschitz. *Foundations of declarative logic programming*. Principles of Knowledge Representation. CSLI publishers, 1996.

[LLM98] G. Lausen, B. Ludascher, and W. May. On logical foundations of active databases. *Logics for Databases and Information Systems*, pages 389–422, 1998.

[Llo87] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of disjunctive logic programming*. MIT Press, 1992.

[LR91] T. L. Lakshman and U. S. Reddy. Typed prolog: A semantic reconstruction of the mycroft-o'keefe type system. In *Proceedings of ISLP*, San Diego, California, USA, 1991.

[LR96] A. Levy and M-C. Rousset. A representation language combining horn rules and description logics. In *European Conference on Artificial Intelligence (ECAI-96)*, 1996.

[LRLLM99] L.M. Laita, E. Roanes-Lozano, L. de Ledema, and V. Maojo. Computer algebra based verification and knowledge extraction in rbs application to medical fitness criteria. In *EUROVAD'99*, 1999.

[LSE03] D. D. Lamanna, J. Skene, and W. Emmerich. Slang: A language for defining service level agreements. In *Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, San Juan, Puerto Rico, 2003.

[LT01] Z. Lonc and M. Truszcynski. On the problem of computing the well-founded semantics. *Theory and Practice of Logic Programming*, 1(5):591–609, 2001.

[Lu98] L. Lu. Polymorphic type analysis in logic programs by abstract intepretation. *Journal of Logic Programming*, 36(1):1–54, 1998.

[LW92] V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasioning (preliminary report). In *Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'92)*. Morgan-Kaufman, 1992.

[Mah00] M. J. Maher. A denotational semantics of defeasible logic. In *Computational Logic*, volume LNCS 1861, pages 209–222. Springer, 2000.

[Mah01] M. J. Maher. Propositional defeasible logic has linear complexity. *Theory and Practice of Logic Programming*, 1(6):691 – 711, 2001.

[Mah02] M.J. Maher. A model-theoretic semantics for defeasible logic. In H. Decker, J. Villadsen, and T. Waragai, editors, *ICLP2002 Workshop on Paraconsistent Computational Logic*, pages 67–80, Roskilde, Denmark, 2002. Datalogiske Skrifter, vol. 95. Roskilde University,.

[Mak86] D. Makinson. On the formal representation of rights relations. *Journal of Philosophical Logic*, 15:403–425, 1986.

[Mal26] E. Mally. *Grundgesetze des Sollens. Elemente der Logik des Willens.* Leuschner and Lubensky, Graz, 1926.

[Mar91] V.W. Marek. Autoepistemic logic. *Journal of the ACM*, 38(3):588–619, 1991.

[MBL+06] J. Mei, H. Boley, J. Li, V. C. Bhavsar, and Z. Lin. Datalogdl: Datalog rules parameterized by description logics. In *Canadian Semantic Web Working Symposium*, Quebeck, Canada, 2006.

[McC59] J. McCarthy. Programs with common sense. In *Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959.

[McC80] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Journal of Artificial Intelligence*, 13(1-2):27–39, 1980.

[McC83] J. McCarthy. Permissions and obligations. In *Int. Conf. on Artifical Intelligence*, pages 287–294, Karlsruhe, 1983. Morgan Kaufmann.

[McC92] J. McCarthy. Defeasible deontic reasoning. In *4th Int. Workshop on Nonmonotonic Reasoning*, Plymouth, 1992.

[MG99] M.J. Maher and G. Governatori. A semantic decomposition of defeasible logics. In *In Proc. AAAI/IAAI-1999*, pages 299–305, 1999.

[MGL+04] Z. Milosevic, S. Gibson, P.F. Linigton, J. Cole, and S. Kulkarni. On desing and implementation of a contract monitoring facility. In *1st IEEE Int. Workshop on Electronic Contracting*, San Diego, CA, 2004.

[MH69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[MH04] D. L. McGuinness and F. v. Harmelen. Owl web ontology language, http://www.w3.org/tr/owl-features/, accessed june 2005, 2004.

[MIB04] MIB. Ietf sim/mib, http://snmp.cs.utwente.nl/ietf/ rfcs/rfcby-module.html, accessed, nov. 2005, 2004.

[Mic02] Microsoft. *Microsoft Operations Framework Executive Overview.* 2002.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

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

[Min93] J. Minker. An overview of nonmonotonic reasoning and logic programming. *Journal of Logic Programming*, 17(2-4):95–126, 1993.

[Mis84] P. Mishra. Towards a theory of types in prolog, 1984.

[MJSSW03] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne. Contract representation for run-time monitoring and enforcement. In *IEEE Int. Conf. on E-Commerce (CEC),*, Newport Beach, USA, 2003.

[MKB00] R. Meolic, T. Kapus, and Z. Brezonic. Verification of concurent systems using actl. In *IASTED Int. Conf. on Applied Informatics (AI'00)*, pages 663–669, Anaheim, Calgary, 2000. IASTED/ACTA Press.

[MKB03] R. Meolic, T. Kapus, and Z. Brezonic. An action computation tree logic with unless operator. In *Proc. of South-East European Workshop on Formal Methods (SEEFM'03)*, pages 100–114, Thessaloniki, Greece, 2003.

[MLF02] T. Millstein, A. Levy, and M. Friedman. Query containment for data integration systems. In *Proc. 21st PODS*, pages 67–75, 2002.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Programming Lnaguages and Systems*, 4:258–282, 1982.

[MM01] O. Marjanovic and Z. Milosevic. Towards formal modeling of e-contracts. In *Enterprise Distributed Object Computing Conference (EDOC '01)*, Seattle, WA, USA, 2001.

[MN86] D. Miller and G. Nadathur. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, pages 448–462, 1986.

[MNFS91] D. Miller, G. Nadathur, Pfenning F., and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125 – 157, 1991.

[MO84] A. Mycroft and R. A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984.

[Moo85] R. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[Mor96] S. Morishita. An extension of van gelder's alternating fixpoint to magic programs. *Journal of Computer and System Sciences*, 52:506–521, 1996.

[MPC96] R. Meo, G. Psaila, and S. Ceri. Composite events in chimera. In *EDBT*, Avingnon, France, 1996.

[MR85] P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 7–21, New Orleans, Louisiana, United States, 1985.

[MS98] M. Marchiori and J. Saarela. Metalog: Query + metadata + logic = metalog, http://www.w3.org/tands/ql/ql98/pp/metalog.html, 1998.

[MS99] R. Miller and M. Shanahan. The event calculus in classical logic - alternative axiomatisations. *Electronic Transactions on Artificial Intelligence*, 3:77–105, 1999.

[MS03] Meteor-S. Meteor-s semantic web service annotation framework (mwsaf) , http://lsdis.cs.uga.edu/projects/meteor-s/, accessed nov. 2006, 2003.

[MSS05] B. Motik, U. Sattler, and R. Studer. Query answering for owl-dl with rules. *Journal of Web Semantics*, 3(1):41–60, 2005.

[Mul06] Mule. Mule enterprise service bus, http://mule.codehaus.org/display/mule/home, 2006.

[MW98] D. McGuinness and J. R. Wright. An industrial strength description logic-based configuration platform. In *IEEE Intelligent Systems*, pages 69–77, 1998.

[Nai92] L. Naish. *Types and the Intended Meaning of Logic Programs.* In: Types in Logic Programming. MIT Press, 1992.

[NBB+02] D. Nardi, R.J. Brachman, F. Baader, W. Nutt, F.M. Donini, U. Sattler, D. Calvanese, R. Molitor, G. De Giacomo, R. Kuesters, F. Wolter, D. L. McGuinness, P. F. Patel-Schneider, R. Moeller, V. Haarslev, I. Horrocks, A. Borgida, C. Welty, A. Rector, E. Franconi, M. Lenzerini, and R. Rosati. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2002.

[NK80] D.L. Nazareth and M.H. Kennedy. Static and dynamic verification, validation and testing: The evolution of a discipline. In *AAA'90*

on Knowledge-based Systems, Validation Verification and Testing, Boston, 1980.

[NK88] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *ACM Symposium on Principles of Database Systems*, pages 251–262, New York, 1988. ACM.

[Nut94] D. Nute. Defeasible logic. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 3*. Oxford University Press, 1994.

[Obe62] A. Oberschelp. Untersuchungen zur mehrsortigen quantorenlogik. *Math. Ann.*, 145:297–333, 1962.

[OGC00a] OGC. *Office of Government Commerce: IT Infrastructure Library*. The Stationary Office, London, 2000.

[OGC00b] OGC. *Office of Government Commerce: Service Support*. The Stationary Office, London, 2000.

[OGC01] OGC. *Office of Government Commerce: Service Delivery*. The Stationary Office, London, 2001.

[Ogl03] et. al. Ogle, D. The common base event, ibm report. Technical report, IBM, 2003.

[Or06] OASIS-refmod. Oasis soa reference model, http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, accessed nov 2006, 2006.

[OS03] OWL-S. http://www.daml.org/services/owl-s, accessed feb. 2005, 2003.

[PA92] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. *Proceedings of ECAI'92*, 1992.

[PAA92] L.M. Pereira, J.J. Alferes, and J.N. Aparicio. Adding closed world assumptions to well founded semantics. In *Fifth Generation Computer Systems*, pages 562–569, 1992.

[Pas04a] A. Paschke. Rbsla: Rule based service level agreement markup language, http://ibis.in.tum.de/projects/rbsla/index.php, accessed jan. 2006, 2004.

[Pas04b] A. Paschke. Rule based sla management - a rule based approach on automated it service management (in german language). Working paper, IBIS, TUM, Technical Report, 6/2004, 06/2004 2004.

[Pas05a] A. Paschke. Eca-lp: A homogeneous event-condition-action logic programming language. Technical report, Internet-based Information Systems, Technical University Munich, http://ibis.in.tum.de/research/projects/rbsla/, November, 2005.

[Pas05b] A. Paschke. Eca-ruleml: An approach combining eca rules with interval-based event logics. Working paper, IBIS, TUM, Technical Report, 11/2005 2005.

[Pas05c] A. Paschke. Eca-ruleml: An event-condition-action rule markup language, http://ibis.in.tum.de/research/projects/eca-ruleml/, 2005.

[Pas05d] A. Paschke. Npl: Negotiation pattern language. Technical report, 01/2005, IBIS, TUM, 2005.

[Pas05e] A. Paschke. Owl2prova: Homogeneous and heterogeneous integration of description logics into logic programming, http://prova.ws/forum/viewtopic.php?t=152, accessed dec. 2005, 2005.

[Pas05f] A. Paschke. Rbsla - a declarative rule-based service level agreement language based on ruleml. In *International Conference on Intelligent Agents, Web Technology and Internet Commerce (IAWTIC 2005)*, Vienna, Austria, 2005.

[Pas05g] A. Paschke. Typed hybrid description logic programs with order-sorted semantic web type systems based on owl and rdfs. Working paper, IBIS, TUM, Technical Report, 12/2005 2005.

[Pas06a] A. Paschke. Eca-ruleml 0.2: A homogeneous event-condition-action logic programming language for ruleml, http://ibis.in.tum.de/research/reactionruleml/0.1/docs/reaction-ruleml-talk-2006-9-06.pdf, 2006.

[Pas06b] A. Paschke. Eca-ruleml/eca-lp: A homogeneous event-condition-action logic programming language, http://2006.ruleml.org. In *Int. Conf. of Rule Markup Languages (RuleML'06)*, Athens, Georgia, USA, 2006.

[Pas06c] A. Paschke. Reaction ruleml, http://ibis.in.tum.de/research/reactionruleml/ events/reactionrulemlevent06.htm, accessed, nov. 2006. In *Special Event on Reaction RuleML at ISWC'06/RuleML'06*, Athens, Georgia, USA, 2006.

[Pas06d] A. Paschke. Reaction ruleml part of tutorial bridging research and practice - making real-world application of ontologies and rules, http://2006.ruleml.org/. In *Int. Conf. on Rule Markup Languages (RuleML'06)*, Athens, GA, USA, 2006.

[Pas06e] Kiss C. Al-Hunaty S. Paschke, A. Npl: Negotiation pattern language - a design pattern language for decentralized (agent) coordination and negotiation protocols. In R. Banda, editor, *E-Negotiation - An Introduction*. ICFAI University Press, ISBN 81-314-0448-X, 2006.

[Pat99] N.W. Paton. *Active Rules in Database Systems.* Monographs in Computer Science. Springer, 1999.

[PB05] A. Paschke and M. Bichler. Sla representation, management and enforcement - combining event calculus, deontic logic, horn logic and event condition action rules. In *EEE05*, Hong Kong, China, 2005.

[PCFW95] N. Paton, J. Campin, A. Fernandes, and M. Williams. Formal specification of active database functionality: A survey. In *2nd Int. Workshop RIDS-95*, Athens, Greece, 1995.

[Pet03] R. Peterson. Integration strategies and tactics for information technology governance. In W.Van Grembergen, editor, *Strategies for Information Technology Governance.* Idea Group Publishing, 2003.

[Pfe92] F. Pfenning. *Types in Logic Programming.* The MIT Press, 1992.

[PG03] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.

[PH02] A. Paschke and W. Huemmer. Xml - hoeherwertige mechanismen. In L Schlesinger, W. Huemmer, and A. Bauer, editors, *Heterogene Informationssysteme*, volume Band 25. FAU, Erlangen, 2002.

[PH04] F. Pan and J. Hobbs. Owl time, http://www.isi.edu/ pan/owl-time.html, accessed jan. 2006, 2004.

[PKB+06] A. Paschke, A. Kozlenkov, H. Boley, M. Kifer, S. Tabet, M. Dean, and K. Barrett. Reaction ruleml, http://ibis.in.tum.de/research/reactionruleml/, 2006.

[PKH05] B. Parsia, V. Kolovski, and J. Hendler. Expressing ws-policies in owl. In *Policy Management for the Web Workshop*, 2005.

[PKH06] A. Paschke, A. Kozlenkov, and B. Harold. Reaction ruleml consensual presentation, http://ibis.in.tum.de/research/reactionruleml/docs/rrcp.pdf, accessed nov. 2006. White paper, 2006.

[Pla03] R.T. Plant. Tools for validation and verification of knowledge-based systems 1985-1995, internet source, 2003.

[Plo70] G.D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5, 1970.

[PP88] H. Przymusinska and T.C. Przymusinski. Weakly perfect semantics for logic programs. In *5th International Conference and Symposium on Logic Programming*, pages 1106–1121, 1988.

[PP90] H. Przymusinska and T.C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.

[PPW04] G. Papamarkos, A. Poulovassilism, and P.T. Wood. Rdftl: An event-condition-action rule language for rdf. In *Hellenic Data Management Symposium (HDMS04)*, 2004.

[Pre01] A.D. Preece. *Evaluating verification and validation methods in knowledge engineering.* University of Aberdeen, 2001.

[Prz88] T.C. Przymusinski. Perfect model semantics. In *5th Int. Conf. and Symp. on Logic Pro- gramming*, pages 1081–1096, Cambridge, Ma, 1988. MIT Press.

[Prz89a] T.C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. *Proceedings of ACM Symp. on Principles of Database Systems*, pages 11–21, 1989.

[Prz89b] T.C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasonig*, 5:167–205, 1989.

[Prz90a] T.C. Przymusinski. Non-monotonic reasoning vs. logic programming: A new perspective. In D. Partridge and Y. Wilks, editors, *The Foundations of Artifical Intelligence - A Sourcebook.* Camebridge University Press, London, 1990.

[Prz90b] T.C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13:445–463, 1990.

[Prz91] T.C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.

[PS94] A.D. Preece and R. Shinghal. Foundations and applications of knowledge base verification. *Int. J. of Intelligent Systems*, 9:683–701, 1994.

[PSG06] A. Paschke and E. Schnappinger-Gerull. A categorization scheme for sla metrics. In *Multi-Conference Information Systems (MKWI06)*, Passau, Germany, 2006.

[PSHH04] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. Owl web ontology language semantics and abstract syntax. w3c recommendation 10 february 2004. http://www.w3.org/tr/owl-semantics/., 2004.

[Ras92] L. Raschid. A semantics for a class of stratified production system programs. Technical report, Univ. of Maryland Institute for Advanced Computer Studies-UMIACS-TR-91-114.1, 1992.

[Red88] U. S. Reddy. Notions of polymorphism for predicate logic programs. In *Proc of the 5th International Symposium on Logic Programming*, 1988.

[Rei80] R. Reiter. A logic for default reasoning. *Journal of Artificial Intelligence*, 13:81–132, 1980.

[Rei82] R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 191–233, 1982.

[Rei01] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamic Systems.* MIT Press, Camebridge, 2001.

[Rei02] Rei. Rei, http://rei.umbc.edu/, accessed jan. 2004, 2002.

[Ric05] R. Riccardo. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, 3(1), 2005.

[RIF05] RIF. W3c rif: Rule interchange formant, http://www.w3.org/2005/rules/, accessed oct. 2005, 2005.

[RKL+05] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.

[RL96] L. Raschid and J. Lobo. Semantics for update rule programs and implementation in a relational database management system. *ACM Transactions on Database Systems*, 22(4):526–571, 1996.

[RLM89] A. Rajasekar, J. Lobo, and J. Minker. Weak generalized closed world assumption. *Journal of Automated Reasonig*, 5(3):293–307, 1989.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution-principle. *JACM*, 12(1):23–41, 1965.

[Ros92] K. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13(1):1–22, 1992.

[Ros94] K. Ross. Modular stratification and magic sets for datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, 1994.

[RSYS97] M. Ramaswamy, S. Sarjar, and C. Ye Sho. Using directed hypergraphs to verify rule-based expert systems. *IEEE TKDE*, 9(2):221–237, 1997.

[Rul06] RuleCore. Rulecore, http://www.rulecore.com, accessed august. 2006, 2006.

[Sal04] M. Salle. It service management and it governance: Review, comparative analysis and their impact on utility computing. Technical report hpl-2004-98, HP Labs, 2004.

[SAM01] A. Sahai, Durante. A., and V. Machiraju. Towards automated sla management for web services. Research report hpl-2001-310, HP Labs Palo Alto, 2001.

[San89a] E. Sandewall. Combining logic and differential equations for describing real world systems. In *KR89*. Morgan Kaufman, 1989.

[San89b] E. Sandewall. *Features and Fluents: The Representation of Knowledge about Dynamic Systems*, volume 1. Oxford University Press, Oxford, 1989.

[SBV06] SBVR. Omg sbvr: Semantics of business vocabulary and business rules, http://www.omg.org/docs/dtc/06-03-02.pdf, accessed oct. 2006, 2006.

[Sch92] J. Schlipf. Formalizing a logic for logic programming. *Annals of Mathematics and Artificial Intelligence,*, 5:279–302, 1992.

[Sch95] J. Schlipf. The expressive powers of logic programming semantics. *J. Computer and System Sciences*, 51(1):64–86, 1995.

[SD02] M. Sintek and S. Decker. Triple - a query, inference and transformation language for the semantic web. In *ISWC'02*, pages 364–378. Springer, 2002.

[Sha82] E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1982.

[Sha90] M. Shanahan. Representing continous change in the event calculus. In *ECAI'90*, 1990.

[Sha97a] M. Shanahan. *Solving the Frame Problem*. MIT Press, London, 1997.

[Sha97b] M.P. Shanahan. Event calculus planning revisited. In *4th European Conference on Planning (ECP97)*, volume 1348, pages 390–402. Springer LNAI, 1997.

[She88] J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases*, pages 19–88. Morgan Kaufmann, 1988.

[She91] J.C. Shepherdson. Unsolvable problems for sldnf resolution. *J. of Logic Programming*, pages 19–22, 1991.

[Sie89] J. Siekmann. Unification theory. *J. of Symbolic Computation*, 7:207–274, 1989.

[SK95] F. Sadri and R. Kowalski. Variants of the event calculus. In *Int. Conf. on Logic Programming*, Kanagawa, Japan, 1995.

[SLR93] T. Sellis, C.C. Lin, and L. Raschid. Coupling production systems and database systems. In *ACM Sigmond Int. Conf. on the Management of Data*, 1993.

[SMJ00] R. Sturm, W. Morris, and M. Jander. *Foundations of Service Level Management*. SAMS, Indianapolis, 2000.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types, PhD Thesis*. 1989.

[SNGM89] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. Order sorted equational computation. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*. Academic Press, New York, 1989.

[SOA06] SOA. Service oriented architecture, wikipedia, http://en.wikipedia.org/wiki/service-oriented_architecture, accessed oct. 2006, 2006.

[SRAAW03] A. ShaikhAli, O. F. Rana, R. Al-Ali, and D. W. Walker. Uddie: An extended registry for web services. In *Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, Orlando, Florida, 2003.

[SRG96] V. A. Saraswat, Jagadeesan R., and V. Gupta. Timed default concurrency constraint programming. *Journal of Symbolic Computation*, 22(5/6), 1996.

[SS97] P. J. Stuckey and S. Sudarsham. Well-founded ordered search: Goal-directed bottom-up evaluation of well-founded models. *The Journal of Logic Programming*, 32(3):171–205, 1997.

[Sti86] M. E. Stickel. Schubert's steamroller problem: Formulations and solutions. *Journal of Automated Reasoning*, 2:89–101, 1986.

[Str93] K. Stroetman. A completeness result for sldnf resolution. *The Journal of Logic Programming*, 15(21):337–357, 1993.

[Str00] C. Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, 2000.

[Sup01] J. Suppan. *Service Level Management: Die Koenigsdiziplin des System-Managements*. Das Netzwerk Insider Jahrbuch. Dr. Suppan International Institute, 2001.

[SVSM03] K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding semantics to web services standards. In *ICWS'03*, 2003.

[SWE05] SWEET. Sweetrules: http://sweetrules.projects.semwebcentral.org/, accessed jan. 2006, 2005.

[SWS04] SWSL. Swsl: Semantic web services language, http://www.daml.org/services/swsl/, accessed oct. 2005, 2004.

[SWS05] SWSF. Semantic web services framework (swsf), http://www.daml.org/services/swsf/1.0/overview/, accessed july 2006, 2005.

[SYY02] Y.-D. Shen, L.-Y. Yuan, and J.-H. You. Slt-resolution for the well-founded semantics. *Journal of Automated Reasoning*, 28(1):53–97, 2002.

[SZ91] D Sacca and C. Zaniolo. Partial models and three-valued models in logic programs with negation. In *Workshop of Logic Programming*

and Non-Monotonic Reasoning, pages 87–104, Washington D.C, 1991. MIT Press.

[Tae77] S.A. Taernlund. Horn clause computability. *BIT*, 17:215–216, 1977.

[Teu93] F. Teusink. A proof procedure for extended logic programs. In *ILPS'93*, MIT Press, 1993.

[TGN+03] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Jchiller. A concept for qos integration in web services. In *Int. Web Services Quality Worksop (WQW'03)*, Rome, Italy, 2003.

[Thi99] M. Thielscher. From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111:277–299, 1999.

[TK01] M. Thorpe and C. Ke. Simple rule markup language (srml) - a general xml rule representation for forward-chaining rules, http://xml.coverpages.org/srml.html, accessed june 2005, 2001.

[TPP+03] V. Tosic, B Pagurek, K. Patel, B. Esfandiari, and W. Ma. Management applications of the web service offerings language (wsol). In *15th Int. Conf. on Advanced Information Systems Engineering (CaiSE03)*, Velden, Austria, 2003.

[TS86] H. Tamaki and T. Sato. Old resolution with tabulation. In *3rd Int. Conf. on Logic Programming*, pages 84–98, London, 1986.

[TWS+04] S. Tabet, G. Wagner, S. Spreeuwenberg, P. Vincent, G. Jacques, C. de Sainte Marie, J. Pellant, J. Frank, and J. Durand. Omg prr: Production rule representation, http://www.w3.org/2004/12/rules-ws/paper/53/, accessed oct. 2005, 2004.

[Ull89] J. D. Ullman. *Principles of Database and Knowlegebase Systems*. Vol. 2. Computer Science Press, Rockville, 1989.

[VAG05] K. Verma, R. Akkiraju, and R. Goodwin. Semantic matching of web service policies. In *SDWP '05 Workshop*, 2005.

[VG89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *8th ACM SIGACT-SIGMOND-SIGART Symposium on Principles of Database Systems*, pages 1–10, 1989.

[VG93] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

[VSB84] W. VanMelle, H. Shortliffe, and G. Buchanan. Emycin: A knowledge engineer's tool for constructing rule-based expert systems.

In *Rule Based Expert Systems*, pages p.301–313. Addison-Wesley, 1984.

[W3C01] W3C. The semantic web, http://www.w3.org/2001/sw/, accessed oct. 2006, 2001.

[Wag02] G. Wagner. How to design a general rule markup language. In *XML Technologien fuer das Semantic Web - XSW 2002, Proceedings zum Workshop*, pages 19–37, 2002.

[Wal87] C. Walther. *A Many-Sorted Calculus Based on Resolution and Paramodulation*. Research Notes in Artificial Intelligence. Pitman London and Morgan Kaufman, Los Altos, CA, 1987.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[WCB01] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for wellconditioned, scalable internet services. In *Proceedings of Eighteeth Symposium on Operating Systems (SOSP-18)*, Chateau Lake Louise, Canada, 2001.

[Web04] WebService. Web service architecture, http://www.w3.org/tr/2004/note-ws-arch-20040211/, accessed dec. 2005, 2004.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *ACM Sigmond Int. Conf. on the Management of Data*, 1990.

[Wid92] J. Widom. A denotational semantics for starbust production rule language. *SIGMOD record*, 21(3):4–9, 1992.

[WL97] C.H. Wu and S.J. Le. Knowledge verification with an enhanced high-level petri-net model. *Journal of IEEE Expert*, pages 73–80, 1997.

[Wri51] G.H. Wright. Deontic logic. *Mind*, 60:1–15, 1951.

[WSM05a] WSML. Web services modelling language, http://www.wsmo.org/wsml/, 2005.

[WSM05b] WSMO. Web services modelling ontology, http://www.wsmo.org/, accessed nov. 2006, 2005.

[WSO05] WSOL. Wsol: Web services offering language, http://flash.lakeheadu.ca/ vtosic/wsolpublications.html, accessed nov. 2005, 2005.

[WTB03] G. Wagner, S. Tabet, and H. Boley. Mof-ruleml: The abstract syntax of ruleml as a mof model. In *OMG Meeting*, Boston, 2003.

[WTM+04] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvello, and P. Devanbu. Glueqos: Middleware to sweeten quality-of-service policy interactions. In *ICSE 2004*, pages 189–199, 2004.

[XHT06] XHTML. Xhtml modularization, http://www.w3.org/tr/2006/pr-xhtml-modularization-20060213/, accessed june 2006, 2006.

[XW88] J. Xu and D.S. Warren. A type inference system for prolog. In *Proceedings of ICLP SLP'88*, pages 604–619, 1988.

[YKZ03] G. Yang, M. Kifer, and C. Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *Proceedings of ODBASE'03*, pages 671–688, 2003.

[You04] C.M. Young. An introduction to it service management. Research note, com-10-8287, Gartner, 2004.

[YY90] L.H. You and L. Y. Yuan. Three-valued formalization of logic programming: is it needed. In *Proceedings of 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 172–182. ACM Press, 1990.

[Zan93] C. Zaniolo. A unified semantics for active databases. In Springer, editor, *Int. Workshop on Rules in Database Systems*, pages 271–287, Edinburgh, U.K., 1993.

[Zan95] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *Int. Conf. on Deductive and Object-Oriented Databases*, pages 55–72, 1995.

[ZHM97] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[Zob87] J. Zobel. Derivation of polymorphic types for prolog programs. In *Proceedings of ICLP*, pages 817–838, 1987.

[Zol05] E. Zolin. Complexity of reasoning in description logics, http://www.cs.man.ac.uk/ ezolin/logic/complexity.html, accessed dec. 2005, 2005.