The ContractLog Approach Towards Test-driven Verification and Validation of Rule Bases - A Homogeneous Integration of Test Cases and Integrity Constraints into Evolving Logic Programs and Rule Markup Languages (RuleML)

Adrian Paschke, Technical Report 10/2005, IBIS, TUM, 2005.

Internet-based Information Systems, Dept. of Informatics, TU Munich, Germany
adrian.paschke@gmx.de

Abstract. Rules are often being used as a (declarative) programming language to describe real-world decision logic such as business rules or contract/policy logic and create production systems upon (see e.g. RBSLA/ContractLog project hosted at Sourceforge). For this reason, it is important to support testing mechanisms, that can help rule programmers to determine the reliability of the results produced by their rule systems. Different approaches and methodologies to verification and validation (V&V) of rule-based systems have been proposed in the literature. Simple operational debugging approaches which instrument the rule system and explore its execution trace using e.g., spy and trace commands, place a huge cognitive load on the user, who needs to analyze each step of the inference process. On the other hand, typical heavy-weight validation methodologies 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 rules with collaborations of different roles such as domain experts, system developers and knowledge engineers. In this paper, we adapt a recent trend in software engineering (SE), namely test driven development, to define self validating rule bases in the domain of logic programming, one of the most prominent and successful approaches in declarative rule based programming. We argue that test cases, an agile development methodology of Extreme Programming (XP), have a huge potential to be a successful tool for declarative verification and validation of LP-based rule bases. Test cases in combination with other XP related techniques such as test coverage measurement or evolving rule base refinements (by applying *refactorings* which improve the existing rule code, e.g., via removing inconsistencies, redundancy or missing knowledge without breaking its functionality) are suitable for typically frequently changing requirements and models of rule-based projects. Due to their inherent simplicity test cases better facilitate rule interchange in distributed applications and support different roles which are involved during the rule engineering process. 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 extensional facts and intensional rules at runtime. Here the test cases are used as highly expressive integrity constraints, e.g. as post conditional tests in ECA(P) rules. If the post condition test in an ECAP rule fails the update action which adds or removes rules or facts from the KB is completely "rolled back". Another very interesting application domain of test cases for logic programs (LPs) is rule interchange. Test cases can be used to ensure correct execution of an interchanged LP in a target execution environment by validating the LP in the target inference engine with the attached and interchanged test cases. Meta annotations of the inference engine and the interchanged LPs disclosing the supported/intended semantics (e.g. well-founded semantics, stable model semantics etc.) of both the engine and the program together with appropriate test case variants for different semantics can be used to decide wether the interchanged rule base can be executed correctly in the target rule system. In case such annotations are not given, test cases with meta test programs (meta test rules) as assertions allow analyzing typical properties of LP based semantics. An adequate combination of meta tests for semantic properties uniquely characterizes a particular semantics and hence allows automatic determination of the semantics supported by the target inference engine.
We show that test cases can be represented in a homogeneous LP representation language. The major advantage of this homogenous integration is, that the test cases and the test suites can be managed, maintained and executed within the same rule based environment, based on a well-defined declarative LP-related semantical basis. We further investigate, 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 a particular LP and use the novel notion of test coverage to give feedback and hints on how to optimize and refine the test cases and the rule code in an iterative process. We discuss the implementation of these ideas in the ContracLog KR and present concrete syntax and semantics of test cases for general LPs. We further extend this approach to support well-adopted SE test frameworks like JUnit [23] and to facilitate a very tight integration of tests into existing IDEs such as Eclipse and tools such as Ant. In addition to the representation of test cases in a Prolog-like scripting syntax, we introduce a markup serialization as an extension to the emerging Semantic Web Rule Markup Language RuleML.

Key words: Verification, Validation and Integrity (V&V&I), Declarative Debugging, Logic Programming, Test Cases, Integrity Constraints, Test Coverage, Weak and Strong Semantic Properties, Meta Test Programs, Dynamic LPs, ID-based Updates, Evolving Rule Bases, Knowledge Engineering, Test-drive Rule Development, Rule-Interchange, Software Engineering, Extreme Programming, Agile Development

1 On the Need of Verification and Validation in Rule-based Engineering of Logic Programs

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 applications in commercial software have been limited throughout the past years. However, increasing interest in industry and academia in rule-based applications encompassing not only low-level operational decision rules or database rules/triggers, but also higher-level quality of service (QoS), business rule decision logics or contractual agreements such as service level agreements (SLAs) or policies has led to much recent development - see e.g. the rule based Service Level Agreement project (RBSLA - http://ibis.in.tum.de/staff/paschke/rbsla/index.htm) or SweetRules (http://sweetrules.projects.semwebcentral.org/). These rule-based application domains appear to be particularly suitable to logic programming. For example, IT service providers need to manage large amounts of SLAs / policies / business rules which describe various decision, behavioral or business logic using different rule types such as deeply nested conditional clauses (derivation rules), reactive or even proactive behavior (ECA rules), normative statements (deontic rules), legal rules (compliance rules) or integrity definitions (integrity constraints). This rule types have been shown to be adequately represented as logic programs (LPs)- see e.g. the ContractLog KR for an expressive combination of different logical formalisms into the framework of logic programming. For such rule-based applications, which are often managed in a distributed way and are interchanged between domain boundaries, the knowledge including the intensional rules must necessarily be modelled and maintained evolutionary, in a close collaboration between practitioners, domain experts and rule engineers. The rules are not of static nature and need to be continuously adapted to changing needs. Furthermore, the derived conclusions and results need to be highly reliable and traceable to count even in the legal sense. For this reason, it is important to support expressive but nevertheless easy to use V&V mechanisms, that can help rule engineers to determine the reliability of the results produced by their rule systems, ensure the correct execution in a target rule inference environment and safeguard the life cycle of rules in rule-based projects which are likely to change frequently.

In this paper we endeavor to adapt test-driven V&V techniques from Software Engineering to rule-based knowledge engineering in the domain of logic programming. Although Extreme Programming techniques [7, 6] and similar approaches to "agile" software engineering have been applied very successful in recent years and are widely used among mainstream software developers, its ideas have not been transferred into the recent rule-based application communities in areas such as rule-based policy, contract or business rule representation. However, especially as rule-bases grow larger and more complex such SE principles become more important. The exchange of rule sets and future growth of

distributed rule-based projects will be seriously obstructed if developers do not firmly face the problem of reliability. Test cases, an approach from agile software development in Extreme Programming, can help here. They represent a *black box* view on the program and are significantly simpler than the code they describe. Hence, test cases in combination with other SE methodologies such as *test coverage measurement* and *refactorings* are geared for typically frequently changing requirements and models of rule-based projects. In particular, rule base refinements by applying refactorings which improve the existing rule code, e.g. by removing inconsistencies, redundancy or missing knowledge without breaking the functionality of the rule program, leads to optimized programs. Moreover, due to its inherent simplicity test cases better support different roles which are involved during the LP engineering process and helps to safeguard the life cycle of LPs. Using the test cases as constraints which describe the intended model of a LP gives rule engineers an expressive but nevertheless easy to use modelling language, and makes LPs self validating, i.e the LP is correct if and only if the integrated test cases succeed. Changing requirements and detected faults (bugs) are first translated into new test cases, rendering the LP incorrect with respect to the new set of test cases. The LP is then modified until the old and the new test cases succeed. There are several other SE technologies to facilitate test driven development. Test coverage metrics can be used to quantify the completeness and quality of test cases for V&V of a particular program. Test frameworks like JUnit facilitate a tight integration of tests into code and allow for automated testing and test report creation. We attempt to adapt these values, principles and practices of test-drive development in Extreme Programming to the agile development of Logic Programs and rule-based knowledge engineering. We develop a homogeneous representation approach based on general LPs (GLPs) where rules and test cases are written in the same representation language. We have implemented the test-driven V&V approach in the ContractLog KR and applied it successfully in the rule based SLA (RBSLA) project for the development of complex, distributed service level agreements.

The further paper is structured as follows: In section 2 we describe basic concepts and definitions in rule-based verification and validation research. In section 3 we adapt the concept of test cases from extreme programming to logic programming and describe a homogeneous integration approach where test cases are represented in the same rule language as the logic program. In section 4 we extend this test-driven approach with test coverage measurement for LP-based test case by exploiting techniques from inductive logic programming (ILP). We show how the feedback derived by the test coverage measurement can be used to refine and optimize the test cases and the tested programs. In section 5 we elaborate on the use of test cases in the context of rule interchange. Here, evaluation of the possibly unknown target inference engine and in particular of its semantics is vital to ensure correct execution and establish trust to this inference service. We show that test cases can be used to solve this task. Moreover, we introduce a method to determine the semantics of inference engines by testing general semantic properties with meta test programs, in case the inference

engine gives no meta information about its features and semantics. In section 6 we elaborate on another important application domain of test case, that is knowledge self-updates which evolve the state of the LP. We show how integrity constraints or test cases (as highly expressive integrity constraints) can be used to safeguard transactional knowledge updates of rules such as derivation rule or reactive ECA rules. In section 7 we describe the implementation of test cases in the ContractLog KR. The implementation integrates the test-driven development approach for LPs into IDEs such as Eclipse and common test frameworks and tools such as JUnit and Ant. Beside an ISO Prolog related scripting syntax to write LPs and test cases we have also implemented a markup serialization syntax as an extension to RuleML, a well-known Semantic Web rule markup language. In section 8 we discuss related works. In section 9 we give a conclusion of the work and discuss future steps.

2 Basic in Rule-based Verification and Validation Research

Verification and validation (V&V) of rule bases is vital to assure that the LP used, e.g. to represent a policy or a contract, performs the tasks which it was designed for. Accordingly, the term V&V is used as a rough synonym for "evaluation and testing". Both processes guarantee that the LP provides the intended answer, but also imply other goals such as to assure the security or maintenance and service of the rule-based system. There are many definitions of verification and validation in the SE literature, e.g., [76, 77]. In the context of V&V of rule bases we use the following definitions:

Verification ensures the technical correctness of a LP. Akin to traditional software engineering 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.

As discussed by Gonzales [78] *validation* should not be confused with verification. Validation is concerned with the logical correctness of a rule-based system in a particular environment/situation and domain. Typically, validation is based on tests, desirably in the real environment and under real circumstances, where the rule base is considered as a "black box" which produces certain outputs (answer to queries) given a set of input data (assertions represented as facts).

Another distinction which can be made is between errors and anomalies:

Errors represent problems which directly effect the operations of a rule base. 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 via adding, changing or refining the knowl-

edge which might easily lead to incompleteness and contradictions.

Anomalies are considered as symptoms of genuine errors, i.e. they man 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 [79] or Ayel and Laurent [65]. A general distinction can be made between errors/anomalies concerned with the design of rule bases and those concerned with the inferences. Typical inference errors are e.g. redundant rules, circular rules or dead end rules (in forward chaining systems). Typical design errors/anomalies are e.g. duplication, inconsistency or subsumedness. For a detailed discussion of potential errors and anomalies that may occur in rule bases see e.g. [80]. Here, we briefly review the notions that are commonly used in the literature:

- *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, e.g. $p \wedge q \rightarrow \neg p$
  self-contradicting rule chains, e.g. $p \wedge q \rightarrow r$ and $r \rightarrow \neg p$
  contradicting rules, e.g. $p \wedge q \rightarrow s$ and $p \wedge q \rightarrow \neg s$
  contradicting rule chains, e.g. $p \rightarrow q$ and $q \rightarrow s$ and $p \rightarrow \neg s$
  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.

- *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 via 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.

- *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 knowledge base 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 $p \rightarrow r$. Clearly, the first rule is redundant.
Self-contradicting rules, such as $p \wedge q \wedge \neg p \rightarrow r$ or simply $p \rightarrow \neg p$, which can never succeed (as already discussed in the consistency properties).
Loops in rules of rule chains, e.g. $p \wedge q \rightarrow q$ or tautologies such as $p \rightarrow p$.

A similar classification of logic and design errors/anomalies can be attributed to reactive rules (e.g. ECA rules, triggers) found e.g. in active databases, e.g. loops in rule chains have been also identified as a problem for active rules which use actions as input events to trigger other reactive rules [81, 82].

3 Homogeneous Integration of Test Cases into Logic Programs

The relevance of verification and validation of LPs has been recognized in the past and several approaches have been propose (see section 8). Most of these approaches rely on debugging the derivation trees or transforming the program into other 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 (see section 2) is then given in the language used for analyzing the program, i.e. the V&V information is not expressed in the same logical rule language as the LP. This is a strong contrast to the way people would like to engineer, manage and maintain rule-based programs and systems. Different skills for writing LPs and analyzing them are needed as well as different systems for reasoning with rules and verifying and validation rules. In many V&V approaches the used methodologies are much more complicated than the rules / programs, which should be tested (cf. e.g. model checking techniques in SE). However, 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 with respect to verification and validation of the rules. Therefore, what we would like to have is an approach that allows us to represent rules and tests in the same homogeneous representation language, so that they can be engineered, executed, maintained and interchanged together using the same inference system for reasoning about both. This design goal is

also backed up by the success of agile test-driven development in SE where test cases are written in the target programming language, e.g. Java test cases for Java programs. Before we elaborate on a homogeneous integration approach of LPs and test cases we first briefly introduce some basic concepts of logic programs, in particular general logic programs (GLP).

Syntax of General Logic Programs

A GLP over an alphabet L is a set of rules of the from $H \leftarrow B$, where H is a literal over L called the head of the derivation rule, and B is a set of literals over L called the body of the rule. A literal is either an atom or the negation *not* of an atom. A rule is called a fact if it only consists of the head H. It is called a definite rule (a.k.a. Horn rule) if it does not contain negations. An atom B 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 (possibly free) variable X or a n-ary complex term/function $f(...)$. A term is ground, if no variables occur in it and an atom is ground, if all terms are ground. Throughout this paper we use a notation for rules as described above with head and body separated by $\leftarrow$. However, our reference implementation uses a Prolog-related scripting syntax where LPs a written as scripts - see section 7. We sometimes also use this scripting notation, in particular in section 7, where we elaborate on the implementation of the introduced concepts.

Semantics of General Logic Programs

A Herbrand interpretation of a GLP is any subset $I \subseteq B_p$ of the Herbrand base of the program P. The Herbrand universe U_P is the set of all ground terms which can be formed by the functions and constants. The Herbrand base B_P is the set of all ground atoms B_i that can be formed by using the predicates and the terms from the Herbrand universe U_P. A Herbrand model of P is a Herbrand interpretation of P such that for each rule $H \leftarrow B_1, ..., B_n$ in P the interpretation satisfies the logical formula $\forall X((B_1 \wedge ... \wedge B_n \Rightarrow H))$, where X is a list of variables in the rule. A ground atom B resp. $\neg B$ is a consequence of P ($P \models B$ resp. $P \models \neg B$) if $B \in T_P^\infty$ resp. $B \ni T_P^\infty$, where T_P^∞ is a least fixpoint (according to Knaster-Tarski theorem) of the generalized operator $T_p : 2^{B_P} \mapsto 2^{B_P}$ for infinite sets of clauses, where 2^{B_P} denotes the set of all Herbrand interpretations of P. The semantics of a set P of ground clauses is defined as the set $M(P) = \{B|P \models B\} \cup \{\neg B|P \models \neg B\} = T_P^\infty \cup \{\neg B|B \in B_P \setminus T_P^\infty\}$. Note, that in practice different strategies, e.g. SLD-resolution and variants, to determine $P \models B$ have been propose which not always need to compute a ground program P and $M(P)$. In section 5 we will give a more general definition of semantics for LPs, which also incorporates non-monotonicity.

Syntax and Semantics of Test Cases for LPs

The general idea of test cases in software engineering is to predefine the intended output of a program or method and compare the intended results with the actual results of the program/method. If both match each other, the test case is said to capture the intended behavior of the program/method or alternatively, the test case covers the program/method. Although there is no 100% guarantee that the test cases defined to verify or validate 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 test cases and the program are refined in an iterative process until their is large evidence that the program satisfies the goals it was designed for. In logic programming we think of a LP as formalizing our knowledge about the world and how the world behaves. The world is defined by a set of models. The rules in the LP constrain the set of possible models to the set of models which satisfy the rules w.r.t the current knowledge base (knowledge state), i.e. the satisfiable sentences which have at least one possible model. In other words, rules provide an unambiguous way to describe the intended models of the LP by excluding models. A query Q to the LP is typically a conjunction of atoms $G_1 \wedge .. \wedge G_n$, where the atoms G_i may contain variables. Note, that, as we will see in section 5, depending on the class of the LP also disjunctive and negated queries might be possible. Asking a query Q to the LP then means asking for all possible substitutions θ of the variables in Q such that $Q\theta$ logically follows from the LP P and $P \models Q$, i.e. asking a query means finding models which satisfy the sentence formed by the query. The substitution set θ is said to be the answer to the query, i.e. it is the output of the program P. Hence, following the idea of test cases, to verify and validate the rules of a program P we need to predefine the intended outputs of P for a set of (test) queries to P and compare it with the actual results / answers derived from P by asking the test queries to P. In general the set of possible models of a program might be quite large (even if many constraining rules exist, e.g. because of a large fact base) or even infinite (e.g. in case of infinite functions in LPs). As a result the set of test queries needed to test the program and verify the actual models of P would be in worst case also infinite. However, we claim that most of the time correctness of a set of rules can be assured by testing a much smaller subset of these models. In particular, as we will see in section 4, in order to be a cover for a program the test queries need to be only an as general as possible instantiation of the rules' terms in order to fully investigate and test all rules in P. This also supports our second claim, that V&V of LPs with test cases can be almost ever done in reasonable time, due to the fact that the typical test query is a ground query which has a small search space (as compared to queries with free variables) and only proves existence of at least one model satisfying it.

In analogy to test cases in SE we define a test case $TC := \{A, T\}$ for a GLP P to consists of

- a set of possibly empty input *assertions* "A" being the set of temporarily asserted test input facts (and additionally meta test rules) defined over the alphabet "L". The assertions are used to setup the test environment. They can be e.g. used to define test facts, result values of (external) functions called by procedural attachments, events and actions for testing reactive rules or additional meta test rules

- a set of one ore more *tests* T. Each test T_i, $i > 0$ consists of
  - a *test query* Q with goal atoms of the form $q(t_1, ..t_n)?$, where $Q \in rule(P)$ and $rule(P)$ is the set of predicates in the head of rules (since only rules need to be tested)
  - a *result* R being either positive "$true$", negative "$false$" or "$unknown$".
  - an answer set θ of expected variable bindings for the variables of the test query Q: $\theta := \{X_1, ..X_n\}$ where each X_i is a set of variable bindings $\{X_i/a_1, X_i/a_2, X_i/a_3, .., X_i/a_n\}$. For ground test queries $\theta := \emptyset$.

We write a test case T as follows: $T = A \cup \{Q => R : \theta\}$. If a test case has no assertions we simply write $T = \{Q => R : \theta\}$. For example a test case $T1 = \{p(X) => true : \{X/a, X/b, X/b\}, q(Y) => false\}$ defines a test case $T1$ with two test queries $p(X)?$ and $q(Y)?$. The query $p(X)?$ should succeed and return three answers a,b and c for the variable X. The query $q(Y)$ should fail. The intended answer set might be empty either because the test query is ground and hence the test case only succeeds if the result of the query corresponds to the expected result or because the test query fails, i.e. no answer can be derived and the test case fails in case of a positive expected result. In this case we shorten the notation of a test case to $T = \{Q => R\}$. The syntax defined in this sections is only a notational syntax used in this paper to write test in a compact form. In section 8 we will introduce a Prolog related scripting syntax which is used to write LPs and test cases. In fact, test cases are written as special stand-alone LPs which can be imported to the knowledge base.

After having defined the semantics and syntax of test cases for V&V of LPs, in the next section we elaborate on improving the quality of test cases via test coverage measurement. The feedback provided by the test coverage measurement can be used to adapt the test cases to better capture the intended semantics of the LP but also to improve the quality of the LP by a refactoring of the rule base [16]. Refactoring in software engineering [17] aims at improving the structure but retaining the behavior. In section 2 we have discussed some typical design errors and anomalies which amount for a refactoring of the rule base. There are some results in the context of logic programming on refactorings in the strong sense [45]. In section 5 we will introduce some general properties a semantics for (non-monotonic) LPs should satisfy such as *Elimination of Tautologies* or *Elimination of Non-minimal Rules*. These "transformation" principles also form the basis for applying refactorings on LPs without changing their semantics.

4 Test Coverage: Measuring the Quality of Tests for Logic Programs

An important tasks in test-driven V&V is test coverage determination to evaluate the quality of the actual test cases. *Test coverage* determines whether a test case covers a program, i.e. it provides an evaluable measure for the evidence of correctness of a program by the applied test case. Test coverage gives feedback to which extend the program has been investigated by a test case and how to extend this test case with additional goals in order to increase the coverage level, i.e. test coverage also can be used to refine the test case and the rule base in the sense of *refactoring cycles*. However, differences between the declarative and imperative programming paradigm directly impact the development of a testing methodology for LPs and constrain the applicability of traditional SE test coverage measures evaluating to what extend the test cases investigate the program. Conventional SE testing methods for imperative languages, such as Java, C++, 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, e.g. [20, 43, 49, 14]. This is not directly applicable in logic programming (LP). Here the focus lies on evaluating facts and using them to infer more knowledge and find answers to problems by means of rules, whereas it is up to the rule engine (the generic solver/ inference engine) to find the solutions. In logic programming the procedural semantics is typically based on resolution with backtracking and unification (e.g. SLD(NF) resolution or SLG resolution), i.e. no explicit control flow exists and the data flow due to the declarative, global rules as well as the central concept of unification is completely different from an imperative program (it is expressed only implicitly). Hence, no conventional coverage measures from structural software testing can be used and an appropriate abstract model of a logic program for building a coverage measure has to be defined, which should be based on execution model of LPs.

Since 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, a test coverage notion for LPs can be build on this central concept of unification. In logic programming unification is used to derive specific information out of general rules which assert general information about a problem. The rules are instantiated via goals, leading to specific instances of these rules. A goal $G?$ initiates a refutation attempt unifying the goal $G?$ with the head of an appropriate rule $H \leftarrow B$ leading to an instance of the rule $(H \leftarrow B)'$ if there exists a substitution $\theta = \{V_1/t_1, .., V_n/t_n\}$ which assigns terms t_i to variables V_i such that $(H \leftarrow B)' = (H \leftarrow B)\theta$. Applying a substitution θ to a term, atom or rule (program clause) yields the instantiated term, atom or clause. For example, the rule $son(X, Y) : -parent(Y, X), male(X).$ and a goal $son(adrian, Y)?$ leads to the more specialized instance $son(adrian, Y) : -parent(Y, adrian), male(adrian)$. The instance body $B\theta$ is the goal reduction (sub goal) for further derivation leading to more specific instances. Repeating this process leads to an instance order $(H \leftarrow B) \geq (H \leftarrow B)' \geq$ whereas $\geq$ de-

notes the relation "more general as". The unification algorithm finds the greatest lower bounds (glb) of terms under this instance order $\geq$, i.e. if θ is a most general unifier (mgu) for a set of terms T then $T\theta$ is the glb of T. Since we use goals to test a logic program and unification determines the rules to be selected in the derivation of these goals an initial definition of test coverage is:

A test covers a logic program P, if the test queries (goals) lead to an as general as possible instantiation of each rule in P, such that the full scope of terms and in particular variable instantiations, of each rule is investigated by the test queries.

Clearly, such a coverage notion must take into account the concept of unification which instantiates the variables in the rule heads with the test queries. The unification should lead to a full instantiation of the terms in each rule so that they are completely investigated by the tests. In other words if the complete information defined in the LP by the rules is accessed by the test queries or the subsequent (body instance) subgoals, which are derived via instantiating the rules with the test queries, the test case covers the LP. Finding general information from specific goals is a task solved by anti-unification. In contrast to unification which finds the mgu, anti-unification can be used to find the least general generalizations (lgg) of a set of terms/clauses, i.e. the lgg describe the general information a set of instances has in common. Accordingly, the abstract model of a LP to base a coverage measure on might be build on the ordering of the instances build by the unification process starting with the test goals and their subsequent goal reductions (sub-goals), as described above. The lggs of all instances on each level under this instance order then gives the test coverage. The idea is to apply the test goals on the LP, which leads to a specialization of the rules, i.e. an instance order of rule specializations via unification. Then, based on the concept of anti-unification, we try to reconstruct the original LP via generalizing all successful specializations on each level, starting with the most specific instances at the bottom level up to the top level instances. The resulting "*level of reconstruction*" then gives a measure for the test coverage. This measure determines how much of the rule' terms has been tested by the test case. In particular this means, if the generalizations lead to a complete reconstruction of the original LP the test case covers the program. For example consider a program P with following rules:

$father(Y, X) : -son(X, Y), male(Y).$
$son(X, Y) : -parent(Y, X), male(X), male(Y).$
$son(X, Y) : -parent(Y, X), male(X), female(Y).$

and the following facts:

$male(adrian).male(uwe).male(hans).$
$female(hariet).female(babara).$
$parent(uwe, adrian).parent(hariet, adrian).parent(hans, uwe).parent(babara, uwe).$

Let $T = \{father(uwe, adrian)? => true, father(hans, uwe)? => true, son(hans, uwe)?)\}$ be a test case with two test goals.
The set of top level instances are

$father(uwe, adrian) : -son(adrian, uwe), male(uwe).$
$father(hans, uwe) : -son(uwe, hans), male(hans).$
$son(uwe, hans) : -parent(hans, uwe), male(uwe), male(hans).$

The subsequent second level instances are

$son(adrian, uwe) : -parent(uwe, adrian), male(adrian), male(uwe).$
$son(uwe, hans) : -parent(hans, uwe), male(uwe), male(hans)$

The lggs of the second level instances are

$son(X, Y) : -parent(Y, X), male(X), male(Y).$

and the lggs of the first level instances are

$father(Y, X) : -son(X, Y), male(Y)$

Accordingly, the first and the second program clause are covered, while the third is not, i.e. the overall coverage is 67%. We can see that the second rule is already covered by the first two goals in the test case, which test the father relation. Hence, we can remove the goal "$son(hans, uwe)?$" from the test case. We further can see that we are missing some level of generality in our LP, because the third rule is not tested at all. This give us a clue how to extend either the test case with additional goals so that the third program clause is tested or extend the LP with an additional rule describing the missing relation between a female parent and the son of this parent. We extend P with the following rule

$mother(Y, X) : -son(X, Y), female(Y).$

and add two additional test goals $mother(hariet, adrian)? => true$
and $mother(babara, uwe)? => true$ to T. This leads to two new top level instances

$mother(hariet, adrian) : -son(adrian, hariet), female(hariet).$
$mother(babara, uwe) : -son(uwe, babara), female(babara).$

and the subsequent second level instances

$son(adrian, hariet) : -parent(hariet, adrian), male(adrian), female(hariet).$
$son(uwe, babara) : -parent(babara, uwe), male(uwe), female(babara).$

The lggs are then

$mother(Y, X) : -son(X, Y), female(Y).$
$son(X, Y) : -parent(Y, X), male(X), female(Y).$

The test case now covers the "refined" LP, i.e. coverage = 100%. Before we further elaborate on this approach to test coverage, we first give an insight into the basic concepts of anti-unification.

Anti-Unification
The basic concept of anti-unification was introduced by Plotkin [41, 42] and further explained e.g. in [47, 30]. Anti-unification has been successfully used in theorem proving [9] and inductive logic programming (ILP) [29, 8, 1, 38]. Recalling the instance ordering described above unification and anti-unification have a dual operation of specialization and generalization. Figure 1 illustrates this.

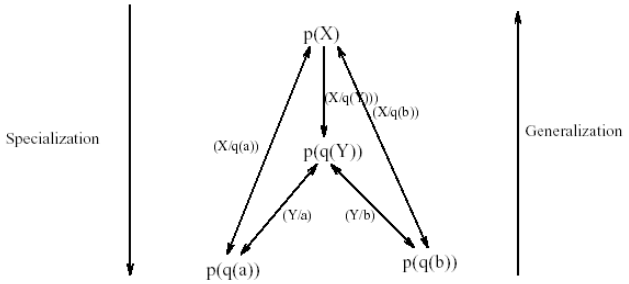

Fig. 1. Unification vs. Anti-Unification

Whereas unification finds the most general unifier (mgu) of two terms and leads to the greatest lower bounds (glb), anti-unification will find the least general generalization (lgg) of two terms and leads to the least upper bounds. In other words anti-unification provides the mathematical formalization of the concept of generalization. This generalization means finding the lgg that expresses what two instance terms have in common. This is important, because obviously terms can be generalized in many ways, however our interest is in finding the least general generalization of specialized instances. The lgg is the generalization that keeps a anti-unified term t as special as possible so that every other generalization would increase the number of possible instances of t in compari-

son to the possible instances of the lgg. Different extensions to Plotkin's purely syntactical anti-unification [41, 42] have been proposed such as semantic generalizations which additionally take background knowledge into account [39], anti-unification with respect to equational theories [44], high-order anti-unification [40] and different definitions have been proposed for the "more-general-as" relation $\geq$ which imposes an instance ordering on the space of program clauses (rules), e.g θ-subsumption, generalized subsumption, relative subsumption, inverse resolution [32]. However, in practice the large majority of anti-unification algorithms used in ILP systems such as FOIL, CLAUDIEN, TILDE, use syntactical anti-unification with θ-subsumption and least general generalization. This is due to syntactic generalization does not take into account any background knowledge. The implementation is therefore much easier and is computationally more feasible as compared to semantic generality (relative least general generalization), inverse resolution or inverse implication, which are in general computationally intractable and undecidable. Because of this excellent computational properties θ-subsumption and lgg are the right framework for generality to base our test coverage notion on. In the following we give a brief review of θ-subsumption and the concept of least general generalization.

Recall the instance ordering described above: A term t' is an instance of a term t if there exists a substitution θ such that $t' = t\theta$, whereas t is more general than t' denoted by $t \geq t' \Leftrightarrow \exists\theta : t\theta \rightarrow t'$. θ-subsumption introduces a syntactic notion of generality: A rule 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 *at 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. [41]). The anti-unification theorem given in [30] states that a set of terms extended by a least element $\perp$ under the instance order relation, forms a lattice, whereas there exists a least upper bound (lub) and a greatest lower bound (glb). The most general unifier (mgu) is the glb and the least general generalization (lgg) is the lub: Let S be a set of terms $t''_1, .., t''_n$. The mgu of S corresponds to the glb of S under $\geq$, i.e. if the subsumption θ is an mgu for S then $S\theta$ is the glb of S. The lgg of S is the term t with $t \geq t''_i (1 \geq i \geq n)$ and for every term t' with $t' \geq t''_i$ it is also true that $t \geq t'$. For each t''_i there exists a substitution θ_i to transform the lgg back into t''_i. A term t θ-subsumes a term t' iff there exists a substitution θ, such that $\exists\theta : t\theta \leq t'$, i.e. θ-subsumption defines a syntactic notion of generality and specialization:

- If a least element $\perp$ is added to the instance order, the relation $\leq$ forms a lattice on the set of reduced clauses with a partial ordering (reflexive and transitive)
- If t θ-subsumes t' then t is at least as general as t': $t \leq t'$.
- If $t \leq t'$ and $t' > t$ then t' is a specialization of t and t is a generalization of t'.
- If $t \leq t'$ and $t' \leq t$ then t is a variant of t'.
- If t θ-subsumes t' then t logically entails t': $t \models t'$.

For example a rule
$R_1 = father(Y, X) : -son(X, Y), male(X)$

is more general than a rule
$R_2 = father(uwe, adrian) : -son(adrian, uwe), son(uwe, babara), male(adrian), female(babara).$
because $R_1\theta \leq R_2$ with $\theta = \{X/adrian, Y/uwe\}$. A rule $p() : -q(X,Y), q(X,Y)$ is more general than $p() : -q(a,a)$ under θ-subsumption as well as $p() : -q(X)$ is more general than $p() : -q(f(a))$. Two rules $p() : -q(X,Y).$ and $p() : -q(X,Y), q(X,Z).$ are variants under θ-subsumption.

The lgg of two terms t and t' denoted by $lgg(c,c')$ is the lub of t and t' under the θ-subsumed instance order. Plotkin has given a first algorithm to compute the lgg:

- $lgg(t,t) = t$: the lgg of two identical terms is the term itself
- $lgg(f(a_1,..,a_n)), f(b_1,..b_n)) =$
  $f(lgg(a_1,b_1),..,lgg(a_n,b_n))$: the lgg of the terms $f(a_1,..,a_n))$ and $f(b_1,..b_n)$ is $f(lgg(a_1,b_1),..,lgg(a_n,b_n))$
- $lgg(f(a_1,..,a_n), g(b_1,..,b_n)) = X$ where $f \neq g$ and X represents the pair of terms throughout
- $lgg(p(s_1,..,s_n), p(t_1,..,t_n)) =$
  $p(lgg(s_1,t_1),..lgg(s_n,t_n))$: The lgg of two literals $p(s_1,..,s_n)$ and $p(t_1,..,t_n)$ is $p(lgg(s_1,t_1),..lgg(s_n,t_n))$
- The lgg being undefined when the sign or the predicate symbols are different

This holds for lggs of atoms and clauses (rules) alike. The lgg of two atoms $lgg(A_1, A_2)$ is computed as follows:

- $lgg(p(s_1,..,s_n), p(t_1,..,t_n)) =$
  $p(lgg(s_1,t_1),..lgg(s_n,t_n))$, if atoms have the same predicate symbol p,
- $lgg(p(s_1,..,s_m), q(t_1,..t_n))$ is undefined if $p \neq q$

The lgg of two literals $lgg(L_1, L_2)$ is defined as follows:

- if L_1 and L_2 are atoms, then $lgg(L_1, L_2)$ is computed similar to atoms.
- if both L_1 and L_2 are negative literals then $lgg(L_1, L_2) = \overline{lgg(L_1, L_2)}$

Finally, the lgg of two clauses C_1 and C_2 is then

- $\bigvee_{L_1 \in set(C_1), L_2 \in set(C_2)} lgg(L_1, L_2)$

  Examples:

$lgg(parent(uwe, adrian), parent(hariet, adrian)) = parent(X, adrian)$
$lgg(parent(uwe, adrian), \neg parent(hariet, adrian))$ is undefined
$lgg(parent(uwe, X), son(adrian, uwe))$ is undefined.

Accordingly, anti-unification works as follows: It takes two terms as input and the algorithm returns a list containing the lgg of the terms and the bindings to transform each input-term into the lgg. The algorithm can fail only in the case the top-symbols or the length of the two terms are different (because otherwise it might be possible to get just a new variable when anti-unifying the clauses /

terms. There are efficient algorithms for the computation of the (relative or partial) lgg including Prolog related implementations and several implementations have been proposed in ILP systems such as GOLEM. In our reference implementation in the ContractLog KR we have implemented a meta logic program to compute (relative) lggs. The following example will illustrate the basic algorithm:

Input terms:
$f(a, g(b, h(X)), c)$
$f(d, g(j(X), a), c)$

1. Take the first subterm out off the input terms, anti-unify them and add the adequate bindings to the list of bindings:
   $\theta_1 = ((a/X))$
   $\theta_2 = ((d/X))$
2. Repeat this step until every subterm has been anti-unified. If necessary apply the step recursively on the subterms.
3. Replace every subterm in the input terms by its computed binding. Take care that every used substitution is compatible with the substitutions in the second input term:
   $\theta_1 = ((a/X)(b/Y)(h(X)/Z))$
   $\theta_2 = ((d/X)(j(X)/Y)(a/Z))$

The result after anti-unification is then the $lgg = f(X, g(Y, Z), c)$.

Test Coverage
Using the concepts of θ-subsumption and least general generalization we now refine our initial test coverage notion. As already discussed a set of test goals covers a LP if the goals lead to a full unification, i.e. an instantiation of every rule in the program in such a way that the instantiation of the arguments of the rule terms is as general as possible. In order to determine and measure the level of coverage, the instance order under θ-subsumption ordering of the LP is computed via specializing the program clauses with the test goals by unification. Then via generalizing this specialization, i.e. computing the lggs of all successful instances on each level via anti-unification, a reconstruction of the original LP is tried. This gives the level of test coverage, which can be interpreted as how much of the general information given by the LP rules is actually tested by the goals in the test case. In particular, if the complete LP can be reconstructed via generalization of the specialization then the test case fully covers the LP. Formally we express this as follows:

Let T be a test with a set of test queries $T := \{Q_1?, .., Q_n?\}$ 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 an equivalence relation and r'_i is the specialization (instance) of r_i, i.e. the instantiation of r_i by a query $Q_i \in T$. It is a cover for a set of rules with head h: $S := \{r^h_1, .., r^h_m\}$, if T or the sets of subgoals $\{T', T'', .., T^k\}$ under the instance ordering derived by instantiating P with T are a cover for each

rule $r_i^h \in S$ and it is a cover for a program P, if T is a cover for each $S_j \in P$. With this definition it can be determined whether a test covers a LP or not. The coverage measure for P is then given by the number of cover rules r_i divided by the number k of all rules in P, i.e. the relative number of rules covered by T is measured:

$$cover_P(T) : -\frac{\sum_{i=1}^{k} cover_{r_i}(T)}{k}$$

For example consider a LP with the following clauses:

$tail([]).$
$tail([A|B]) : -tail(B).$

and a test case with the following test queries:

$$\{tail([])? => true, tail([a])? => true, tail([a|b])? => true\}$$

This leads to the following instantiations:
$tail([]).$
$tail([a]) : -tail([]).$
$tail([a,b]) : -tail([b]).$

The lggs are then:

$tail([]).$
$tail([a|B]) : -tail([B]).$

Therefore, $tail([])$ is covered but not $tail([A|B]) : -tail(B)$, which gives a coverage of 50%. Adding the test case $tail([b,a])?$ to the test set leads to the additional instantiation:

$$tail([b,a]) : -tail([a]).$$

and the additional lgg:

$$tail([A|B]) : -tail([B]).$$

Thus, the test case now covers the program.

The coverage measure determines how much of the general information expressed by the rules in the program are already covered by the actual test case. Furthermore, the lggs give feedback how to extend the set of goals in order to increase the coverage level. For example given the lgg "$tail([a|B]) : -tail([B])$." it can be concluded that any additional test goal, where the head of the list is not a (e.g. $tail([b,a])?$) will lead to a coverage of this clause. Therefore, the coverage notion described in this article might also form the basis for test case

generation and further optimization of the test case and/or the logic program. For example by applying refactorings the most specific version P' of a LP P can be created, which has the same success set (has the same semantics) as the more general P but leads to a increased set of finitely failed goals, i.e. failed derivation trees can be detected more quickly and P' is more efficient than P, see e.g. [36]. To give an example consider the two rules:

$son(X,Y) : -male(X), male(Y), parent(Y,X).$
$son(X,Y) : -male(X), female(Y), parent(Y,X).$

These rules are more specific than the rule

$son(X,Y) : -male(X), parent(Y,X).$

and can be computed much more efficiently.

The used θ-subsumption has some nice computational properties and it works for simple terms as well as for complex terms, e.g. $p() : -q(f(a))$ is a specialization of $p : -q(X)$. θ-subsumption and least general generalization are purely syntactic notions. Their computation is therefore simple, as compared to inverse resolution or inverse implication, which are both computationally intractable. So θ-subsumption and least general generalization qualify to be the right framework of generality in the application of our test coverage notion. In the next section we will elaborate on the application of test cases to safeguard rule interchange and on the use of meta test programs to verify and validate inference engines against general properties, which any reasonable semantics used in logic programming should satisfy.

5 V&V of Rule Engines and Safeguarding Rule Interchange via Test Cases

A strong demand for rule-based functionalities comes from the web community, in particular in the area of Semantic Web, e.g. hybrid resp. homogeneous integration of rules and ontologies (see e.g. [83] or SWLR http://www.w3.org/Submission /SWRL/), Rule Markup Languages (e.g. RuleML http://www.ruleml.org) or the forthcoming Rule Interchange Format (RIF http://www.w3.org/2005/rules/). Here rules are managed and maintained in a distributed environment and interchanged over domain and system boundaries using more or less standardized rule markup interchange formats, e.g. serialized in XML/RDF. For example, rule-based policy or contract representation projects typically amount for a distributed management of contract/policy specifications with alternative, default, exceptional and hierarchical rule sets resp. modules (see e.g. the RBSLA/ContractLog KR http://ibis.in.tum.de/staff/paschke/rbsla/). The interchanged rule bases need to be interpreted and correctly executed in the target environment, i.e. in a target rule engine (a LP inference engine), which might be

project/application specific or provided as an open (web) service by a third-party provider or a standardization body such as OMG or W3C (see [84]). Clearly, the correct execution of the interchanged LP depends on the semantics of both, the LP and the the inference engine (under the assumption that the LP is specified correctly and the implementation of the inference engine is correct). For example. a more expressive generalized or extended LP with e.g. default negation or explicit negation can not be executed properly by an inference engine which implements only pure SLD-resolution without negation and which supports only definite LPs; or an expressive LP might be too less efficient in terms of efficient query-answering for the application domain it was intended for, when interpreted in a certain semantics, e.g. applying stable model semantics, while it behaves computationally adequate when another semantics is used, e.g. well founded semantics with linear worst case complexity. Test cases, which are interchanged together with the rule bases, can be used to test whether a particular LP still has the same intended behavior if it is executed in the target environment, i.e. provides the intended answers and has adequate computational properties. Moreover, standard meta tests verifying typical semantical properties of KR semantics (LP semantics and/or non-monotonic semantics) can be used to tests the semantics implemented by a target inference engine and establish trust to this engine. Different meta test case variants for V&V for verifying and validating well-known semantics for typical LP classes can be provided, either published in public repositories or directly attached to the interchanged programs. In the following we will further elaborate on the use of test cases in rule interchange and in V&V of inference engines.

To address this issues the inference engine, the interchanged LP and the provided test cases must reveal their (intended resp. implemented) semantics. In the easiest way, this can be solved with explicit meta annotations based on a common vocabulary, e.g. an ontology which classifies typical semantics such as *COMP*, *STABLE*, *WFS* and relates them to typical classes of LPs such as *positive definite LPs*, *strafied LPs*, *normal LPs*, *extended LPs*, *disjunctive LPs*. The ontology can then be used to describe additional meta information about the semantics and logic class (or rule language) of the interchanged rules and test cases and find appropriate inference engines to correctly and efficiently interpret and execute the LP, e.g.

- by configuring the rule engine for a particular semantics in case it supports different ones (see e.g. the configurable ContractLog inference engine)
- by executing an applicable variant of several interchanged semantic alternatives of the LP
- by automatic transformation approaches which transform the interchange LP into an executable LP in the inference engine using predefined transformation rules.

However, we do not believe that each rule engine vendor or rule base / test case programmer will annotate its implementation with such meta information, even when there is an official standard vocabulary on hand (e.g. released by OMG or W3C). Therefore, means to automatically determine the supported semantics of

inference engines or logic programs is needed. Test cases can be extended to meta test programs testing typical properties of well-known semantics. A meta test program is a test case with additional meta rules as assertions. These meta rules can be used to test the inferences and semantic properties of the rule environment via running appropriate test queries against the meta rule set and comparing the answers with the expected results. By applying such meta test programs (which might be bundled to test suites) to an inference engine the semantics provided by the engine can be determined by the set of satisfied and failed tests. For example, a property a semantics SEM might have is that its semantics should not be changed if a tautology such as $p \rightarrow p$ is eliminated from the rule base, which is e.g. not support by Clark's completion semantics (COMP). Hence, if the meta test program for this property fails, the set of all possible semantics is constrained to semantics which do not support this property, e.g COMP. In fact, the closely related standard SLD(NF) resolution implemented in common Prolog interpreters will lead to an loop for such rules. In the second part of this section we will introduce a first taxonomy of semantics for different classes of LPs and general properties which these semantics should fulfil.

Semantics for LPs (LP-semantics) have been considered in the logic programming community since the early 70's and led to the definition and implementation of a great variety of semantics, e.g. Clark completion semantics (COMP) and variants such as 3-valued COMP_3, which are closely related to SLD-resolution and mainly rely on the idea of negation-as-finite-failure as a procedural approach to negation. In parallel, starting at about 1980, various declarative semantics for some form of non-monotonic reasoning (a.k.a. NMR-semantics) have been defined such as stable model semantics (STABLE) or well-founded semantics (WFS) [93]. Intensive work in the 90's has been done to study the close relationships between the two independent research lines and successfully apply methods from non-monotonic reasoning such as defaults, explicit negation, disjunctions, updates or priorities/preferences in logic programming or vice versa. We do not want to present an overview on the different LP- and NMR-semantics, nor discuss their merits or shortcomings. For an overview we relate to [85–87]. In general, there are three ways to determine the semantics (and hence the inference engine) to be used for an LP: by its *complexity and expressiveness class* (which are in a trade-off relation to each other), by its *runtime performance* or by the *semantic properties* it should satisfy. A generally accepted criteria as to why one semantics should be used over another does not exists, but two main competing approaches, namely WFS and STABLE, have been broadly accepted as declarative semantics for general LPs. We first discuss the complexity and runtime performance considerations.

Validating Complexity

The following theorem gives a brief review of complexity results derived for major LP classes and semantics.

Theorem:

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 LPs with functions are
(h) r.e.-complete for full LPs but NEXPTIME complete for non-recursive LPs
(i) Π_1^1-complete for full LPs under WFS and SMS

Proof: (a) [90], (b) [91], (c) [92], (d) [93, 94], (e) [92] and [95], (f) [93, 94], (g) [100], (h) [96, 97] , (i) [95]

For a more detailed discussion of the complexity and expressiveness of several classes of LPs and well-known semantics we refer to [89, 88]. Based on these worst-case complexity results for different semantics and expressive classes of LPs, which might be published in a machine interpretable format for automatic decision making, certain semantics might be already excluded to be usable for a particular rule-based application resp. LP. For example non-recursive LPs with functions which are in *NEXPTIME* are obviously not adequate in an application domain which needs real-time query answering from large rule bases. Furthermore, the complexity classes might be used to find more efficient (and complete) inference engines on the web, e.g. if we want to execute a LP with Datalog restriction we can search for a inference engine supporting WFS with polynomial time complexity for general Datalog LPs.

Validating Runtime Performance

However, asymptotic worst-case results are not always appropriate to quantify performance and scalability of a particular rule execution environment. Implementation specifics of an inference engine such as the use of recursions or inefficient memory-structures might lead to low performance or memory overflows in practice. Therefore, it might be important to provide performance test cases which can be used to measure the execution performance and scalability of the target inference engine. Up to now we have only used test cases to validate the

correctness of a LP, i.e a rule set is considered valid if it correctly solves all test cases, and there is evidence that it will correctly solve any further added test case. However, while this is a required property of a high-quality rule set, it is rarely a sufficient one (except in very small domains). In most cases inference engines in particular with very expressive formal rule languages are facing high computational tasks, where the run-time efficiency is essential for the acceptance. Here another validation dimension must be added, namely the measuring of the maximum efficiency of the computation process. It is essential to equip a validation tool with an automated procedure, which can recognize problematic relations between rules. Test cases can be used to measure the runtime performance for different outcomes of a rule set given a certain test fact base as input. By this certain points of attention, e.g., test case with long computations, loops or deeply nested derivation trees, can be identified and a refactoring of the rule base optimizing the rule sequences (e.g. by reordering rules, narrowing rules, deleting rules etc.) according to these attention points can be attempted, in order to reach the maximum efficiency of the computation process. We call this *dynamic validation* in opposite to *functional validation* of LPs. Dynamic validation can be best applied after the rule base or the inference engine which should be tested has undergone the functional testing and all detected anomalies have been removed from the knowledge base/inference engine. Dynamic test cases with maximum time values (time constraints) for answering certain queries can be defined to assure that a rule set solves its intended tasks in an acceptable time frame, i.e.,that with any further added rule the computational time stays below this limit. If the test case fails it becomes an attention point for a refactoring of the rule base or a reimplemention of the inference engine. We define a dynamic test case as an extension to a functional test case (see section 3): $TC = A \cup \{Q => R : \theta < MS\}$, where MS is a maximum time constraint for the test query Q. If the query was not successful within this time frame the test is said to be failed. For example, consider the dynamic test case $TC_{dyn} : q(a)? => true < 1000ms$. The test succeeds iff the test query succeeds and the answer is computed in less than 1000 milliseconds. To do dynamic validation with test cases we have one prerequisite: That the tested fact base and the used test cases must be an adequate benchmark for the later run-time environment and that we can predict how well a rule set would do on a different compiler, computer or data set. Although this might be unsatisfactory because dynamic test cases are therefore very domain specific or because we do not want to give a rule set a special order we argue that in many applications where higher-level rules are applied we need at least a clue if our rule set is able to solve its purpose in an adequate time and we need methods to do this benchmarks already in the development phase of LPs. Test cases constraining the computation time might help to carry out this estimation. Furthermore, when the rule set went into action we can adjust the upper computational limits to the particular application domain which must not be exceeded even if new rules are added to model to assure that the rule set fits into the application.

In the context of rule interchange and runtime performance validation of infer-

ence engines by using dynamic test cases we need to extend them to dynamic meta test programs, which benchmark performance of typical inference processes of the engine. The test LPs provided as assertions of these dynamic meta test cases can be evaluated in several repeated experiments in the target inference engine. The resulting average performance values are then compared with the predefined upper limits for the tests.If they are below the maximum valued, execution performance of the inference engine is adequate otherwise it is unsatisfying and might lead to problems when querying the interchanged LP in the target inference engine. An important question is: " *What are appropriate performance tests to validate an arbitrary inference engine*?"
Validation has two dimension: one is performance of query answering, i.e. how long will it take to answer a certain query given a LP of a certain complexity and size, and the other is scalability, i.e. what it the maximum size of a LP which allows answering certain queries. Clearly, both evaluation dimensions are closely related and not orthogonal. Scalability is influenced in general by the used inference algorithms and by the particular implementation of the inference engine. For example, insufficient memory structures might lead to memory overflows during query answering and hence to low scalability, i.e. only small LPs with small rule or fact bases can be executed. Typical factors which influence the performance of query answering for a LP are, e.g. the depth of the search tree, recursion in rules, complexity of rules such as number of body atoms, number of variables in the rule terms, negation, loops. To experimental evaluate performance and scalability of an inference engine we use *scalable performance benchmark tests* . *Scalable* here means that the test program can be varied in size and *performance* means that for each size of the program the performance of query answering is measured. We employ a test theory for testing defeasible theories [127] to test logic programs. Useful benchmark tests are:

Scalable Derivation Rule Benchmarks

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.*

$$chains(n,p,v) = \begin{cases} a_n(X_1,..,X_v) :- a_{n-1}(X_1,..,X_v) \\ \quad ... \\ a_1(X_1,..,X_v) :- a_1(X_1,..,X_v) \\ \\ a_0(x_1,..,x_v)....a_0(x_1,..,x_v).\%facts \end{cases}$$

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.*

$$loop(n,p,v) = \begin{cases} a_n(X_1,..,X_v) :- a_{n-1}(X_1,..,X_v) \\ \quad ... \\ a_1(X_1,..,X_v) :- a_0(X_1,..,X_v) \\ \\ a_0(x_1,..,x_v) :- a_n(X_1,..,X_v).\%loop \\ \\ \quad a_0(x_1,..,x_v).\%facts \end{cases}$$

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.*

$$dag(n,k,v) = \begin{cases} a_0(X_1,..,X_v) :- a_1(X_1,..,X_v), a_2(X_1,..,X_v), ..., a_k(X_1,..,X_v). \\ a_1(X_1,..,X_v) :- a_2(X_1,..,X_v), a_3(X_1,..,X_v), ..., a_{k+1}(X_1,..,X_v). \\ ... \\ a_{nk}(X_1,..,X_v) :- a_{nk+1}(X_1,..,X_v), a_{nk+2}(X_1,..,X_v), ..., a_{nk+k}(X_1,..,X_v). \\ a_{nk+1}(x_1,..,x_v).a_{nk+2}(x_1,..,x_v)....a_{nk+k}(x_1,..,x_v).\%facts \end{cases}$$

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.*

$$rules(a,n,k,v) = \begin{cases} a(X_1,..,X_v) :- a_1(X1,..,Xv), a_2(X_1,..,X_v), .., a_k(X_1,..,X_v). \\ \quad rules(a_1, n-1, k, v) \\ \quad rules(a2, n-1, k, v) \\ \quad ... \\ \quad rules(ak, n-1, k, v) \end{cases}$$

$and : rules(a,0,k) = a(X_1,..,X_v).\%facts$

Such tests can be varied in size and adapted to different LP classes, with e.g. variables, negation, disjunction, and different rules types, e.g. normal strict rules, defeasible rules, reactive rules. The results of the experiments (running the tests in a target inference engine) are used to validate runtime performance of the engine.

Validating the Semantics of Inference Engines

The correct execution of a LP in a target inference engine strongly depends on the supported semantics. Hence, it is important to reveal both the intended semantics of the interchanged LP and the semantics of the target inference engine. This can be done either by additional machine-readable meta annotations

of the engine and program, which specify the semantics and the logic class or by automated determination of the semantics via meta test programs. Note, an inference engine might implement different semantics and might be dynamically configurable depending on the program which should be executed by it (see e.g. the ContractLog inference engine which can be configured to a special SLDNF variant extended with goal memoization and loop prevention, but can be also configured to support WFS). We first give some general definitions and an initial taxonomy of semantics and LP classes which can be used as the basis for an ontology (e.g. represented in OWL and published on the web) used to meta annotate the interchanged LPs, the inference engines and the test cases. We draw on a general semantics classification theory developed by J. Dix [98, 99] and reuse the results in the context of our verification and validation approach. We then introduce so called meta test programs to validate inference engines and establish trust to these services. We further refine this idea to automatic determination of the semantics supported by an inference engine, in case no meta information are given.

Definition

- *A semantics $SEM(P)$ of a program P is proof-theoretically defined as a set of literals that are derivable from P using a particular derivation mechanisms, such as SLDNF-resolution with negation-as-finite-failure.*
- *Model-theoretically, a semantics $SEM(P)$ of a program P is a subset of all models of P: $MOD(P)$. In this paper in most cases a subset of the (3-valued) Herbrand-models of the language of L_P: $SEM(P) \subseteq MOD_{Herb^{L_P}}(P)$.*
- *A semantics SEM' extends a semantics SEM denoted by $SEM' \geq SEM$, if for all programs P and all atoms l the following holds: $SEM(P) \models l \Rightarrow SEM'(P) \models l$, i.e. all atoms derivable from SEM with respect to P are also derivable from SEM', but SEM' derives more true or false atoms than SEM.*
- *The semantics SEM' is defined for a class of programs that strictly includes the class of programs with the semantics SEM.*
  SEM' coincides with SEM for all programs of the class of programs for which SEM is defined.

Clearly, a LP which should be interpreted with a semantics SEM' can hardly be executed in an inference engine with $SEM \leq SEM'$, but an inference engine implementing SEM' can executed all LPs with $SEM \leq SEM'$ (in particular if both are in the same logic class). However, complexity of SEM' in most cases is higher then in SEM, and hence for performance reasons an inference engine supporting SEM is in most cases optimal for programs of the logic class for which SEM is defined. Figure 2 illustrate different classes of LPs and table 1 associates well-known semantics to each class of logic program.

The LP classes are ordered by expressiveness, i.e. a more expressive class P' includes a less expressive class P. For example an extended disjunctive LP is

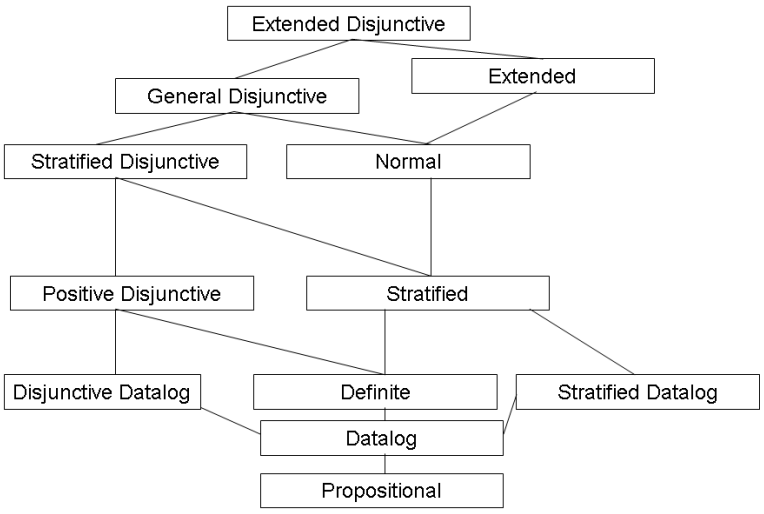

Fig. 2. Classes of LPs

Table 1. Table (Semantics for LP Classes)

Class	Semantics	Ref.
Definite LPs	Least Herbrand model: M_p	[100]
Stratified LPs	Supported Herbrand model: M_p^{supp}	[100]
Normal LPs	Clark's Completion: $COMP$	[101]
	3-valued Completion: $COMP_3$	[102, 114]
	Well-founded Semantics: WFS	[93]
	WFS^+ and $WFS^{'}$	[103]
	WFS_C	[104]
	Strong Well-founded Semantics: WFS_E	[105]
	Extended Well-founded Semantics: WFS_S	[106]
	Stable Model Semantics: $STABLE$	[107]
	Generalized WFS: $GWFS$	[113]
	$STABLE^+$	[99]
	$STABLE_C$	[104]
	$STABLE^{rel}$	[108]
	Pereira's $O-SEM$	[109]
	Partial Model Semantics: $PARTIAL$	[110]
	Regular Semantics: $REG-SEM$	[111]
	Preferred Semantics: $PREFERRED$	[112]
General Disjunctive	Disjunctive WFS: $DWFS$	[118]
	Generalized Disjunctive WFS: $GDWFS$	[120]
	Disjunctive Stable: $DSTABLE$	[119]
Stratified Disjunctive	Perfect model $PERFECT$	[115]
	Weakly Perfect: $WPERFECT$	[116]
	Generalized Closed World Assumption: $GCWA$	
Positive Disjunctive	Weak generalized closed world assumption: $WGCWA$	[117]

more expressive than a normal / general LP. It extends a normal LP with explicit negation and disjunctions. As a result the associated semantics $SEM(P')$ defined for a class of LPs P' extend the semantics $SEM(P)$ for all classes of LPs P which are "sub-classes" of P': $SEM(P') \geq SEM(P)$ if $P \subset P'$. These sub-class relations define a natural order of the extension ordering of semantics $SEM' \geq SEM$ resp $SEM' \not\leq SEM$ and $SEM' \not\geq SEM$ if none of both semantics is an extensions of the other (this is e.g. the case, if the logic class for which SEM' is defined for is not a subclass of the logic class of SEM). Figure 3 (adapted from [98, 99]) illustrates the extension ordering of semantics up to extended disjunctive LPs. Hence, given the information to which class a partic-

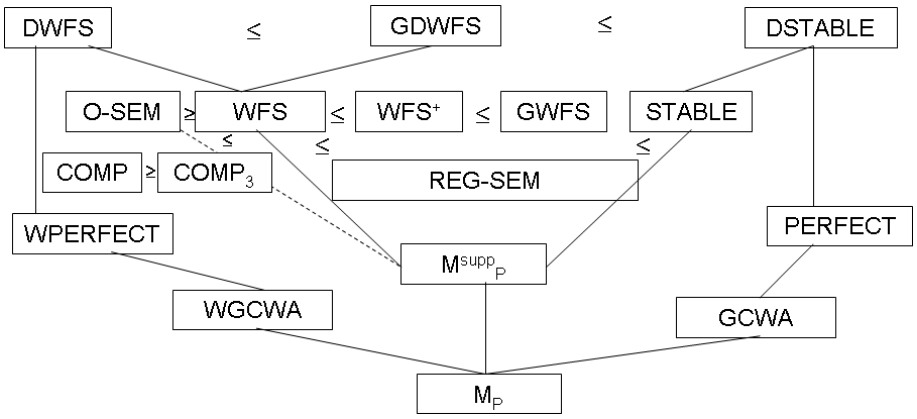

Fig. 3. Semantics for LPs and their Extension Relations

ular LP belongs or which is the intended semantics of this LP and given the information which semantics is implemented by the (target) inference engine, we can easily decide wether the LP can be executed by the inference engine or not. In short, a LP can not be executed by an inference engine, if the inference engine derives less literals than the intended SEM for which the LP was design for would do, i.e. $SEM'(InferenceEngine) \geq SEM(P)$ or the semantics implemented by the inference engine is not adequate for the program, i.e. $SEM'(InferenceEngine) \neq SEM(P)$. This information can be give by meta annotations, e.g.:
- *class:* defines the class of the LP / inference engine
- *semantics:* defines the semantics of the LP / inference engine
- *syntax:* defines the rule language syntax

In the context of rule interchange with open, distributed inference engines, which might be provided as public services, an important question is, wether the inference engine correctly implements a semantics. Test cases can be used to

verify and validate the interchanged LP in the target environment and therefore establish trust to this service. They can be interchanged together with the LP and used to validate the LP in the target inference engine. Since the results of a tests provided by the inference engine might differ according to the semantics it implements but this differences might not necessarily be errors or anomalies, test cases should be also annotated with meta information about semantics and logic class and should possibly provide different tests variants to test different semantics. For example assume a simple LP with the following rules:

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

and a test case T with one test: $\{c? => true\}$, i.e. the test query $c?$ should yield true. In case the inference engine supports stable model semantics (STABLE) this test case succeeds. But, if the inference engine supports well-founded semantics (WFS) the test query leads to the answer "unknown" and accordingly the test case fails. If the answer "unknown" in addition to "true" is also a valid answer in the application domain of the LP the test case can provide a second variant of the test $T2 : \{c? => unknown\}$ which should be applied in case of WFS.
While these test cases directly relate to the validation of a particular interchanged LP in a particular domain, *meta test cases* or more precisely *meta test programs* can be used to verify the correct implementation of an inference engine resp. the semantics of a inference engine. For example standard SLDNF resolution typically suffers from loops, whereas most non-monotonic semantics for LPs such as WFS and resolution algorithms such as SLG typically implement some form of loop checking / loop prevention . A simple meta test case for verifying loop avoidance might be e.g. the following propositional meta test program:

```
a
b
a <- b
b <- a
```

and the test $\{a? => true\}$, which should succeed in a inference engine implementing WFS, whereas it fails e.g. in an inference engine implementing COMP or using SLDNF resolution (due to the loop). Another problem of SLDNF resolution is that it can not handel free variables in negative subgoals due to the procedural negation-as-finite-failure, e.g.:

```
p(a)
q(b)
r(f(a)
p(X) <- not q(X), r(f(X))
```

In case of a test $\{p(a)? => true\}$ the test case succeeds as expected. However, a test $\{p(X)? => true : \{X/a\}\}$ with the expected variable binding $X = a$ fails,

because the negation-as-failure tree is entered with a free variable, which fails due to the fact $q(b)$, but no variable binding for X is computed. This is because SLDNF treats the subgoal as universally quantified $\forall X not\ q(X)$ instead of existentially quantified $\exists X not\ q(X)$ and only makes a simple test. This problem is also known as floundering problem. For more unsolvable problems related to SLDNF see e.g. [121]. In a similar way other well-known problems and paradoxes from literature, e.g. Yale Shooting Problem, which are solved by a particular semantics resp. resolution can be used as test cases. For example to verify the tabled not (tnot) implementation in SLG resolution, Russel's paradox might be used as a test case:

```
person(barber)
person(mayor)
shaves(barber,Person) <- person(Person), tnot(shaves(Person,Person))
```

A test case $T : \{shaves(barber, X)? => true : \{X/mayor\}\}$ should succeed and return the answer mayor.
Meta test cases of this sort can be bundled to test suites for verifying certain resolution implementations, semantics or logical formalisms and either interchanged directly together with the program or provided in a public repository e.g. released by standard body such as W3C or OMG.

A more general approach to verify the implementation of a particular semantics is to check them against general principles which a semantics should fulfil. Kraus et al. [26] and Dix [98, 99] proposed several weak and structural (strong) properties for arbitrary (non-monotonic) semantics. While some properties of Dix are adaptations and extensions to those developed by Kraus et al. to compare non-monotonic theories, other such as *partial evaluation*, *modularity* or *relevance* were newly developed. In the following we will briefly review these properties and show how they can be adapted to meta test cases in order to verify a particular semantics. Moreover, we will show how the combination of these meta test cases can be used to automatically determine the semantics of an inference engine, in particular the two major semantics WFS and STABLE. This is vital for a reliable execution of an interchanged rule set in an arbitrary inference engine, when the semantics supported by the inference engine is unknown.

The following structural properties for an entailment relation $|\sim$ for a classical logic language L are derived from [26]:

- *Right Weak*: $\models a \rightarrow b$ and $c \,|\sim a \Rightarrow c \,|\sim a$, where a,b,c are sets of atoms.
- *Reflexivity*: $a \,|\sim a$
- *And*: $a \,|\sim b$ and $a \,|\sim c \Rightarrow a \,|\sim b \wedge c$
- *Or*: $a \,|\sim c$ and $b \,|\sim c \Rightarrow a \vee b \,|\sim c$
- *Left Log. Equiv.*: $\models a \leftrightarrow b$ and $a \,|\sim c \Rightarrow b \,|\sim c$
- *Cautious Monotony*: $a \,|\sim b$ and $a \,|\sim c \Rightarrow a \wedge b \,|\sim c$
- *Cut*: $a \,|\sim b$ and $a \wedge b \,|\sim c \Rightarrow a \,|\sim c$
- *Rationality*: *not* $a \,|\sim \neg b$ and $a \,|\sim c \Rightarrow a \wedge b \,|\sim c$
- *Negation Rat.*: $a \,|\sim b \Rightarrow a \wedge c \,|\sim b$ *or* $a \wedge \neg c \,|\sim b$
- *Disj. Rat.*: $a \vee b \,|\sim c \Rightarrow a \,|\sim c$ *or* $b \,|\sim c$

In [98] the general entailment relation $|\sim$ (see [26] for a model-theory) is adapted to sceptical entailment $|\sim_p$ and the rules for *cumulativity* (i.e. cautious monotony and cut) and *rationality* are refined to:

- *Cumulativity*: If $U \subseteq V \subseteq SEM_P^{scept}(U)$, then $SEM_P^{scept}(U) = SEM_P^{scept}(V)$, where U and V are are sets of atoms and SEM_P^{scept} is an arbitrary sceptical semantics for the program P, i.e. if $a \mid\sim b$ then $a \mid\sim c$ iff $(a \wedge b) \mid\sim c$.
- *Rationality*: If $U \subseteq V, V \cap \{A : SEM_P^{scept}(U) \models \neg A\} = \emptyset$, then $SEM_P^{scept}(U) \subseteq SEM_P^{scept}(V)$

In addition to this structural / strong properties the following weak properties describing general conditions a reasonable (sceptical) semantics should satisfy are derived from [99]:

- *Elimination of Tautologies*: If a rule $a \leftarrow b \wedge not\ c$ with $a \cap b = \emptyset$ is eliminated from a program P, then the resulting program P' is semantically equivalent: $SEM(P) = SEM(P')$. a,b,c are sets of atoms: $P \mapsto P'$ iff there is a rule $H \leftarrow B \in P$ such that $H \in B$ and $P' = P \setminus \{H \leftarrow b\}$
- *Generalized Principle of Partial Evaluation (GPPE)*: If a rule $a \leftarrow b \wedge not\ c$, where b contains an atom B, is replaced in a program P' by the n rules $a \cup (a^i - B) \leftarrow ((b - B) \cup b^i) \wedge not\ (c \cup c^i)$, where $a^i \leftarrow b^i \wedge not\ c^i (i = 1, ..n)$ are all rules for which $B \in a^i$, then the $SEM(P) = SEM(P')$
- *Positive/Negative Reduction*: If a rule $a \leftarrow b \wedge not\ c$ is replaced in a program P' by $a \leftarrow b \wedge not\ (c - C)$ (C is an atom), where C appears in no rule head, or a rule $a \leftarrow b \wedge not\ c$ is deleted from P, if there is a fact a' in P such that $a' \subseteq c$, then $SEM(P) = SEM(P')$:
  Positive Reduction: $P \mapsto P'$ iff there is a rule $H \leftarrow B \in P$ and a negative literal $not\ B \in B$ such that $B \ni HEAD(P)$ and $P' = (P \setminus \{H \leftarrow B\}) \cup \{H \leftarrow (B \setminus \{notB\})\}$
  Negative Reduction: $P \mapsto P'$ iff there is a rule $H \leftarrow B \in P$ and a negative literal $not\ B \in B$ such that $B \in FACT(P)$ and $P' = (P \setminus \{H \leftarrow B\})$
- *Elimination of Non-Minimal Rules / Subsumption*: If a rule $a \leftarrow b \wedge not\ c$ is deleted from a program P if there is another rule $a' \leftarrow b' \wedge not\ c'$ such that $a' \subseteq a, b' \subseteq b, c' \subseteq c$, where at least one $\subseteq$ is proper, then $SEM(P) = SEM(P')$: $P \mapsto P'$ iff there are rules $H \leftarrow B$ and $H \leftarrow B' \in P$ such that $B \subset B'$ and $P' = P \setminus \{H \leftarrow B'\}$
- *Consistency*: $SEM(P) = \emptyset$ for all disjunctive LPs
- *Independence*: For every literal L, L is true in every $M \in SEM(P)$ iff L is true in every $M \in SEM(P \cup P')$ provided that the language of P and P' are disjoint and L belongs to the language of P
- *Relevance*: The truth value of a literal L with respect to a semantics $SEM(P)$, only depends on the subprogram formed from the *relevant rules* of P ($relevant(P)$) with respect to L: $SEM(P)(L) = SEM(relevant(P, L))(L)$

In the first category of properties for a reasonable semantics we mainly focus on *rationality* and *cumulativity*, since in the settings of the redefined sceptical consequence relation the properties *Right Weakening, Relfexivity, Left*

Logical Equivalence are trivial and always satisfied for the sceptical semantics which we consider in this paper. Hence, we only have to proof *Cut* and *Cautious Monotony* for *Cumulativity* and *Rationality* which implies *Cautious Monotony*, *Disjunctive Rationality* and *Negation Rationality*: $Rationality \Rightarrow DisjunctiveRationality \Rightarrow Negation\ Rationality$. The other properties such as *Or* are useful in the context of disjunctive LPs, e.g. to distinguish between an exclusive and an inclusive interpretation of $\vee$. *Cut* is a natural condition for non-monotonic formalisms. To prove *Cut* resp. *Cautious Monotony* we only need to add those atoms that have to be added in order to satisfy *Cut* resp. *Cautious Monotony* such that $SEM(P) \leq SEM(P \cup M_i)$ where M_i is as sequence of atoms a added to P with $M_0 = \emptyset$. *Rationality* is in any sceptical semantics a stronger form of *Cautious Monotony* because $a \mid\sim b \Rightarrow not\ a \mid\sim \neg b$. Rationality can be inductively proved $SEM(P \cup M_i) \leq SEM(P \cup M_{i+1})$ with $M_0 = \emptyset$ and a maximum M_j. In other words, if we can find an extension of atoms to an initial program such that this extension violates the property under test, we have a counter example. We reuse this counter example as a meta test case (meta test program) to verify wether an arbitrary inference engine, i.e. the semantics implemented by this inference engine, solves this counter example. If the meta test program is successful the inference engine satisfies this property. To demonstrate this approach we will now give some examples :

Example: STABLE is not cautious

```
P:  a <- neg b        P': a <- neg b
    b <- neg a            b <- neg a
    c <- neg c            c <- neg c
    c <- a                c <- a
                          c
T:{a=>true,c=>true}
```

$STABLE(P)$ has $\{a, neg\ b, c\}$ as its only stable model and hence it derives a and c, i.e. T succeeds. But, by adding the derived atom c to P we get another stable model $P' := P \cup \{c\} = \{neg\ a, b, c\}$, i.e. a can no longer derived (i.e. T fails) and cautious monotonicity is not satisfied.

Example: REG-SEM is not cumulative

```
P:  b <- c, neg a       P': b <- c, neg a
    a <- neg b              a <- neg b
    c <- neg b              c <- neg b
                            c
T:{a=>true,c=>true}
```

$REG - SEM(P)$ has one regular model $\{a, \neg b, c\}$ which coincides with the unique stable model. Hence, the test case T succeeds, because a and c are the only derivable literals. But, if we add the derivable atom c to P we get the extended program P' which has an additional regular model $\{\neg a, b, c\}$. Thus a

does no longer hold and T fails, i.e. *cumulativity* is not satisfied.

Example: O-SEM is not rational

```
P:  a <- neg a              P':   a <- neg a
    p <- a                        p <- a
    q <- neg p                    q <- neg p
                                  a
T:{neg p=>true,q=>true}
```

$O - SEM(P)$ derives $\{\neg p, q\}$ from P since $\neg a$ is not derivable. Hence, T succeeds. But $P' := P \cup a$ derives $\{a, p\}$ and therefore $\neg p$ and q are not derivable (T now fails), as *Rationality* would require.

Example: EWFS does not satisfy CUT

```
P:  d <- neg c           P': d <- neg c
    a <- neg a               a <- neg a
    b <- neg x, a            b <- neg x, a
    c <- neg b               c <- neg b
                             b
T:={a=>true,b=>true,neg c=>fail,d=>fail}
```

$EWFS(P) = \{a, b\}$ and hence $\neg c$ and d fails, i.e. T succeeds. But $EWFS(P \cup \{b\}) = \{a, b, \neg c, d\}$. Thus T fails for the extended program P' with the derivable atom b added and *Cut* and accordingly *Cumulativity* are not satisfied.

Such sets of *"positive"* meta test program and extended counter/negative test program can be used to verify rationality and cumulativity of the semantics implemented by an inference engine. The initial "positive" meta test program is used to verify if the semantics implemented by the inference engine will provide the correct answers for this particular test case. For example, it might be the case that the meta test program can not be executed at all with the semantics implemented by the target inference engine. If the positive test succeeds we can test the extended meta test program, e.g. via simply adding the additional atom(s) and repeating the test.

We will now take a look at the second kind of properties for a reasonable semantics, the weak properties introduced above. These properties are defined with respect to a semantic equivalence relation $SEM(P) = SEM(P')$ for a particular kind of program transformations from P to P'. In a similar way as with the first kind of strong properties we can provide sets of "positive" meta test programs and counter meta test programs. The counter tests are derived from the initial programs via applying the defined transformations of the respective property under test. Hence, if a counter test fails it proves for a semantics that it does not satisfy this properties. Again, we will illustrate this with some examples:

Example: STABLE does not satisfy Relevance

```
P:  a <- neg b          P': a <- neg b
                            c <- neg c
T:={a=>true}
```

The unique stable model of P is a. If the rule $c < -\neg c$ is added, a is no longer derivable because no stable model exists. *Relevance* is violated, because the truth value of a depends on atoms that are totaly unrelated with a.

Example: GWFS does not satisfy GPPE

```
P:  d <- not b          P': d <- not b
    a <- not b              a <- not b
    b <- c                  b <- d, not a
    c <- d, not a           c <- p, not a
T:={not c=>true,not b=>true,d=>true,a=>true}
```

The minimal models of P are $\{\{b\}, \{d, a\}\}$ and hence $GWFS(P)$ entails *not* c and by negation-as-failure (naf) *not* b, d and a. The minimal models of P' are $\{\{b\}, \{d, a\}\}$, but naf can not be applied as before and therefore $GWFS(P')$ does not entail *not* b, nor d, nor a. Hence, also P' partially evaluates P they are semantically not equivalent which violates the principle of *GPPE*.

Example: SLDNF does not satisfy Elimination of Tautologies and GPPE / COMP does not satisfy Elimination of Tautologies

```
P:  reachable(X)<- edge(X,Y),reachable(Y)
    edge(a,b)
    edge(b,a)
    edge(c,d)

P-GPPE: reachable(a)<-edge(a,b),edge(b,a),reachable(a)
        reachable(b)<-edge(b,a),edge(a,b),reachable(b)
        edge(a,b)
        edge(b,a)
        edge(c,d)

P-Taut: edge(a,b)
        edge(b,a)
        edge(c,d)

T:={not reachable(c)=>true}
```

In the initial program P the SLD-tree for the first rule is not finite, because SLD resolution gets into a loop and hence is neither successful nor fails. Applying the $GPPE$ transformation to P, i.e. replacing the instantiated body atoms $reachable(a)$ resp. $reachable(b)$ by its definitions, leads to the program $P - GPPE$, which still suffers from the loop in SLDNF. Finally, removing the

tautologies leads to the program $P - Taut$, which now neither contains reachable(a) nor reachable(b), nor reachable(c). Here, SLDNF resolution is not in a loop and the negation-as-finite-failure tree for $not\ reachable(c)$ succeeds. As a result, T only succeeds for $P - Taut$, while it neither fails nor succeeds due to the loop for the first and second program. Hence, the semantics of the three programs are not equivalent and both principles are violated. This agrees with the answers of the two-valued COMP. The following completion for the test case for P can be formulated: $reachable(X) \equiv (X \doteq c \vee \exists Y(reachable(Y) \wedge edge(Y, X)))$. However, a completion for programs with tautologies is inconsistent.

Such meta test programs for verifying both kinds of general properties for a reasonable semantics, provides us with a tool for determining an "adequate" semantics to be used for a particular application. For example, an application might require small rule bases, e.g. because the rules are interchanged frequently. Obviously, here the principles of *Elimination of Tautologies* and *Elimination of Non-Minimal Rules* are important, in order to keep the rule base as small as possible without changing its semantics. Moreover, there are strong evidences that by taking both kinds of properties together a semantics might be uniquely determined by these, i.e. via applying the complete test suites consisting of the initial "positive" test programs and the counter examples, we can verify which properties are satisfied by an arbitrary semantics and which are not. The initial "positive" test program acts as a test to decide whether this meta test set can be applied at all, i.e. it might already give us a clue about the logic class supported by the inference engine under test. For example, a test program including disjunctions or explicit negation might not be supported by the inference engine. The second meta test program then verifies the particular property which should be tested by this meta test program, i.e. it acts as a counter example for some semantics. The derived set of satisfied and unsatisfied properties for a particular unknown semantics can then be compared to known results for such properties of different semantics. If the set of satisfied and unsatisfied properties derived by applying the meta test programs in the target inference engine, matches the satisfied and unsatisfied properties of a semantics for a particular logic class, the inference engine most likely supports/implements this semantics. Table 2 (derived from [98, 99]) specifies for some common semantics the properties that they satisfy.

The semantic principles described in this section are also very important in the context of applying refactorings to LPs. In general, a refactoring to a rule base should optimize the rule code without changing the semantics of the program. Removing tautologies or non-minimal rules or applying positive/negative reductions are typically applied in rule base refinements using refactorings [16] and the semantic equivalence relation between the original and the refined program defined for this principles is therefore an important prerequisite to safely apply a refactoring of this kind.

6 Dynamic V&V of ID-based Updates for Evolving Knowledge Bases via expressive Integrity Constraints

In this sections we introduce another important application domain of test-driven V&V, namely dynamic self-updates of the LP at runtime, triggered, e.g. by user queries or by active ECA rules, where an update of the knowledge base is performed as a reaction to an occured/detected (complex) event (see [123, 124]). This leads to an evolving knowledge base, which changes over time. The updates might not be constraint to add or remove facts from the extensional knowledge base (fact base), but might also be used to update the intensional knowledge base (rule base), i.e. add, remove or change rules or complete rule sets and hence dynamically evolve the LP at runtime. Obviously, this might lead to errors as anomalies, as defined in section 2. Furthermore, in many rule-based projects 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 this part of the intended behavior of the LP. A common way to represent such constraints are integrity constraints (IC). 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 new pattern: **V&V&I**. We understand ICs as a way to formulate consistency (or inconsistency) criteria and use ICs to verify the consistency of a evolving KB.

Definition

- *An integrity constraint on a LP is defined as a set of conditions that the constrained KB must satisfy, in order to be considered as a consistent model of the intended (real-world) 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 on a program (LP) P with respect to the set of integrity constraint $IC := \{ic_1, ..ic_i\}$ defined in P is the satisfaction of each*

Table 2. Table (General Properties of Semantics)

Semantics	Class	Cumul.	Rat.	Taut.	GPPE	Red.	Non-Min.	Rel.	Cons.	Indep.
COMP	Normal	-	•	-	•	•	•	-	-	-
$COMP_3$	Normal	•	•	-	•	•	•	-	-	-
WFS	Normal	•	•	•	•	•	•	•	•	•
STABLE	Normal	-	•	•	•	•	•	-	-	-
WGCWA	Pos. Disj.	-	•	-	•	•	-	•	•	•
CGWA	Strat. Disj.	•	-	•	•	•	•	•	•	•
PERFECT	Strat.Disj.	•	-	•	•	•	•	-	•	•

$ic_i \in IC$ at each state $P' := P \cup M_i \Rightarrow P \cup M_{i+1}$ with $M_0 = \emptyset$, where M_i is an arbitrary update adding/removing/changing rules or facts to the KB.

Accordingly, integrity constraints are closely related to our notion of test cases for LPs. Test cases can be seen as more expressive integrity constraints, which can temporarily add (meta) rules or facts to the KB - the so called test assertions - for the purpose of V&V and which are more expressive in defining constraints on the models of the LP, e.g. quantifications on the number of variable bindings for a particular variable or substitution sets for "variable/ground value" bindings. A test cases without temporal assertions and with simply test query constraints, e.g. a test $a? => true$ (a must be derivable), semantically reduces to an integrity constraint in our definition. From a syntactical perspective we distinguish integrity constraints from test cases, since in our (ContractLog) approach we represent and manage test cases as stand-alone LP scripts (files) which are imported to the knowledge base (consulted in the main LP), whereas integrity constraints are directly defined in the main LP. We will describe this in more detail in the "implementation" section 7. However, both ICs and test cases can be used for verification and validation of knowledge updates. We will first address the issue of dynamic updates for evolving LPs/KBs mainly in the context of active ECA rules and then illustrate our approach of safeguarding this process using ICs and test cases including support for transactional self-updates, hypothetical tests and rollbacks.

Knowledge updates at runtime are not included by the standard (Horn) semantics of logic programs. Nevertheless, most LP languages such as Prolog support built-in update primitives such as *assert*, *retract* or *retractall* (bulk updates retracting all specified knowledge) which typically allow for updating the extensional knowledge base. They do not have a logical declarative semantics but are defined procedurally and have some nice properties, e.g. updates are executed directly by the SLD-style proof procedure and not via external runtime systems, updates are persistent and not just hypothetical calculi etc. However, such procedural implementations give no answers how update primitives should interact with other logical operators such as disjunction and negation and how the order of updates can be logically treated, e.g. depending on the order of updates under strict procedural semantics the outcome of an execution might vary so that a unique outcome can not always guaranteed, which is called a confluence problem. Another problem is if a particular update in a transaction (sequence of updates) fails - shall the system proceed with the next update, simply abort, or rollback all previous updates? Using updates as events for further update actions, e.g. in active ECA rules, can lead to loops due to cyclic dependencies between updates which are conditional to each other and therefore induce termination problems. Several solutions to these problems have been proposed in the area of active and deductive databases applying special newly developed logics such as transaction logics, dynamic logics, process logics or applying special techniques for checking termination and confluence properties as well as approaches for rewriting update programs consisting of active rules into deductive rules and applying confluent semantics such as well-founded semantics. Other solutions are more

software engineering related and make use of techniques from the database world with its ACID properties, locks and its commit protocols e.g. 2-phase commits or annotate events/updates with time stamps to explicitly define the order of events/updates. However, most of these approaches are based on extensive procedural and logical extensions to the standard rule engines and LP logics, e.g. new logic types such as transaction logics or additional special systems annotating events/actions with time stamps. Applying declarative, confluent semantics (e.g. well-founded semantics) solves some of the mentioned problems such as termination of update sequences in active rules and to some extend confluence - however needs heavy transformation approaches, e.g. from execution models for active rules to derivation rule LPs [125]. However, this creates other problems, e.g. parallel execution of active ECA rules is forbidden. In a nutshell, most of these approaches are not applicable in standard backward-reasoning rule engines and logic programming without heavy re-implementations and extensions and in general contradict the paradigms of logic programming. In [123] and [124] we have elaborated on a homogeneous ECA-LP language combining ECA rules, derivation rules and KR-based event logics to formulate complex event and action definitions using an event/action algebra defined in terms of a novel interval-based Event Calculus variant. As a result the semantics of the ECA-LP language and in particular the definition of complex events and actions inherits from the declarative semantics of the used KR formalisms (Event Calculus and Logic Programming). Since our focus in this paper is on V&V we concentrate on the test-driven approach using test cases resp. integrity constraints to safeguard arbitrary complex updates, in particular to solve the problems mentioned above.

Semantics of Dynamically Evolving LPs

We start with defining a general notion of a semantics for a dynamically evolving LPs, that allows us to cover a wide range of logic classes and logic programming semantics, as discussed in section 5. In particular we view a program as a 3-valued theory with the values "true", "valse" and "unknown" and $\leftarrow$ is interpreted according to Kleene which is true for $undefined \leftarrow undefined$ and false for $false \leftarrow undefined$. A semantics SEM assigns to every program P a set of 3-valued models of the language L of P: $SEM(P) \subseteq MOD_{3-val}^{L_P}(P)$. 2-valued models are treated as special 3-valued ones, i.e. any semantics based on a two-valued theory Σ can be seen as a three-valued interpretation $I = (T; F)$ with $T = \{A : A \text{ a ground atom with } \Sigma \models A\}$ and $F = \{A : A \text{ a ground atom with } \Sigma \models \neg A\}$. In most cases we are only interested in the Herbrand models: $SEM(P) = M^{Herb}$. In case a particular semantics by definition can only contain a single model such as WFS we simply write $SEM(P) = M$. We assign a sceptical entailment relation to SEM: $SEM^{sceptical}(P)$ which is the set of all atoms which are true in all models of $SEM(P)$. Proof-theoretically a semantics $SEM(P)$ of a program P is defined as a set of literals that are derivable from P using a particular inference mechanisms. We define expressiveness of our LP language (depends on the logic class) in the usual way: The instances D of a

finite EDB-relations of P is an input argument to derive the finite IDB-relation as output under a respective semantics SEM. For every finite D there is associated a relation R^D on D if there is a program P containing an IDB-symbol r s.t. for every database D and tuple t corresponding to r: $r(t) \in SEM(P + D)$ iff $R(t)$ holds in D.

Expressive Logic Programming Knowledge Updates

We define a *positive update* to a program P as a finite set $U^{pos} := \{rule^N : H \leftarrow B, fact^M : A \leftarrow\}$ with A an atom denoting a fact, $H \leftarrow B$ a rule, $N = 0, .., n$ and $M = 0, ..m$. Applying U^{pos} to P leads to the extended program $P' = P \cup U^{pos}$. Applying several positive updates as an increasing finite sequence U_j^{pos} with $j = 0..k$ and $U_0^{pos} := \emptyset$ to P leads to a program $P_k = P_0 \cup U_0^{pos} \cup U_1^{pos} \cup ... \cup U_k^{pos}$. In other words a program P_k reflects a particular knowledge state k and is decomposable in the previous knowledge state $k-1$ plus the update: $P_k = P_{k-1} \cup U_k^{pos}$. We define $P_0 = \emptyset \cup U_0^{pos}$ and $U_1^{pos} = \{P :$ the set of rules and facts defined in the original program $P\}$, i.e. loading/parsing the program from a LP script is the first update.
Likewise, we define a *negative update* to a program P as a finite set $U^{neg} := \{rule^N : H \leftarrow B, fact^M : A \leftarrow\}$ with $A \in P$, $H \leftarrow B \in P$, $N = 0, .., n$ and $M = 0, ..m$, which is removed from P, leading to the reduced program $P' = P \setminus U^{neg}$. Applying arbitrary sequences of positive and negative updates leads to a sequence of program states $P_0, .., P_k$ where each state P_i is either $P_i = P_{i-1} \cup U_i^{pos}$ or $P_i = P_{i-1} \setminus U_i^{neg}$. As we will see later a rollback of an update is therefore a step back to a previous knowledge state by inverting the last update(s), e.g. a "rollback" to the latest state which preserves integrity of the knowledge base.

A Labelled Logic and ID-based Updates

We now extend the logical language to a labelled logic. In fact, the ContractLog language is a l*abelled, typed, prioritized rule language*, implementing a hybrid polymorphic order-sorted typed unification, e.g. using Semantic Web ontologies or Java class hierarchies as explicit type system, see [83, 75] and defeasible rule priorities, see [75]. Our focus here is on the labelled extension. Our goal is to represent such information homogeneously *in* one representation language and derive and use such information dynamically during resolution. Akin to object oriented programming we treat rules (as well as atoms and terms [83]) as objects having an *object id* (*oid*). We use these oids as a set of rule names (labels) used to label rules (extended LPs with default and explicit negation): $< oid > L_0 \leftarrow L_1 \wedge .. \wedge L_n \wedge \sim L_{n+1} \wedge .. \wedge \sim L_m \wedge \neg L_{m+1} \wedge .. \wedge L_k$

The label *oid* is optional. If omitted, the label is said to be empty. The label is required to be an unique object id. The rule labels are treated as a binary naming function $oid(< label >, < rule - head >)$, i.e. the function is a partial injective naming function that assigns an *oid* to some of the rules in P.

Example

```
oid(rule1,p(X)).
p(X):-q(X).
q(1).
```

Rules can be grouped to rule sets, which we call a *module* M. A module also has an unique *oid* as its label with a unary naming function $module(< label >)$. To be able to refer a set of rules to a particular module, we reuse the *oid* function and associate to each rule $r_i \in M$ a relation $oid(< M_{label} >, < r_i head >)$.
Example

```
module(m1).

oid(m1,r1(X)).
oid(rule1,r1(X)).
r1(X):-q(X).

oid(m1,r2(X)).
oid(rule1,r2(X)).
r2(X):-r1(X).
```

This homogeneous definition of labels for rules (facts are special rules) and rule sets (modules) enables (meta) reasoning and is useful in many ways. In the ContractLog KR we have implemented an expressive defeasible logic as a LP meta program which uses the labels to define preferences between rules and also between rule sets (modules). Defining explicit priorities, e.g. $r2 \succ r1$ represented as a binary function $overrides(r2, r1)$ (rule $r2$ overrides rule $r1$), can be used to define default rules, alternatives, exceptions and in general handle conflicts between conflicting rules using the defined preferences. This is crucial in many domains, such as e.g. deontic reasoning (reasoning about norms) or legal reasoning, e.g. to represent typical legal principles such as "Lex Posterior" or "Lex Superior" for resolving legal conflicts. It enables partial meta reasoning on selected rules / modules via their labels. In contrast to most non-monotonic logics such as prioritized circumscription, hierarchic autoepistemic logic, prioritized default logic, which represent the preference information in an "external" manner expressed outside of the logical language, our approach uses a homogeneous representation language, which simplifies engineering, management, maintenance and testing. Grouping rules to modules in particular facilitates large programs or projects to be put together from components which can be developed, compiled and tested separately. Also module systems enforce typical SE principles such as the principle of information hiding and can provide a basis for data abstraction and modularization. This is vital for example, in rule-based contract representation, where contracts, e.g. 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 e.g. Rule Based Service Level Agreement - RBSLA project). For more information on the defeasible approach to conflict resolution and rule prioritization in ContractLog see [75].

After having defined the principles of a labelled logic, we now extend it with labelled updates, which we call *ID-based updates*. As described an update is a set of rules which is added or removed from the knowledge base. Hence, an update can be seen as a module (a set of rules / facts) with a label (an unique ID *oid*): U^{pos}_{oid}. The *oid* of the update is preserved in the knowledge bases naming all rules and relating them to the update id using the naming functions *module* and *oid*. Based on this labelled ID concept we have implemented expressive update functions as declarative LP meta programs, which enable external imports and arbitrary updates, adding, removing or changing rules and facts, including bulk updates (sets of rules / facts). ContractLog supports the following update primitives (functions):

- *update*: used to add new knowledge to the KB either importing an external LP script from an URI or using the set of defined rules and facts.
- *remove*: used to remove knowledge from the KB using its ID (fact oid, rule oid, module oid).

Examples

```
update("./examples/test/test.prova").   % add an external script
update(id1,"r(1):-f(1). f(1).").        % add rule "r(1):-f(1)." and fact "f(1)." with ID "id1"
update(id2,"r(X):-f(X).").              % add rule "r(X):-f(X)." with ID "id2"
p(X,Y) :- update(id3,
                 "r(_0):-f(_0), g(_0). f(_0). g(_1).", %Object place holders _N: _0=X ; _1=Y.
                 [X,Y]).                % update with variable bindings X,Y.
remove(id1).                            % remove update/module with ID "id1"
remove("./examples/test/test.prova").   % remove external update
```

Integrity Constraints

As already discussed in the context of knowledge updates conflicts might naturally arise, in particular when the rule program grows larger and updates are done by different people. Let us first define what we mean by conflicts - see also section 2:

Definition

- *A rule* $r1$ *and a rule* $r2$ *are conflicting iff the head of* $r1$ *is the complement of the head of* $r2$ *and both can be derived*
- *A rule* $r1$ *and a rules* $r2$ *are conflicting with respect to a set of rules* R *iff the application of one rule defeats the other or vice versa.*
- *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 (explicit negated) can be derived at the same time. To describe the second type of conflict consider the following program:

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

Here the heads of both rules are not complementary, but the both rules defeat each other (using WFS) due to the negation. Such conflicts are much harder to detect and might also occur indirect based on the existence of other rules. Internal integrity constraints or external test cases can be used to define conditions which denote a logic or application specific conflict. Integrity constraints in ContractLog are defined as a n-ary function $integrity(< operator >, < conditions >$) in the rule language. We distinguish four typed of integrity constraints:

- *Not-constraints* which express that none (akin to strong negation) of the stated conclusions should be drawn: $integrity(not(p_1(...), .., p_n(...)))$.
- *Xor-constraints* which express that the stated conclusions are mutual exclusive, i.e. should not be drawn at the same time: $integrity(xor(p_1(...), .., p_n(...)))$.
- *Or-constraints* which express that at least one of the stated conclusions must be drawn: $integrity(or(p_1(...), .., p_n(...)))$.
- *And-constraints* which express that all of the stated conclusion must draw: $integrity(and(p_1(...), .., p_n(...)))$.

IC might be also stated conditionally as *integrity rules*, e.g. $integrity(xor(p(), q())) : -a(), b()$., i.e. the integrity constraint that $p()$ and $q()$ are mutual exclusive, denoting a conflict if both can be derived at the same time, must hold iff $a()$ and $b()$ hold.

Model Theory and Proof Theory of Integrity Constraints

The semantics of integrity constraints SEM_{IC} is defined as an inference system that is directly derived from the semantics of logic programs. Integrity constraints are defined as constraints on the set of possible models and therefore describe the model(s) which should be considered as strictly conflicting. Model theoretically we attribute a 2-valued truth value (true/false) to an integrity constraint using the defined set of constraints (literals) in an IC as a goal on the program P. Roughly speaking, the truth of an integrity constraint in a finite interpretation I is determined by running the goal G_{IC} defined by the IC on the clauses in P or more precisely on the actual state of P_i. If the G_{IC} is satisfied, i.e. there exists at least one model for the sentence formed by the G_{IC}: $P_i \models G_{IC}$, the integrity constraint is violated and P is proven to be in an inconsistent state with respect to the integrity constraints IC: IC is violated resp. P_i violates integrity iff for any interpretation I, $I \models P_i \rightarrow I \models G_{IC}$.
In general an interpretation $I = (\triangle^I, \cdot^I)$ consists of a non-empty domain $\triangle^I$ and a interpretation function $\cdot^I$. For example, in case of a Herbrand definition for definite LPs, the Herbrand interpretation of P_i is any subset $I \subseteq B_{P_i}$ of its

Herbrand base. A Herbrand model of P_i is a Herbrand interpretation of P_i such that for each rule $H \leftarrow B_1 \wedge .. \wedge B_n$ in P_i the interpretation satisfies the logical formula $\forall x(B_1 \wedge .. \wedge B_n)$, where x is a list of variables in the rule. We define the following interpretation for the four types of ICs:

And $and(C_1, .., C_n)$: $P_i \models (notC_1 \vee .. \vee notC_n)$ if exists $i \in 1, .., n, P_i \models not\ C_i$
Not: $not(C_1, .., C_n)$: $P_i \models (C_1 \vee .. \vee C_n)$ if exists $i \in 1, .., n, P_i \models C_i$
Or: $or(C_1, .., C_n)$: $P_i \models (notC_1 \wedge .. \wedge notC_n$ if for all $i \in 1, .., n, P_i \models not\ C_i$
Xor: $xor(C_1, .., C_n)$: $P_i \models (C_j \wedge C_k)$ if exists $j \in 1, .., n, P_i \models C_j$ and exists $k \in 1, .., n, P_i \models C_k$ with $C_j \neq C_k$ and $C_j \in C$, $C_k \in C$

$C := \{C_1, ..C_n\}$ are positive or negative (explicit negated) n-ary atoms which might contain variables; *not* 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. If there exists a model for a IC goal (as defined above), i.e. the "integrity test goal" is satisfied $P \models G_{IC}$, the integrity constraint is assigned true and hence integrity is violated.
To proof integrity constraints we have implemented a LP meta program in the ContractLog KR. Meta-programming and meta-interpreters have their roots in the original von Neumann computer architecture where program and data treated in a uniform way and are a popular technique in logic programming for representing knowledge [126], in particular, knowledge in the domains containing logic programs as objects of discourse. LPs representing such knowledge are called meta-programs (a.k.a. meta interpreters) and their design is referred to as meta-programming. The meta program implements two main test axioms:

- *testIntegrity*() tests the integrity of the actual program, 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 literal, which is actually not in the KB, violates any integrity constraint in the KB.

The first integrity test is useful to verify (test logical integrity) and validate (test application/domain integrity) the integrity of the actual knowledge state. The second integrity test is useful to hypothetically test an intended knowledge update, e.g. test wether a conclusion from a rule will lead to violations of the integrity of the program. If the hypothetical test succeeds, i.e. the integrity constraint fails, the rule can be safely added. A similar set of test axioms is provided for test cases:

- *test*() apply all test cases which are currently "loaded" to the KB to the rules and facts in the KB.
- *test*(< *oid* >) apply a particular test case denoted by its *oid* to the rules and facts in the KB.

Two predicates $loadTestCase(< TestCase >)$ and $unloadTestCase(< TestCase >)$ to dynamically load and unload external test cases using an URI are provided. The implementation is mainly based on the previously described update primitives. Since we use a meta programming representation we can directly reason with the defined test axioms, e.g. it is possible to define a negative test $not(testIntegrity(Literal))$ in order to find all literals (rule heads/facts) out of a list of possible intended updates which will violate integrity and exclude those from the update. Another example is defeasible reasoning where conflicts between defeasible conclusions are solved by explicit preference definitions. We have implemented a defeasible logic variant based on integrity constraints in the ContractLog KR - see [75]

Transactional Updates

We now use the concept of (hypothetical) integrity test to extend the ID-based updates to *transactional updates*. A transactional update is an update which must be executed completely or not at all. In case a transactional update fails, e.g. is only partially executed, it will be rolled back otherwise it will be committed. We define a transactional update with a label U_{oid}^{trans} to a program P as pair $U_{oid}^{trans} =< U^{pos/neg}, T >$, where $U^{pos/neg}$ is either a positive or a negative update and T is the set of integrity constraints defined in P (plus a set of additionally specified external test cases). The integrity constraints (and test cases) are use to V&V&I the hypothetically updated program P_{i+1}^{hypo} and rollback the (possibly partial executed) transactional update to the state $P_{i+1} = P_i$ in case a test case was not successful or the integrity is violated with respect to the integrity constraints. In case T is successful the transactional update is committed leading to the new knowledge state $P_{i+1} = P_i \cup U_{oid}^{trans_{pos}}$ for positive transactional updates resp. $P_{i+1} = P_i \setminus U_{oid}^{trans_{neg}}$ for negative updates:

$P_{i+1} = P_i$ iff exists $j \in 1, .., n, P_{i+1} \models IC_j$
$P_{i+1} = P_i \cup U_{oid}^{trans_{pos}}$ resp. $P_{i+1} = P_i \setminus U_{oid}^{trans_{neg}}$ otherwise

Note, that we have defined the violation of an integrity constraint IC_j as the satisfaction of IC_j in the program state P_i. Hence, if there exists a satisfied integrity constraint in the hypothetical state P_{i+1}^{hypo} the transactional update is rolled back via inverting the update primitive:

$P_{i+1} = P_i \cup U_{oid}^{trans_{pos}} \Rightarrow P_{i+1}^{hypo} \setminus U_{oid}^{trans_{pos}}$ iff exists $j \in 1, .., n, P_{i+1} \models IC_j$
$P_{i+1} = P_i \setminus U_{oid}^{trans_{neg}} \Rightarrow P_{i+1}^{hypo} \cup U_{oid}^{trans_{neg}}$ iff exists $j \in 1, .., n, P_{i+1} \models IC_j$

A commit is defined as $P_{i+1}^{hypo} \Rightarrow P_{i+1}$.

To support transactions in ContractLog we have extended the LP meta programs for integrity constraints and ID-based updates with a meta program implementing different variants of a transaction function $transaction()$.

Examples

```
transaction(remove(<ID>)). % transactional remove with test on all ICs/TCs in KB
transaction(update(<ID>,...)). % transactional update with test on all ICs/TCs in KB
transaction(update(<ID>,...),<Test Case URI>). % transactional update with explicit test case.
commit(<ID>).        % commit transaction with specified ID of update
rollback(<ID>).      % rollback transaction with specified ID
```

If no explicit test case is specified the transactional update/remove performs a test on all integrity constraints and test cases in the KB. If the tests fail the transaction is automatically rolled back otherwise it will be committed. An explicit test case can be specified, which will be loaded dynamically (using the ID-based update primitives to load external scripts from a URIs) and used to test the transactional update/remove. After the test the external test case will be removed from the KB. It is also possible to explicitly rollback or commit a transaction. In general integrity constraints and test cases are used for V&V&I but they can be also used to safeguard the completeness of the transaction, which is in particular important in case of long running bulk updates. For example consider the following program (in Prolog like syntax):

```
a() :- not(b()).
b() :- not(a()).
```

If now two facts $a()$. and $b()$. are added to the KB by the transactional update $transaction(update(id1,"a().b().")," t5.test")$ which use the following external test case t5.test: $\{?a => true, ?b => true\}$ to safeguard the transactional update. If only $a()$. or only $b()$. is asserted to the knowledge base the test case fails (we assume WFS) and the partial update is automatically rolled back (by the meta program). If both are added or none of them the test case succeeds which is exactly the intended behavior of this transactional update.

Active ECA rules with Postcondition-Tests

Automatically changing/adapting the program as a reaction to occurred events is crucial in many rule-based application domains which need some kind of reactive behavior. In [123, 124] we have elaborated on a homogeneous Event-Condition-Action logic programming language (ECA-LP) which combines ECA rules and derivation rules and enables active execution of ECA rules in arbitrary backward-reasoning systems. The approach uses the LP inference engine for reasoning and inference tasks such as variable binding, negation or logical connectives, thus benefiting from the declarative semantics of logic programs. We have developed a LP based event / action algebra based on a novel interval-based Event Calculus formalization which supports processing of complex events and actions redefining well-known event operators from event algebras in the active database domain such as Snoop [128]. Knowledge self-updates, as described in this section, can be performed as actions in ECA rules. Note, that external updates in external systems are also supported via a highly expressive typed logic and procedural attachments on Java objects. In short, in ECA-LP an ECA rule is defined as: $eca(Time, Event, Condition, Action, Postcond., Else)$. The terms in an ECA rules are complex terms which are interpreted as goals on derivation rules which

implement the functionality of each part of the ECA rule. This approach leads to a compact syntax for ECA rules, a tight combination of ECA rules and derivation rules and it supports reusability of complex functionalities implemented by arbitrary chained derivation rules in several ECA rules.

```
eca(time(T1,..,Tn), event(E1,..,Em), condition(C1,..,Ck),
action(A1,..,Al), postcondition(P1,..,Po), else(El1,..,Elp))

time(T1,..,Tn) <- [ body of derivation rule]
event(E1,..,Em)  <- [body of derivation rule]
condition(C1,..,Ck) <- [ body of derivation rule]
action(A1,..,Al) <- [ body of derivation rule]
postcondition(P1,..,Po) <- [ body of derivation rule]
else(El1,..,Elp) <- [ body of derivation rule]
```

The time part specifies a special time event - a time function - which can be used to define schedules, intervals or time points at which the ECA rule should be processed. The event, condition and action part are used as usual in ECA rules. The else action is executed as an alternative to the normal action in case the action of the ECA rule can not be executed, e.g. in case the condition is not satisfied. It can be used to define alternative actions or perform some kind of exception handling. Finally, the postcondition can be used to define post conditional constraints. In particular it can be used to apply post conditional test on the outcome of the action as we will see later or it can be used to prevent backtracking to further variable bindings in the action part via setting a cut "!". To illustrate the usage of ECA rules consider an ECA rule which states that

"*every 10 seconds it is checked (time) whether there is an incoming request by a customer to book a flight to a certain destination (event). Whenever this event is detected, a database look-up selects a list of all flights to this destination (condition) and tries to book the first flight (action). In case this action fails, the system will backtrack and book the next flight in the list other-wise it succeeds (post-condition cut) sending a "flight booked" notification If no flight can be found to this destination, i.e. the condition fails, the else action is triggered, send-ing a "booked up" notification back to the customer.*"

This is formalized as a LP (Prolog related syntax) as follows:

```
eca( every10Sec(), detect(request(Customer, Destination),T),
find(Destination, Flight), book(Customer, Flight), !,
notify(Customer, bookedUp(Destination) ).

% time derivation rule
every10Sec() :-  sysTime(T), interval( timespan(0,0,0,10),T).
% event derivation rule
detect(request(Customer, FlightDestination),T):-
occurs(request(Customer,FlightDestination),T),
consume(request(Customer,FlightDestination)).
% condition derivation rule
find(Destination,Flight) :-
 on_exception(java.sql.SQLException,on_db_exception()), dbopen("flights",DB),
    sql_select(DB,"flights", [flight, Flight], [where, "dest=Destination"]).
% action derivation rule
book(Cust, Flight) :- flight.BookingSystem.book(Flight, Cust),
```

```
notify(Cust,flightBooked(Flight)).
% alternative action derivation rule
notify(Customer, Message):- sendMessage(Customer, Message).
```

In the example the action $book(Cust, Fligth)$, implemented within the "action" derivation rule, calls the external (static) method *book* from the Java class $flight.BookingSystem$. The postcondition sets a cut !, i.e. if the action is successfully performed no backtracking to further bindings of the variable $Flight$ in the action part is performed. While this example shows an external update in the $flight.BookingSystem$, ECA rules can also perform internal self-updates. Due to the homogeneous combination of ECA rules and derivation rules in a LP, the formalization is straight-forward using either the update primitive directly in an ECA rule, e.g.:

$eca(..., update(id, "p() :- q()."), ...).$

or via defining a derivation rule for the action:

$eca(..., evolve("p() :- q()."), ...).$
$evolve(Updt) :- update(id, Updt).$

To safeguard the application of self-updates (updating rules and facts) in the context of (re)active rules we use the post condition part. The post condition can be used to define additional constraints which must hold after the action is performed. For example consider the following program (Prolog related syntax; SLDNF resolution):

```
eca(_,not(a()),_,update(tid1,"a(")testTC(),_). % ECA rule definition

%POSTCONDITION
testTC():-
    a(),
    commit(tid1),
    println(["Test succeeded - commit updates"]),
    !.      % cut = prevent rollback

testTC():-
    println(["Test failed - rollback updates !!!!"]),
    rollback(update(tid1)),
    fail().
```

The ECA rule defines an update with id $tid1$ which adds a single fact "a()." to the KB. The post condition is defined by the two derivation rules $testTC$. The first rule commits the update $td1$ in case the test on $a()$ succeeds and prints a success message. The second rule performs a rollback on the update, if the first rule was not successful and finitely fails. Due to the definition of the event in the ECA rule as $not(a())$ the ECA rule is executed as long as the update was not successful, i.e. $a()$ has not been added to the KB.

7 Integration of Test Cases into Testing Frameworks and Rule Markup Languages

We have implemented the test drive approach in the ContractLog KR. The ContractLog KR is an expressive and efficient KR framework hosted at Sourceforge for the representation of contractual rules, contractual policies and service level agreements implementing several logical formalisms such as event logics,

defeasible logic, deontic logics, description logic programs in a homogeneous LP framework. The ContractLog KR is developed within the Rule Based Service Level Agreement (RBSLA) project.

RBSLA project: http://ibis.in.tum.de/staff/paschke/rbsla/index.htm
RBSLA distribution: https://sourceforge.net/projects/rbsla
ContractLog: http://ibis.in.tum.de/staff/paschke/rbsla/contractlog.htm

Test cases in the ContractLog KR are homogeneously integrated into Logic Programs and are written in the same representation language. We use an extended ISO Prolog related scripting syntax (ISO Prolog ISO/IEC 13211-1:1995) to write LP scripts, where a variable starts with a upper-case letter, e.g. X, Y, Z, a constant/individual with a lower-case letter, e.g. a, b, c and a queries is written as a function : $-sovle(...)$ or : $-eval(...)$. A test case script consists of a unique test case ID denoted by $testcase(< ID >)$, optional input assertions such as input facts and test rules, a positive meta test rule defining the test queries and variable bindings $testSuccess(< TestName >, < OptionalMessageforJunit >)$, a negative test rule $testFailure(< TestName >, < Message >)$ and a $runTest$ rule which is used by the meta program which implements the test axioms. The positive and negative test rules are interpreted by the ContractLog KR using the test rules to derive the success or failure of the test case. A test case can be temporarily loaded and removed to/from the knowledge base for testing purposes, using the expressive ID-based update predicates for evolving logic programs.

Example

LP:

```
a():-c().
b():-c().
c(). % input fact
:-solve(test(./examples/tc1.test)).
```

Test Case: $tc1 := \{a()? => true, b()? => true\}$

```
testcase("./examples/tc1.test"). % id

% positive test with success message for JUnit report
testSuccess("test1","succeeded"):-testcase(./examples/tc1.test),a(),b().

% negative test with failure message for Junit report
testFailure("test1","can not derive a and b"):- not(testSuccess("test1",Message)).

% define the active tests - used by meta program
runTest("./examples/tc1.test"):-testSuccess("test 1",Message).
```

The example shows a simple propositional LP with two rules and a query which imports and test the test case $./examples/tc1.test$ from the relative path to

the test case script. The test case defines a positive test named *test*1 with the two test queries $a()$, and $b()$ (represented as subgoals). The test case id "*./examples/tc*1.*test*" is used to distinguish test cases in case several test cases have been imported to the knowledge base. The second term in the test rules' heads specifies a success resp. failure message which is used for reporting, e.g. to create a JUnit test report. The *runTest* rule with the test case ID as argument is evaluated by the meta program implementing the *test* functionality. It specifies which tests (*testSuccess*) of the test cases should be tested. A test case might define several tests (*testSuccess*/*testFailure*) and some of them might be possibly excluded (e.g. might be deprecated) from testing via removing them from the *runTest* rule. In the example the *runTest* only tests the test *test*1.

Integration of Test Cases into the JUnit Test Framework

In the recent years several successful tools to support agile test-driven software development have emerged in extreme programming such as the build tool ANT which automates project build processes or testing tools such as JUnit. JUnit by K. Beck and E. Gamma [129] provides a standard API for testing java code. The RBSLA/ ContractLog KR distribution implements support for JUnit based testing and test coverage reporting where logic programming test cases can be managed in test suites and automatically run by a JUnit Ant task which creates a final test case report from the failure and success messages derived of the executed tests. As described in section 6 test cases in ContractLog are written as LP scripts. Test cases can be bundled to test suites which are also represented as LPs consisting of a test suite ID denoted by the naming function $test\_suite(< name >)$. and a list of test cases referenced by their URI $test\_case(< URI >)$.

Example:

```
test_suite("basic function tests").

test_case("./examples/test_cases/basics/complexTestCase.prova").
test_case("./examples/test_cases/basics/cutTestCase.prova").
test_case("./examples/test_cases/basics/listTestCase.prova").
test_case("./examples/test_cases/basics/varTestCase.prova").
```

A java class *rbsla.regressiontest.RegressionTest.java* provides methods to build a JUnit test suite *junit.framework.TestSuite* from a ContractLog test suite script.

```
public static TestSuite buildSuite(String uri) {
    ProvaWrapper wrapper = new ProvaWrapper(uri);
    String suiteName =
        wrapper.solveVar("test_suite(SuiteName)","SuiteName").get(0).toString();
    TestSuite suite  = new TestSuite (suiteName);
    List testCases = wrapper.solveVar("test_case(TestCase)","TestCase");
```

```
    for(int i = 0; i < testCases.size(); i++) {
        try {
            String testCase = testCases.get(i).toString();
            RegressionTest test = new RegressionTest(testCase);
            suite.addTest (test);
        } catch(Exception x) {
            System.err.println ("Cannot create test suite!");
            x.printStackTrace (System.err);
        }
    }
    return suite;
}
```

It uses a wrapper *rbsla.wrapper.ProvaWrapper* on the underlying rule engine's Java API (Prova http://comas.soi.city.ac.uk/prova/) to load and query a ContractLog test suite script using the specified *uri*. ContractLog implements an abstract wrapper interface *rbsla.wrapper.KnowledgeBaseWrapper* to decouple the ContractLog Java implementations from the implementations of a rule engine, so that the ContractLog framework can be used on top of different inference engines. Currently, ContractLog implements specialized wrappers on Prova (ProvaWrapper) and Mandarax (MandaraxWrapper; Mandarax: http://mandarax.sourceforge.net/). The wrapper is used to derive the name of the ContractLog test suite , which is then used as name for the JUnit test suite. Further, it queries all URIs of the test cases defined in the test suite script and uses this references to instantiate the the main test class *rbsla.regressiontest.RegressionTest* which subclasses the JUnit class *TestCase*. The *RegressionTest* sets up the test case via loading the test case from the URI into the knowledge base (including all tests, assertions and meta information defined in test case script) using the Prova wrapper.

```
    protected void setUp() {
        if (lp!=null) {
            wrapper = new ProvaWrapper(lp);
            wrapper.consult("consult("+tc+").");
        }
        else wrapper = new ProvaWrapper(tc);
    }
```

It then runs the test case querying the positive *testSuccess* and negative *testFailure* tests defined in the ContractLog script (see section 6) using the wrapper and creates a test report from the derived success resp. failure messages using the JUnit asserts. JUnit which has been integrated with an Ant task into the build script of the RBSLA/ContractLog distribution automatically creates a HTML based report (other output format are also possible) from this information.

```
public void test() throws Exception {
    setName(tc); // set test case name
```

```
    StringBuffer type = new StringBuffer("");
    boolean status = false;

    // positive test
    ResultSet rs = wrapper.consult("testSuccess(Name,Message)");
    boolean more = rs.first();
    while (more) {
        Object name = rs.getResult(Object.class, "Name");
        Object message = rs.getResult(Object.class, "Message");
        type=type.append("++++  "+name+" succeeded "+message+"  ++++   |   ");
        more = rs.next();
        status = true;
    }

    // negative test
    rs = wrapper.consult("testFailure(Name,Message)");
    more = rs.first();
    while (more) {
        Object name = rs.getResult(Object.class, "Name");
        Object message = rs.getResult(Object.class, "Message");
        type=type.append("----  "+name+" failed - "+message+"  ----   |   ");
        status = false;
        more = rs.next();
    }


    if (status) assertTrue(type.toString(),true);
    else assertTrue(type.toString(),false);
}
```

After a test case has been processed by JUnit the test case is automatically teared down, which means it is removed completely from the knowledge base.

```
protected void tearDown() {
    wrapper.consult("unconsult(\'"+tc+"\').");
}
```

The RBSLA/ContractLog distribution comes with a set of function, regression and performance test cases to verify and validate the inference implementations and meta programs of the ContractLog KR with respect to typical adequacy criteria of KR formalisms and in particular w.r.t. completeness, soundness, expressiveness and efficiency/scalability.

Test coverage measurement as described in section 4 has been also integrated into ContractLog's Java based test framework. A class *rbsla.regressiontest.TestCoverage* implements respective methods (using the ProvaWrapper) to query the coverage meta program implemented in the ContractLog KR. The meta program implements different functions to compute e.g. the minimalized substitution of two

terms, the instances of clauses under the instance order, the lgg of two clauses, the subsumption of clauses, the generalized subsumption of two clause sets, the relative generalization and the coverage as defined in section 4. The results are use to create an automated test coverage report which can be done by an Ant task.

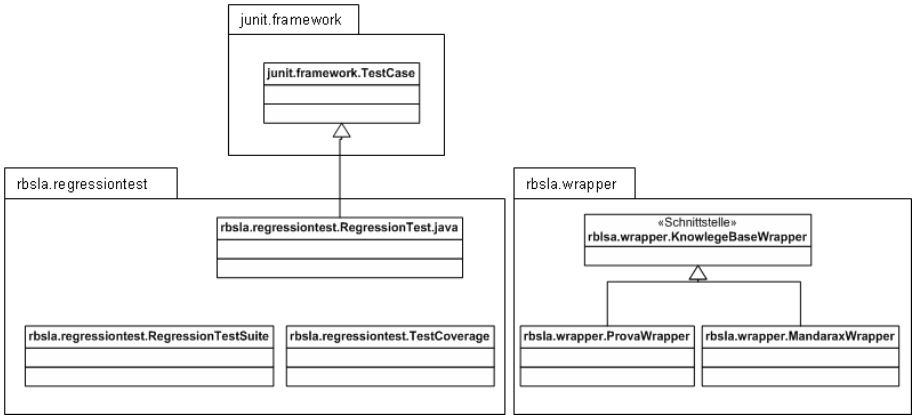

Fig. 4. UML Class Diagram for ContractLog's Test Framework

Figure 4 shows the main Java classes / interfaces of the test framework.

A Rule Based Markup Language for Test Cases

To support distributed management and rule interchange we have integrated test case and updates into RuleML (current version RuleML 0.89). The Rule Markup Language (RuleML) is a standardization initiative with the goal of creating an open, producer-independent XML/RDF based web language for rules.It provides a rich syntax for derivation rules and supports different logic classes such as datalog, hornlog (with naf), extended (with neg) disjunctive, FOL. Our design goals are to stay as close as possible to the RuleML standard, reuse the existing language constructs and fulfill typical criteria of good language design such as minimality, homogeneity, symmetry and orthogonality [130]. In particular, we try to keep the set of new language constructs as small as possible and give these constructs a plain declarative markup semantics. We follow the design principle of RuleML and define the new constructs within separated modules which are added to RuleML as additional layers, i.e. it adds additional expressiveness and modelling power to RuleML for the serialization of integrity constraints, test case, knowledge update primitives in XML. We use XML Schema group definitions to define language constructs which belong together in a group. This approach is easily extensible, i.e. it is easy to add new constructs. We first

describe relevant constructs in RuleML and then introduce our extensions regarding integrity constraints, test cases and updates. The Rule Based Service Level Agreement Language (RBSLA) [131] (see appendix B) for serialization of rule based contracts, policies and SLAs comprises several other layers extending RuleML with modelling constructs for e.g. defeasible rules, deontic norms, temporal event logics, reactive ECA rules. It also comprises the ECA-RuleML sublanguage to serialize ECA rules and complex event/actions. An overview over the complete RuleML structure (RuleML 0.89) can be found in appendix A. Here, we focus on the RuleML horn logic layer extended with negation and equality. The building blocks are [24]:

- 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 ap-plied, 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 proved 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 transformation rules.

A markup serialization syntax for test suites / test cases and knowledge updates has been implemented as an extension to RuleML [24] with the following constructs 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 ({}).:

Test Suites / Test Cases

```
TestSuite ::= [oid,]content | And
TestCase ::= [oid,]{test|Test,} [assertions | And]
Test ::=[oid,] [message | Ind,] test | Query
Integrity ::= [oid,] formula | Not | And | Or | Xor
assertions ::= And
test ::= Test | Query
message ::= Ind
```

ID based update constructs

```
Assert ::= content | And
Retract ::= content | And
```

The update constructs are defined on the level of atoms and can be used within rules, e.g. derivation rules or reaction rules. This differs from the assert construct in RuleML which is a KQML like performative defined on the top level of RuleML. 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 defined under And, e.g.:

```
<Implies @variety="strict">
    ...
    <body>
        <Assert>
            <And>
                    <oid>
                        <Ind>./examples/ContractLog/RBSLA/math.rbsla</Ind>
                    </oid>
            </And>
        </Assert>
    </body>
</Implies>
```

The examples shows an external updated defined in a strict derivation rules' body which imports an external LP script *math.rbsla* from a relative path. (@variety distinguishes "strict" and "defeasible" rules)

Example: $eca(..., update(id1, "f().p() :- f()."))$.

```
<ECA>
       ...
       <action>
             <Assert safety="transactional">
                       <And>
                           <oid><Ind>id1</Ind></oid>
                           <Atom><Rel>f</Rel></Atom>
                           <Implies>
                                  <Atom><Rel>f</Rel></Atom>
                                  <Atom><Rel>p</Rel></Atom>
                            </Implies>
                        </And>
              </Assert>
         </action>
</ECA>
```

The example shows an knowledge self-updated with the oid $id1$ adding a fact and a rule to the KB in the action part of an ECA rule. Due to the transactional mode $safety = "transactional"$, the update should be rolled back, if it is only

partially performed or violates any integrity constraint/test case defined in the KB.

The *content* of a $TestSuite$ is a list of $TestCase$s connected with *And*. A $TestCase$ consists of one or more *test*s and optional additional *assertion*s such as rules ($Implies$ or ECA) or facts ($Atom$) which are dynamically added for testing. A $Test$ defines an optional failure message which might be used to create a test report, e.g. by an test framework such as JUnit. We have decided to reuse and extend the RuleML *Query* construct to define a test query, in order to keep the language as compact as possible. A *test Query* can be used to define several test literals which are connected by *And* or *Or*. Test queries can be negated by default negation (Naf) or explicit negation (Neg). The attribute @*label* defines the expected outcome *true*, $false$ or *unkown*, where *true* is the default value. The @*semantics* attribute assigns meta information about the intended semantics of the complete test case and the tests, e.g. to define different variants of tests for different semantics of target inference engines. Default value is $semantics : WFS$. The value of the attribute might be a reference on an ontology (*semantics* is the namespace) defining different semantics, e.g. a reference on an OWL ontology which models the semantics of figure 3, section 6. The T-box model might define further subcategories (subclasses) of the root class *Semantics*, e.g. $LP\_Semantics \sqsupseteq 2-valued$ and $3-valued$. The attribute @*class* denotes the logic class of the program. The values again might be defined in an ontology which might be based on the logic classes defined in section 6, figure 2 for LPs and further extended with values forward-reasoning (production) rule programs, e.g. $class : Rete$. Similar meta information might be defined to annotate inference engines and RuleML programs in order to use such information for verification and validation of the correct execution of an (interchanged) program - as discussed in section 6.

```
<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 one test query: $test1 : \{p => true, notq => true\}$.
Test queries with lists of expected variable bindings can be serialized as sets of ground queries connected by *And*:

...

```
    <Query>
        <And>
            <Atom><Rel>p</Rel><Ind>a</Ind></Atom>
            <Atom><Rel>p</Rel><Ind>b</Ind></Atom>
            <Atom><Rel>p</Rel><Ind>c</Ind></Atom>
            ...
        </And>
    </Query>
...
```

The example shows a test query $p(X) => true : \{X/a, X/b, X/c, ...\}$ which defines a list of expected variable bindings.

8 Related Works

Verification and Validation (V&V) of knowledge base 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. [52–56]. Several methods for detecting inconsistencies in KBs based on monotonic reasoning have been proposed, such as:

- Tabular methods, e.g. [58], 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. [59, 60], 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. [61] which model the KB as Petri net and test the complete models starting with all possible initial states, which is very costly.
- Methods based on operational debugging via 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, this methods presuppose a deep understanding of the inference processes by the user to detect the inconsistencies.
- Methods based on algebraic interpretation, e.g. [62] transform a KB into an algebraic structure, e.g. a boolean algebra which is then used to verify the KB. This approach can not applied to expressive rule bases with variables, object-valued functions or meta predicates and non-monotonic negations.
- Methods based on declarative debugging 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 is reached.

Two testing approaches are presented by Boug et all. [5] and Gorlick et all. [22], which are based on a description of the test space in terms of context-free

grammars, constraint systems and algebraic specifications. Ruggieri [48] propose a specification based testing approach for LPs. Their testing problem consists of checking whether or not the formal semantics of a program include a given finite set of atoms. However, they concentrate on a special subset of LPs with decidable semantics and therefore lack generality. There are also some approaches on verifying non-monotonic rule bases such as [63] which analyzes rule bases expressed in default logic or [64] which tests rule bases with production rules. For further details concerning inconsistency checking techniques see e.g. [65–67]. Much research has been directed at the automated refinement of rule bases, e.g. [69, 68]. Work has also been done on the refinement of rule bases using test cases, e.g. [70, 68]. Work has also been done on the automatic generation of test cases, e.g. [71]. A few methods have been proposed that are able to verify systems with uncertainty, e.g. [72]. Several rule base debugging and validation tools have been developed. For an overview see e.g. [73, 74]. Test coverage of imperative programs has been intensively investigated in the past decades, e.g. [49], but there are only a few attempts addressing the test coverage measurement for test cases of rule based programs. Luo et al. [28] describe an approach which uses the hidden control flow of Prolog programs to select adequate test data which covers this control flow. However this method is only applicable if the LP can be transformed into a procedural control flow based program. Our approach exploiting ILP techniques to measure the test coverage of test cases for GLPs is more general and can be applied in a homogeneous KR where test cases, rules and LP-based meta programs for anti-unifying the instantiated programs and computing the lggs are represented in a homogeneous KB based on a single representation language. V&V of interchanged rule sets is an important task in upcoming rule interchange formats which to the best of our knowledge has not been researched at all. Using general (non-monotonic) properties of semantics which are adapted to meta test cases for V&V of inference engines and in particular for determining adequacy of the possible unknown semantics of an arbitrary inference engine is an promising novel approach, which establish trust and reliability in open distributed environments, such as the Web, where rules are interchanged between different target rule systems, e.g. inference engines provided as services on the web.

9 Conclusion and Future Work

Test cases for V&V of rule based systems (Logic Programs) are particular well-suited when rule bases grow larger and more complex and are maintained (possibly distributed) by different people. They help to capture the rule engineer's intended meaning of a LP and safeguard the evolution of the intensional knowledge base, i.e. facilitate updates and extensions of the rule base in order to adapt the rule logic to changing requirements. In this paper we have attempted to bridge the gap between the test-driven techniques developed in the Software Engineering community, on one hand, and the declarative rule based programming approach for engineering logic programs, on the other hand.

We have elaborated on an approach using LP-based test cases which use a set of test goals (queries) to describe verify and validate the intended models of LPs and have extended this test-driven approach with the notion of a test coverage measure for determining the quality of a test case. Testing methodologies delivering quality data for LP reliability models and in particular for determining the quality of test data (test coverage) have not been investigated very often. To the best of our knowledge using ILP techniques and least general generalization to compute a test coverage measure for LP testing has not been studied before. The coverage notion define in this paper is a type of structural coverage, but it is not based on control flow as common imperative measures. In fact it is more related to code-based test adequacy criteria based on data flow coverage since substitution of variables by terms, i.e. (full) instantiation of program clauses by test goals, is central to the coverage concept. Data flow criteria have been well researched for imperative languages, e.g. [18], and several empirical studies, e.g. [19], have demonstrated their usefulness in the imperative domain. Therefore, similar results might be expected in the declarative domain since the instance order of program clauses induced by test case goals via unification is the natural abstract model of a logic program.

In the context of rule interchange and validation of target inference engines we have proposed a testing methodology exploiting test cases to analyze the implementation specifics of an inference engine with respect to performance and scalability and w.r.t. general properties of the semantics implemented by the engine. The approach ensures correct execution of an interchanged LP in the target environment even in case no further meta information about the execution environment are available. This helps to establish trust to the inference service and safeguards rule interchange.

We have then extended logic programs to evolving logic programs which can be updated using expressive ID-based update primitives. The update functions allow treating rule sets as modules which can be dynamically added or removed from the KB at runtime using their object / module identifiers (oid). We have extended this approach to transactional updates which perform integrity tests on integrity constraints or test cases and rollback the updates in case the tests fail. This ensures integrity of the LP in case of dynamically changing rule sets.

The concepts introduced in the paper have been implemented within the ContractLog KR an expressive knowledge representation framework incorporating several logical formalisms into logic programming such as LP based ECA rules, deontic logic, defeasible logic, event logics, temporal logics, to name some. The implementation integrates the test-driven development into the common testing framework JUnit with support for Ant. This adds tool-support for test-driven development of logic programs.

Finally, we have introduced a concrete RuleML-based syntax for serialization of test cases in RDF or XML which facilitates e.g. rule interchange and XML related tool support.

The findings about the application of test-driven development for rule-based development and the proposed testing methodologies and techniques in our opinion

give enough reason to make a relevant contribution and simultaneously motivate further research in this field. We plan to extend our approach in several ways, relating to automated test case generation, applying automated refactorings, larger meta test suites to test logic programs and inference engines, in order to provide them as open services on the web, a deeper and more fine grained vocabulary to annotate rule engines and logic programs / test cases. Our vision is to reach a similar market maturity of test-drive development in the agile development of Logic Programs and rule-based knowledge engineering as it is available in Extreme Programming for imperative software development.

References

1. D. Angulin and C.H. Smith. Inductive inference: Theory and methods. *ACm Computing Surveys*, 15(3):237–269, 1983.
2. Patel-Schneider, Peter F., Horroks I., OWL Web Ontology Language Semantic and Abstract Syntax, `http://www.w3.org/2004/OWL`.
3. Klyne G., Caroll J.J. (Eds.), Resource Description Framework (RDF): Concepts and Ab-stract Syntax, W3C, 2004.
4. Horrocks I., Patel-Schneider P. F., Bell Labs Research, Boley H., Tabet S., Grosof B.,Dean M., SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission 21 May 2004, `http://www.w3.org/Submission/SWRL/`.
5. L. Bouge, N. Choquet, L. Fribourg, and M. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of System and Software*, 6(4), 1986.
6. K. Beck. Extreme programming. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 411, Washington, DC, USA, 1999. IEEE Computer Society.
7. K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
8. F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, Cambridge, 1995.
9. R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, New York, USA, 1979.
10. Virginia E. Barker, Dennis E. O'Connor, Judith Bachant, and Elliot Soloway. Expert systems for configuration at digital: Xcon and beyond. *Commun. ACM*, 32(3):298–318, 1989.
11. M. Comini, G. Levi, and G. Vitiello. Efficient detection of incompleteness errors in the abstract debugging of logic programs. In *AADEBUG'95*, 1995.
12. E. Duesterwald, R. Gupta, and M.L. Soffa. Rigorous data flow testing though output influences. In *2nd Irvin Softw. Symp.*, 1992.
13. J. Dietrich and H. Herre. Contributions to the theory of nonmonotonic inference systems. Technical Report 05/97, University of Leipzig, Institut für Informatik, 1997.
14. J.W. Duran and S.C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, 1984.
15. M. Ducasse and J. Noye. Logic programming environments: dynamic program analysis and debuggin. *Journal of Logic Programming*, 19:351–384, 1994.

16. J. Dietrich and A. Paschke. On the test-driven development and validation of business rules. In Roland Kaschek, Heinrich C. Mayr, and Stephen W. Liddle, editors, *Information Systems Technology and its Applications, 4th International Conference ISTA'2005, 23-25 May, 2005, Palmerston North, New Zealand*, volume 63 of *LNI*, pages 31–48. GI, 2005.
17. Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
18. P. Frankl and E. Weyuker. An applicable family of data flow criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, 1988.
19. P. Frankl and E.J. Weyuker. An experimental comparison of the effectiveness of branch testing and data fow testing. *IEEE Trans. Softw. Eng.*, 19(8):774–787, 1993.
20. J. Gannon. Testing tools using formal specification and coverage metrics. *GI Softwaretechnik-Trends*, 6(1):5–11, 1986.
21. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
22. M. Gorlick, C.F. Kesselmann, D. Marotta, and D. Stott Parker. Mockingbird: A logical methodology for testing. *Journal of Logic Programming*, 8(1,2), 1990.
23. E. Harold. An early look at JUnit 4. `http://www-128.ibm.com/developerworks/java/library/j-junit4.html`.
24. Boley H., Wagner G., Tabet S. and Antoniou G., The Abstract Syntax of RuleML - Towards a General Web Rule Language Framework, Rule Markup Initiative (RuleML), Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI'04). Beijing, China. September 20-24, 2004. pp. 628-631.
25. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow and controlflow-based test adequacy criteria. In *16th Int. Conf. Softw. Eng.*, pages 191–200, 1994.
26. Sarit Kraus, Daniel Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2):167–207, 1990.
27. Lucas Layman. Empirical investigation of the impact of extreme programming practices on software projects. In John M. Vlissides and Douglas C. Schmidt, editors, *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 328–329, 2004.
28. 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.
29. N. Lavrac and Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.
30. J.-L. Lassez, M. Maher, and K. Marriott. *Unification revisited*. Foundations of Deductive Databases and Logic Programming. 1988.
31. David Makinson. General theory of cumulative inference. In *Proceedings of the 2nd international workshop on Non-monotonic reasoning*, pages 1–18, New York, NY, USA, 1989. Springer-Verlag New York, Inc.
32. S.H. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *5th Int. Conf. on Machine Learning*, pages 339–352, San Mateo, CA, 1988. Morgan Kaufmann.
33. M. Marre and A. Bertolino. Reducing and estimating the cost of test coverage criteria. In *18th Int. Conf. Softw. Eng.*, pages 486–494, 1996.

34. Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
35. John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
36. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1(2):303–338, 1990.
37. L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10:255–285, 1992.
38. S.H. Nienhuys-Cheng and Wolf. R. *Foundations of Inductive Logic Programming*. Springer, Berlin, 1997.
39. S.H. Niblett. A study of generalisation in logic programs. In *Third European Working Session on Learning*, pages 131–138, London, 1988. Pitman.
40. F. Pfenning. Unification and anti-unification in the calculus of constructions. In *6th LICS*, pages 74–85, 1991.
41. G.D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5, 1970.
42. G.D. Plotkin. A further note on inductive generalization. *Machine Intelligence*, 6:101–124, 1971.
43. R.E. Prather and J.P. Myers. The path prefix software testing strategy. *IEEE Trans. Softw. Eng.*, 13(7):761–766, 1987.
44. L. Pottier. Generalisation de termes en theorie equationelle - cas associatif-commutatif. Technical report, INRIA Technical Report, 1989.
45. Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
46. R. Reiter. A logic for default reasoning. In *Readings in nonmonotonic reasoning*, pages 68–93. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.
47. M. Richter. *Prinzipien der Knstlichen Intelligenz*. Teubner Verlag, 1989.
48. S. Ruggieri. Decidability of logic program semantics and applications to testing. In H. Kuchen and S.D. Swierstra, editors, *Programming Languages: Implementation, Logics and Programs, 8th Int. Symposium (PLILP 96)*, volume LNCS, pages 347–362, Aachen, Germany, 1996. Springer.
49. S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.
50. Alfred Tarski. *On the concept of logical consequence, in: Logic, Semantics, Metamathematics. Papers from 1923 to 1938 by Alfred Tarski.* Clarendon Press, Oxford, 1956.
51. E.J. Weyuker. More experience with dataflow testing. *IEEE Trans. Softw. Eng.*, 19(9), 1993.
52. D.L. Nazareth. Issues in the verification of knowledge in rule-based systems. *Int. J. Man-Machine Studies*, 30, p. 255-271, 1989.
53. T.A. Nguyen. Verifying consistency of production systems. *IEEE Conf. on AI Applications*, Orlando, Florida, 1987, p. 4-8.
54. A.D. Preece. A new approach to detecting missing knowledge in expert system rule bases. *Int. J. of Man-Machine Studies*, 38, 1993, p. 161-181.
55. M.C. Rousset. On the consistency of knowlege bases: the COVADIS system. *Computational Intelligence*, 4, 1988, p. 166-170.
56. M. Suwa, A.G. Scott, E.H. Shortliffe. Completeness and consistency in a rule based system. In: B.G. Buchanan, E.H.Shortliffe,*Rule Based Expert Systems*, Addison Wesley, 1985, p. 159-170.

57. W.-T. Tsai, R. Vishnuvaijala, D. Zang. Verification and Validation of Knowledge-Based Systems. *IEEE Transactions on Knowledge and Data Engineering*,11(1), 1999, p. 202-212.
58. W. Van Melle, H. Shortliffe, G. Buchanan. EMYCIN: A Knowledge Engineer's Tool for Constructing Rule-Based Expert Systems. *Rule Based Expert Systems*, Addison-Wesley, 1984, p.301-313.
59. D.L. Nazareth, M.H. Kennedy. Static and dynamic verification, Validation and testing: The Evolution of a Discipline. *AAA'90 on Knowledge-based Systems Validation*, Verification and Testing, Boston, 1990.
60. M. Ramaswamy, S. Sarjar, C. Ye Sho. Using directed hypergraphs to verify rule-based expert systems. *IEEE TKDE*, 9(2), 1997, p. 221-237.
61. X. He, W.C Chu, H. Yang, S.J.H. Yang. A new Approach to Verify Rule Based Systems using Petri Nets. *Int. Conf. on Computer Software and Applications*, Los Alamitos, CA, USA, 1999, p. 462-467.
62. L.M. Laita, E. Roanes-Lozano,L. de Ledema, V. Maojo. Computer Algebra based Verification and knowledge extraction in RBS application to Medical Fitness criteria. *EUROVAD'99*, 1999.
63. G. Antoniou. Verification and Correctness Issues for Nonmonotonic Knowledge Bases. *Int. J. of Intelligent Systems*, 12(10), 1997, p. 725-739.
64. C.H. Wu, S.J. Le. Knowledge Verification with an Enhanced High-Level Petri-Net Model. *IEEE Expert*, 1997, p. 73-80.
65. M. Ayel, J.P. Laurent. Validation, Verification and Testing of Knowledge Based Systems. Jon Wiley and Sons, England, 1991.
66. F.P. Coenen, T. Bench-Capon. Maintenance of Knowledge-Based Systems: Theory, Techniques and Tools. Academic Press, London, 1993.
67. A.I. Vermesan, T. Bench-Capon. Techniques for the Verification and Validation of Knowledge-Based Systems: A Survey Based on the Symbol/Knowledge Level Distinction. *Software Testing, Verification and Reliability*, Vol5 no.4, 1995, p.233-272.
68. S. Craw, D. Sleeman. Automating the Refinement of KBS. *Proceedings ECAI'90*, 1990.
69. F. Bouali, S. Loiseau,M-C. Rousset. Verification and Revision of Rule Bases. *in: J. Hunt,R. Miles(Eds.)*, Reserach and Development in Expert System, SGES publication, p 253-264.
70. P. Meseguer. Expert System Validation Through Knowledge Base Refinement. *IJCAI'93*, 1993.
71. C.L. Chang, J.B. Combs, R.A. Stachowitz. A Report on the Expert Systems Validation Associate (EVA). *Expert Systems with Applications*, Vol.1,No.3,p.219-230.
72. D.E. O'Leary. Verification of uncertain knowledge based systems: an empirical verification approach. *Management Science* 42(12), 1996, p. 1663-1675.
73. A. Preece. Methods for Verifying Expert System Knowlege Bases. 1991.
74. R.T. Plant. Tools for Validation&Verification of Knowledge-Based Systems 1985-1995. Internet Source.
75. A. Paschke. ContractLog - A Logic Framework for SLA Representation, Management and Enforcement. IBIS, TUM, Technical Report, 07/2004.
76. ANSI/IEEE Standard 729. Standard Glossary of Software Engineering Terminology. 1983.
77. Adrion W. Validation, verification and testing of computer software. *ACM Computer Surveys*, vol. 14, No. 2, 1982, pp. 159-192.
78. A.J. Gonzales, V. Barr. Validation and verification of intelligent systems. *Journal of Experimental and Theoretical AI*. 2000.

79. A.D. Preece and Shinghal R. Foundations and applications of Knowledge Base Verification. *Int. J. of Intelligent Systems*. Vol. 9, pp. 683-701, 1994.
80. F. Coenen and T. Bench-Capon. Maintenance of Knowledge-Based Systems: Theory, Techniques and Tools. *Academic Press*, London, 1993.
81. A. Aiken, J. Widom and Hellerstein. Behaviour of Database Production Rules: Termination, Confluence and Observable Determinism. *Int. Conf. on the Management of Data*, ACM SIGMOD, pp. 59-68.
82. A.P. Karadimce and Urban, S.D. Refined Trigger Graphs: A Logic-Based Approach to Termination Analysis in an Active Object-Oriented Database Management System. *iCDE'96*, 1996. pp. 384-391.
83. A. Paschke. Typed Hybrid Description Logic Programs with Order-Sorted Semantic Web Type Systems based on OWL and RDFS, Internet Based Information Systems, Technical Uni-versity Munich, Technical Report 12/05, 2005.
84. A. Paschke, J. Dietrich and H. Boley. W3C RIF Use Case: Rule Interchange Through Test-Driven Verification and Validation. *http* : $//www.w3.org/2005/rules/wg/wiki/Rule_Interchange_Through_Test-Driven_Verification_and_Validation$, 2005.
85. Apt, K. and Bol, R. Logic Programming and Negation: A Survey. Journal of Logic Programming 19,20, 9-71, 1994.
86. J. Minker. An Overview of Nonmonotonic Reasoning and Logic Programming. Journal of Logic Programming, Special Issue, 17, 1993.
87. J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In Andre Fuhrmann and Hans Rott, editors, Logic, Action and Information – Essays on Logic in Philosophy and Artificial Intelligence, pages 241–327. DeGruyter, 1995.
88. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. IEEE Conference on Computational Complexity, pages 82–101, Ulm, Germany, 1997.
89. J.S. Schlipf. Complexity and undecideability results for logic programming. AMAI 15(3-4): pp. 257-288, 1995.
90. Dowling, W. and H. Gallier, Linear time algorithms for testing the satisfi-abiltiy of propositional horn formulae. Journal of Logic Programming, 1984. 1: p. 267-284.
91. Kolaitis, P. and C. Papadimitriou. Why not negation by fixpoint? in Prod. of PODS-87. 1987.
92. Marek, W. and M. Truszcynski, Autoepistemic logic. Journal of the ACM, 1991. 3(38): p. 588-619.
93. Van Gelder, A., K. Ross, and J. Schlipf, The Well-Founded Semantics for General Logic Programs. JACM, 1991. 38(3): p. 620-650.
94. Gelder, A.v. The Alternating Fixpoint of Logic Programs with Negation. in Proc. PODS'89. 1989.
95. Schlipf, J., The Expressive Powers of Logic Programming Semantics. J. Computer and System Sciences, 1995. 51(1): p. 64-86.
96. Trnlund, S.A., Horn clause computability. BIT, 1977. 17: p. 215-216.
97. Dantsin, E. and A. Voronkov. Complexity of query answering in logic databases with complex data. in LFCS'97. 1997: Springer LNCS.
98. J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Properties," Fundamenta Informaticae XXII(3) pp. 227-255, 1995.
99. J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Properties. Fundamenta Informaticae, XXII(3):257-288, 1995.

100. K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, Foundations of Deductive Databases, chapter 2, pages 89148. Morgan Kaufmann, 1988.
101. K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, Logic and Data-Bases, pages 293322. Plenum, New York, 78.
102. Kenneth Kunen. Negation in Logic Programming. Journal of Logic Programming, 4:289308, 1987.
103. 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 591602. San Mateo, CA, Morgan Kaufmann, 1992.
104. John S. Schlipf. Formalizing a Logic for Logic Programming. Annals of Mathematics and Artificial Intelligence, 5:279302, 1992.
105. Jianhua Chen and Sukhamay Kundu. The Strong Semantics For Logic Programs. In Z.W. Ras and M. Zemankova, editors, Proceedings of the 6th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, 1991, pages 490499. Springer, Lecture Notes in Artificial Intelligence 542, 1991.
106. Yong Hu and Li Yan Yuan. Extended Well-Founded Model Semantics for General Logic Programs. In Koichi Furukawa, editor, Proceedings of the 8th Int. Conf. on Logic Programming, Paris, pages 412425. MIT, June 1991.
107. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, 5th Conference on Logic Programming, pages 10701080. MIT Press, 1988.
108. J. Dix and M. Mueller. Partial Evaluation and Relevance for Approximations of the Stable Semantics. In Z.W. Ras and M. Zemankova, editors, Proceedings of the 8th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, 1994, pages 511520. Springer, Lecture Notes in Artificial Intelligence 869, 1994.
109. L.M. Pereira, J.J. Alferes, and J.N. Aparicio. Adding Closed World Assumptions to Well Founded Semantics. In ICOT, editor, Fifth Generation Computer Systems, Proceedings of the Conference, pages 562569. OMSHA, June 1992.
110. D. Sacca and C. Zaniolo. Partial models and Three-Valued Models in Logic Programs with Negation. In Anil Nerode,Wiktor Marek, and V. S. Subrahmanian, editors, Logic Programming and Non-Monotonic Reasoning, Proceedings of the first International Workshop, pages 87104. Washington D.C, MIT Press, July 1991.
111. Jia-Huai You and Li-Yan Yuan. Three-valued Formalization of Logic Programming: is it needed. In Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,, pages 172182. ACM Press, 1990.
112. P. M. Dung. Negation as Hypotheses: An Abductive Foundation for Logic Programming. In Koichi Furukawa, editor, Proceedings of the 8th Int. Conf. on Logic Programming, Paris. MIT, June 1991.
113. Chitta Baral, Jorge Lobo, and Jack Minker. Generalized Well-founded Semantics for Logic Programs. In M. E. Stickel, editor, 10th International Conference on Automated Deduction, LNAI 449, subseries LNCS, pages 102116. Springer, J. Siekmann, July 1990.
114. Melvin Fitting. A Kripke-Kleene Semantics of logic Programs. Journal of Logic Programming, 4:295312, 1985.
115. Przymusinski, T.C., "Perfect Model Semantics", Proc. 5th Int. Conf. and Symp. on Logic Pro- gramming, MIT Press, Cambridge, Ma, 1988, pp. 1081-1096.
116. H. Przymusinska and T. Przymusinski, Weakly Perfect Semantics for Logic Programs, Proceedings of the 5th International Conference and Symposium on Logic Programming 1106-1121, August, 1988.

117. A. Rajasekar, J. Lobo, and J. Minker. Weak generalized closed world assumption. Journal of Automated Reasonig, 5(3): pp. 293-307, 1989.
118. S. Brass and J. Dix. Characterizations of the Disjunctive Wellfounded Semantics: Confluent Calculi and Iterated GCWA. Journal of Automated Reasoning, 1997.
119. Teodor C. Przymusinski. Stable semantics for disjunctive programs. New Generation Computing, 9:401–424, 1991.
120. C. Baral, J.Lobo, and J.Minker. Generalized disjunctive well-founded semantics for logic programs. Annals of Math and Artificial Intelligence, 5:89-132, 1992.
121. J.C. Shepherdson. Unsolvable problems for SLDNF resolution. Journal of Logic Programming, pages 19–22, 1991.
122. J. Dix. Classifying Semantics of Disjunctive Logic Programs. In K.R.Apt (editor), Int. Conf. and Sym. of Logic Programming, pp. 798-812, Camebridge, Mass., Nov. 1992, MIT Press.
123. Paschke, A.: ECA-RuleML: An Approach combining ECA Rules with interval-based Event Logics and Event Notifications, IBIS, Technische Universitt Mnchen, Technical Report 11 / 2005.
124. Paschke, A. : ECA-LP: A Homogeneous Event-Condition-Action Logic Programming Language, IBIS, Technische Universitt Mnchen, Technical Report 11 / 2005.
125. Flesca, S. and S. Greco. Declarative Semantics for Active Rules. in Int. Conf. on Database and Expert Systems Applications. 1998: Springer LNCS
126. Bowen, K.A. and R.A. Kowalski., Amalgamating language and meta-language in logic programming. Logic programming, 1982: p. 153-172.
127. M.J. Maher, A. Rock, G. Antoniou, D. Billington, T. Miller, "Efficient defeasible reasoning systems," ictai, p. 0384, 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00), 2000.
128. Chakravarthy, S., et al. Composite Events for Active Databases: Semantics Contexts and Detection. in VLDB 94. 1994.
129. Beck, K,; Gamma, E.: The JUnit Cookbook. http://junit.sourceforge.net/doc/cookbook/cookbook.htm
130. Codd, E. 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. 1971. San Diego, CA.
131. 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.

Appendix A
Appendix B

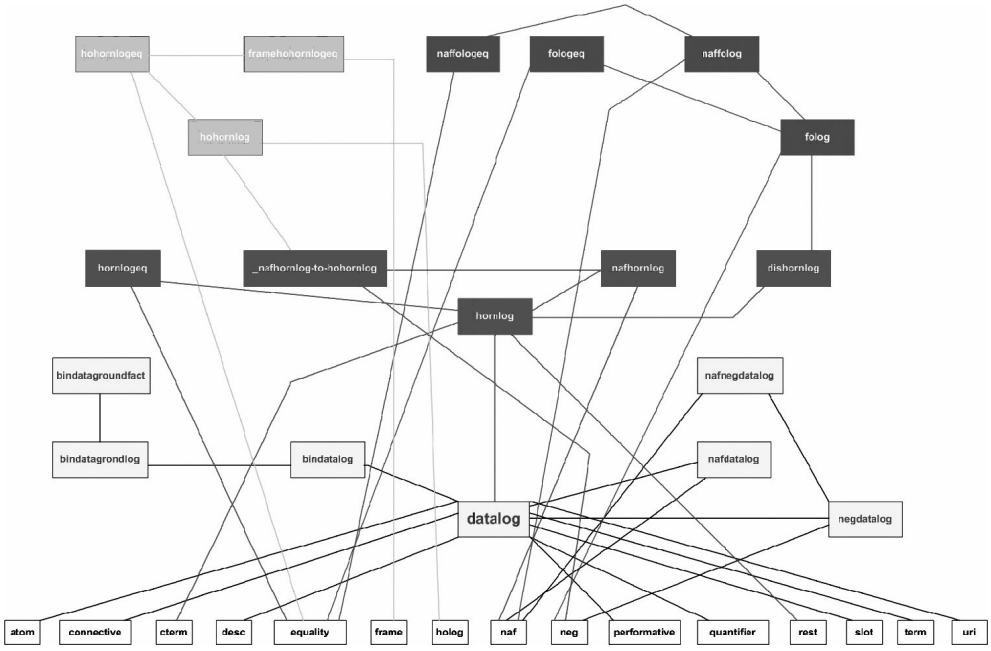

Fig. 5. RuleML 0.89

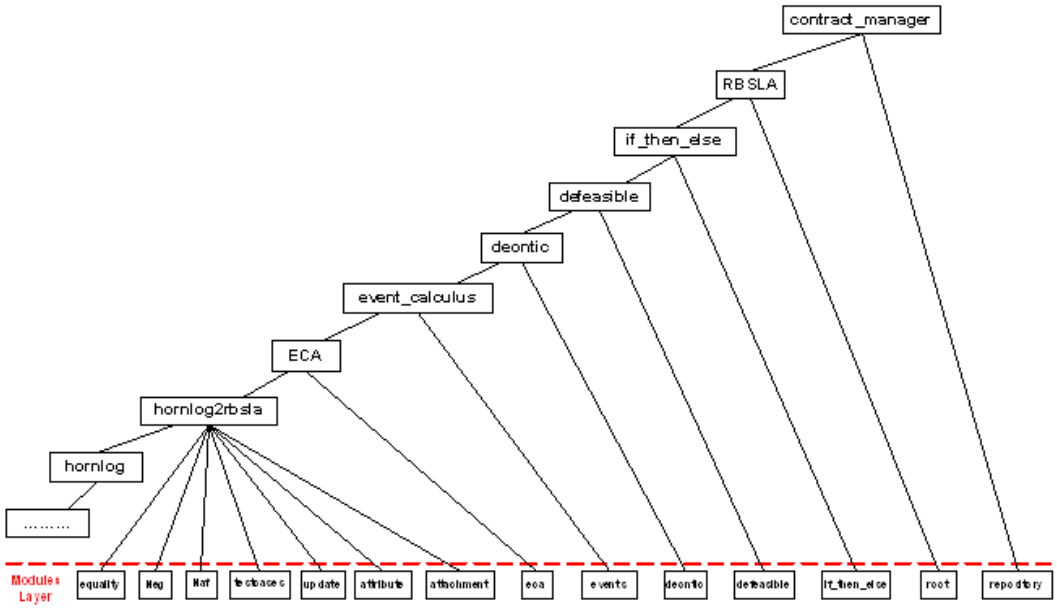

Fig. 6. RBSLA 0.1