On Self-Validating Rule Bases

A. Paschke[1], J. Dietrich[2], A. Giurca[3], G. Wagner[3], S. Lukichev[3]

[1]Internet-based Information Systems, Dept. of Informatics, TU Munich, Germany
Adrian.Paschke@gmx.de
[2]Institute of Information Sciences and Technology, Massey University, New Zealand
J.B.Dietrich@massey.ac.nz
[3]Institute of Informatics, BTU Cottbus, Germany
{Giurca,G.Wagner,Lukichev}@tu-cottbus.de

Abstract. In this paper, we analyze a recent trend in software engineering (SE), test driven development, and discuss how it can be adapted to define self validating rule bases. We argue that test cases can be used to specify the semantics of rules, and that the presence of these test cases safeguards the life cycle of rules. We introduce an abstract conceptual framework, in the tradition of Tarski, that allows us to cover a wide range of adequate logics for rule-based representation. We investigate how the concept of test coverage can be adapted to quantify the quality of test cases. We discuss the implementation of these ideas, which includes a discussion on how Semantic Web rule languages can be extended to serialize self validating sets of rules.

Key words: Verification and Validation (V&V), Test-driven Development, Test Cases, Test Coverage, Software Engineering

1 Introduction: Lessons learned from Extreme Programming

Extreme programming [2, 1] and similar approaches to "agile" software engineering have been very successful in recent years. Superficially, extreme programming seems to neglect formal analysis and design, negating established wisdom that only thorough formal modeling (particularly the adoption of process models such as the rational unified process RUP and modeling languages such as UML) can lead to high quality software delivered within time and cost estimates. However, the experiences made in the last decade support the fact that extreme programming does work well for small and medium sized projects and leads to better software quality and customer satisfaction than heavyweight approaches [10]. One of the key propositions of agile software engineering is test driven development: whenever new features are added, test cases are written first. Test cases are written in the target programming language. A certain code infrastructure is required to write those test cases, this is facilitated by programming language features such as abstract classes and interfaces, and common design patterns such as Factory [6]. The combination of interfaces and test cases is *the*

model produced in extreme programming, and much of the success of extreme programming can be attributed to the fact that these models are semantically richer than the models produced using modeling techniques like UML.

The main outcome of UML modeling are visual models describing the interaction of classes, their instances and other artifacts. The internal behavior is somehow constraint by those visual models. For instance, the classes that can be referenced by a class are restricted by the associations modeled, and additional constraints can be added using the object constraint language (OCL) and narrative descriptions. In practice, OCL is rarely used for various reasons. Firstly, people in charge of domain modeling are often reluctant to use any formal language. Secondly, tool support (editors and runtime validation tools) for OCL is poor. Thirdly, many constraints apply only in a certain context (for instance, in a certain network environment), and it is difficult or impossible to make assertions about such an environment in OCL. Constraints expressed in natural language are weak by nature as they cannot be automatically validated.

Using the programming language to write the constraints which describe the intended model gives software engineers a more expressive modeling language, and makes programs self validating: by definition, the program 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 software incorrect with respect to the new set of test cases. The program is then modified until the old and the new test cases succeed. Test cases are significantly simpler than the code they describe: they represent a *black box* view on the program. Test cases are an abstraction from programs, and there are many possible programs validating against a given set of test cases. This makes test cases models. For instance, to describe the semantics of a complex arithmetic algorithm that computes integers from given integers, only pairs of integer numbers have to be provided. While there is no guarantee that this description is complete, test case driven development defines a *process* that will eventually approach a complete description. While test cases are usually written by software engineers, they can easily be communicated to domain experts. Yet another advantage is that test cases do not use artifacts that have no direct counterpart in the programming language.

There are several other technologies to facilitate test driven development. Test coverage metrics can be used to quantify the completeness of test cases. Most metrics and tools are based on the idea that tests should visit all branches of the source code tree (AST). In our terminology, coverage measures the quality of the model. In the arithmetic example used above, test coverage metrics would measure whether each IF and each ELSE branch in the algorithms at least visited once when the test cases are executed. Newer test frameworks like JUnit 4 [8] facilitate a very tight integration of tests into code, particularly employing annotations. The model becomes an *aspect* of the software. The problem of increasingly complex system environments that are necessary to perform tests is addressed by mock object frameworks: proxies simulating the aspects of the environment relevant to perform a certain test.

2 On Testing Rules

Rule based systems have been investigated comprehensively in the realms of declarative programming and expert systems over the last two decades. Using rules has several advantages: reasoning with rules is based on a semantics of formal logic, usually a variation of first order predicate logic, and it is relatively easy for the end user to write rules. The basic idea is that users employ rules to express *what* they want, the responsibility to interpret this and to decide on *how* to do it is delegated to an interpreter (e.g., an inference engine or a just in time rule compiler). In recent years rule based technologies have experienced a remarkable come back namely in two areas: business rule processing, and reasoning in the context of the semantic web.

The first trend is caused by the need to accelerate the slow and expensive software development life cycle. The vision of treating program logic as data is particularly interesting for businesses with rapidly changing business logic, like the telecommunication industry (which has been subject to deregulation in many countries and has become very competitive in the last few years), insurance, and investment banking.

The second trend is related to the semantic web initiative of the W3C. New standards such as RDF (http://www.w3.org/RDF/) and OWL (http://www.w3.org/2004/OWL) aim to turn the web into a huge database of cross referenced, machine processable knowledge, and rules can be used to extract and process this knowledge in a platform independent manner. Emerging standards for rules operating in the context of the semantic web include RuleML (http://www.ruleml.org/) and SWRL (http://www.w3.org/Submission/SWRL/).

A general advantage of using rules is that they are usually represented in a platform independent manner, often using XML. This fits well into nowadays distributed, heterogeneous system environments. Rules represented in standardized formats can be discovered and invoked at runtime, and interpreted and executed on any platform. The weakness of this approach is the assumption that rules are easy to understand for users (both end users or average software engineers). It appears that rule systems that are useful for practical purposes are of significant complexity. The first source of complexity is simply quantity. There are deployed expert systems with more than 10000 rules [3]. Secondly, there is complexity caused by the structure of the rule languages used. Factors contributing to the complexity of rules include:

1. The number of primitives in the rule language (e.g., number of connectives used).
2. The depth of the syntax tree (e.g., the depth is restricted for logic programs without function symbols, but unrestricted if connectives and terms can be nested).
3. Restrictions in the rule language (e.g., no negations in rule heads).
4. Non-standard language elements and procedural elements (e.g., priorities, cut, different flavors of logical and procedural conjunctions as used in Java and similar languages, procedural attachments).
5. Different language elements with a similar meaning (e.g., weak and strong negation, OR and XOR).

6. Polymorphic language elements (e.g., one negation that is interpreted as a weak or strong negation depending on the predicate symbol of the negated atom).
7. The lack of a standard interpretation for certain language elements. In particular, this is a problem in many flavors of modal logic, or for logic using negation as failure. Even in the academic community there is no consensus on issues like what the canonical semantics for negation as failure or deontic modalities is. Therefore, it can not be expected from the end user to fully comprehend the effects certain rules may have.
8. Cross-references between rules. (e.g., loops in the dependency graph between predicate symbols).

These conditions can easily be checked, and can be considered as a suite of complexity metrics that can be used to quantify and compare the complexity of rule languages. There is an obvious tradeoff between simplicity and expressiveness of rule languages. On the other hand, simplicity should be seen as a primary design goal for languages in order to make them accessible for a wide audience. This resembles the requirement for code simplicity in software engineering measured with metrics such as cyclomatic complexity [12], or the requirement for simplicity of text measured using readability tests like Flesch-Kincaid. We propose to address this problem by separating the two roles rules have - the specification of an intended model and the implementation of this model.

Rules define a set of intended models by selecting models from a set of possible models. Model is here used in a very abstract sense as a set of entities (worlds), following Tarski's tradition. This includes true-false mappings as models for classical propositional logic (CPL), the models used in predicate logic, and Kripke style models for modal logic. For instance, the presence of the single rule $A \rightarrow B$ restricts models as follows:

$$\Sigma(M_0, \{A \rightarrow B\}) = \{m \in M_0 | m \models A \Rightarrow m \models B\}$$

Σ is the model selection function that defines intended models. Model selection functions have been comprehensively studied in the area of non-monotonic reasoning[13, 11, 9]. But as discussed before, we do not believe that rules are suitable means for users to describe their intended models. Hence, we are looking for alternatives to describe these models which are significantly simpler than rules. Tests can help here. In analogy to test cases in agile software engineering, we consider a *test* to consist of two parts: a set of assertions that sets up a test environment, and an expected outcome. The set $X \subseteq L$ consists of formulas from L, it is the *fact base* of the test. The *expected outcome* of the test is a formula $A \in L$ plus a label. The label can be either $+$ or $-$. If the label is $+$ then the test case is called *positive*, if the label is $-$ then the label is called *negative*. A set of tests is called a *test suite*. Let Mod be the function that associates sets of formulas with sets of models, and let Σ be a model selection function. We define a compatibility relation $\models_{TC}$ between Σ and test cases as follows:

$$M_0 \models_{TC} (X, A, +) \text{ iff } \forall m \in M_0 : m \in \Sigma(Mod(X), R) \Rightarrow m \in Mod(A)$$
$$M_0 \models_{TC} (X, A, -) \text{ iff } \exists m \in M_0 : m \in \Sigma(Mod(X), R) \Rightarrow m \notin Mod(A)$$

We call a set of models compatible with a test suite iff it is compatible with all tests in the test suite. There can be different model selection functions that are compatible with the same test suite, checking compatibility ensures that the selection function meets the constraints defined by the test cases. Executing a test case means to check the following conditions: $A \in C_R(X)$ for positive test cases $(X, A, +)$, and $A \notin C_R(X)$ for negative test cases $(X, A, -)$, respectively. $C_R(X)$ is the deductive closure of X. Here we assume that the semantics consisting of Σ and Mod is equivalent to an inference operator C_R that is based on formal proofs. This is, that completness and correctness can be shown, and that C_R is decidable.

Tests are inherently simple as they represent a *black box view* on rule base system: no rules have to be created in order to write tests. Furthermore, we can assume that only a sublanguage of the rule language is used in tests. For instance, if the logic used is a flavor of predicate logic, test cases should neither contain function symbols nor variable symbols, neither in the fact base nor in the expected outcome (the query). The complexity conditions listed above make this relative simplicity quantifiable.

Like rules, tests are constraints on the set of possible models and therefore describe an approximation of the intended model(s). We do not expect many situations where tests describe exactly one model. But the approximation defines the quality of the rules - to which degree they are supposed to approximate the intended model(s). Automated validation against a test suite checks whether $\Sigma(Mod(X), R)$ is compatible with the test suite.

The experience with extreme programming has shown that it is not only important to use test cases to automatically validate software, but to define a process that ensures that the quality of validation and software is permanently improved. This reflects two important issues which equally apply to rules-based systems: firstly, the knowledge of users about what is to be modeled is usually incomplete and the iterative process supports the acquisition of this knowledge. Secondly, the realities of project management and budgeting often do not allow the complete specification of the problem. Modeling is only done as thoroughly as possible under the current circumstances, and empirical data shows that that a good approximation can be achieved with a rather small investment. In software engineering, this situation is often described by the famous Pareto principle of *80-20 rule*.

One particularly interesting use case for automated validation through tests is the refactoring of rule bases [4]. In analogy to refactoring in software engineering [5], refactoring aims at improving the structure but retaining the behavior. In particular, this is achieved by simplifying rules, or removing redundancies from rules. For rules, refactoring can be defined as follows: in a strict sense, a refactoring is a transformation of sets of rules that satisfies the condition $\Sigma(M_0, R_0) = \Sigma(M_0, refact(R_0))$ for all $M_0 \in 2^M$ and $R_0 \in 2^R$. I.e., refactorings preserve the intended models. In the presence of a test suite ts, a mapping $refact$ is called a refactoring with respect to ts iff the following holds: if $\Sigma(Mod(X), R)$ is compatible with ts then $\Sigma(Mod(X), refact(R))$ is also compatible with ts.

I.e., refactorings with respect to a set of test cases may change the intended model, but the intended model of refactored rule base satisfies the constraints defined by the test suite. Note that we do not consider refactorings that change the test suite itself, such as renaming of predicate symbols. There are some results in the context of logic programming on refactorings in the strong sense [15].

In the next section we focus on one of the prominent representatives of rule KRs, namely declarative logic programming, and refine the test-driven approach with the notion of test coverage to assess the quality and completeness of test cases for logic programs (LPs).

3 Measuring the Quality of Tests for Logic Programs

Following the definition of the abstract test framework introduced in the last section, a test case T for a NLP P consists of test input assertions being the set of temporarily asserted test input facts (or meta test rules) X, one or more test queries $Q_n? : q_n(t_1, ..t_n)?$ where $q_n \in rule(P)$, $n > 0$ and $rule(P)$ is the set of literals (positive or negative atoms) being the heads of rules, since only rules need to be tested, and the intended possibly empty output result sets A_n for each test query $Q_n?$ with either a positive $+$, negative $-$ or unknown $?$ label (to support 3-valued semantics e.g. well-founded semantics). The expected output result set A_n for a test query $Q_n?$ and a test input X on a program P is the set of specializations of $Q_n?$, i.e. the variable bindings of the computed answers derived e.g. by SLD(NF)-refutations of $Q_n? \cup P$. A_n might be empty either because the test query is ground and hence the test case only succeeds if the answer of the query corresponds to the label or because the free test query fails, i.e. no answers can be derived for the variables and the test case fails in case of a positive expected answer label. Accordingly, a test case for a NLP P is a triple $T_P = \{X, Q_n, A_n\}$ where $n > 0$.

An important task in test-driven validation of rule bases is test coverage determination to evaluate the quality of the actual test cases and improve it in an iterative development process. The coverage feedback highlights aspects of the rule program which may not be adequately tested and which require additional testing. This loop will continue until coverage of the intended models meets an adequate approximation level by the test cases / test suites. In a nutshell, test coverage is vital to know how well the tests actually test the rule code, to know whether there has been enough testing in place and to maintain the test quality over the lifecycle of the rules. However, differences between the declarative logic programming and imperative (procedural) programming paradigm directly impact the development of a test coverage measure. Conventional testing methods for imperative languages rely on the control flow graph as an abstract model of the program or the explicitly defined data flow and use coverage measures such as branch or path coverage. This is not directly applicable in logic programming (LP). Here the procedural semantics is based on resolution with backtracking and unification, i.e. no explicit control flow exists and the data flow due to the

globally defined rules and the central concept of unification is different from an imperative program. Hence, coverage measure for a LP should take this unification based execution model into account where test goals (queries) are used to specialize the rules, leading to specializations of these rules . Using the goal reductions as sub-goals for further derivations leads to more specific specializations on the next level and to an specialization order $(H \leftarrow B) \geq (H \leftarrow B)' \geq ...$, whereas $\geq$ denotes the level relation "more general as". Accordingly, *a test covers a logic program P, if the test queries (goals) lead to a least general specialization of each rule in P, such that the full scope of terms of each literal in each rule is investigated by the test queries.*

Inductively deriving general information from specific knowledge is a task which is approached by inductive logic programming (ILP) techniques which allow computing the least general generalization (lgg) i.e. the most specific clause (e.g. w.r.t. theta subsumption) covering two input clauses. A lgg is the generalization that keeps an anti-unified term t as special as possible so that every other generalization would increase the number of possible instances of t in comparison to the possible instances of the lgg. Basically, anti-unification works as follows: It takes two terms as input and the anti-unification algorithm returns a list containing the lggs of the terms and the bindings to transform each input-term into a lgg. The algorithm can fail only in the case the top-symbols (predicate name/functor) or the length of the two terms are different (number of arguments). Efficient algorithms based on syntactical anti-unification with θ-subsumption ordering for the computation of the (relative) lgg(s) exist and several implementations have been proposed in ILP systems such as GOLEM, or FOIL. θ-subsumption introduces a syntactic notion of generality: A rule (clause) r (resp. a term t) θ-subsumes another rule r', if there exists a substitution θ, such that $r \subseteq r'$, i.e. a rule r is *as least as general as* the rule r' ($r \leq r'$), if r θ-subsumes r' resp. *is more general than* r' ($r < r'$) if $r \leq r'$ and $r' \not\leq r$. (see e.g. [14]) Using the concepts of θ-subsumption and least general generalization we now refine our initial test coverage notion. In order to determine the level of coverage the specializations of the LP rules on the top level are computed via specializing the rules with the test queries by standard unification. Then via generalizing these specializations under θ-subsumption ordering, i.e. computing the lggs of all successful specializations, a reconstruction of the original LP is attempted. The number of successful "recoverings" then gives the level of test coverage, i.e. the level determines those statements (rules) in a LP that have been executed/investigated through a test run and those which have not. In particular, if the complete LP can be reconstructed via generalization of the specialization then the test 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 denoting variants of clauses/terms and the r_i' are the specializations of r_i by a query $Q_i \in T$. It is a cover for a program P, if T is a cover for each rule $r_i \in P$. With this definition it can be determined

whether a test covers a LP or not. The coverage measure for a LP P is then given by the number of covered rules r_i divided by the number k of all rules in P, 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 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).
mother(Y,X):-son(X,Y),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 $T = \{father(uwe, adrian)?, father(hans, uwe)?, son(adrian, uwe), son(uwe, hans)?\}$ be a test with four test queries. The set of specializations 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).
son(adrian,uwe):-parent(uwe,adrian),male(adrian),male(uwe).
```

The lggs are:

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

Accordingly, the first two rules are covered and the overall coverage is 50%. Hence, we need more tests to investigate all rules and their terms in the LP. We extend T with the following additional test goals $mother(hariet, adrian)?$, $mother(babara, uwe)?$, $son(adrian, hariet)?$ and $son(uwe, babara)$. This leads to four new specializations:

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

The additional lggs are then:

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

The test now covers P, i.e. coverage = 100%.

The coverage measure determines how much of the general information expressed by the rules in the program is already covered by the actual tests and highlights those rules which may not be adequately tested and which require additional testing. The actual lggs give feedback how to extend the set of test goals in order to increase the coverage level. Moreover, repeatedly measuring the test coverage each time when the rule base becomes updated (e.g. when new rules are added) keeps the test suites (set of test cases) up to acceptable testing standards and one can be confident that there will be only minimal problems during runtime of the LP, because the rules do not only pass their tests but they are also

well tested. In contrast to other computations of the least general generalizations such as implication (i.e. a stronger ordering relationship), which becomes undecidable if functions are used, θ-subsumption has nice computational properties and it works for simple terms as well as for complex terms, e.g. $p() :- q(f(a))$ is a specialization of $p :- q(X)$. Although, it must be noted that the resulting clause under generalization with θ-subsumption ordering may turn out to be redundant, i.e. it is possible to find an equivalent one which is described more shortly, this redundancy can be reduced and since we are only generalizing the specializations on the top level this reduction is computationally adequate. So θ-subsumption and least general generalization qualify to be the right framework of generality in the application of our test coverage notion.

4 Integrating Test Cases into Rule Markup Languages

In this section we propose an abstract syntax for rule tests as a MOF metamodel (See figure 1) and a concrete XML syntax, which is validated with help of a W3C XML Schema. We argue that our metamodel can be used for testing rule bases, expressed in different markup languages. We give examples of rule base tests in RuleML and SWRL.

In order to test a rule base we define a concept of a test suite. We assume that one rule base may have several test suites and test suites may be distributed and refer to a rule base via URI. A test suite has an optional URI reference to a testing rule base.

A test suite consists of several tests. A test has a testing purpose, which is expressed informally by means of an attribute *purpose*. A purpose is, for instance, a testing of one predicate or one rule.

We define an abstract concept of an AbstractTest to allow test suites to be composed of mixed sets of other test suites and tests. This follows the design by Gamma and Beck used in several XUnit's. A test suite may contain *ignored* tests, which are excluded from the test run. A test consists of test assertions, which are ground formulas without Negation As Failure in a hosting rule language, and a semantics of an inference engine. A class InferenceEngineSemantics can be instantiated by different concrete semantics, for instance, stable model semantics, well-founded semantics, etc. This is to take into account that the outcome of a test run depends not only on the rules but also on the inference algorithm used.

A test refers to at least one test item, which consists of a formula in the hosting rule language as a test query. A test item refers to a list of results as expected results of a test query, using test assertions of a corresponding test.

A result consists of variable-value pairs, defining a substitution of each variable by the corresponding term as a value.

A test item has an optional purpose and expected answer attribute with possible values *yes/no/unknown*. If result set is non-empty, then the answer must be "yes".

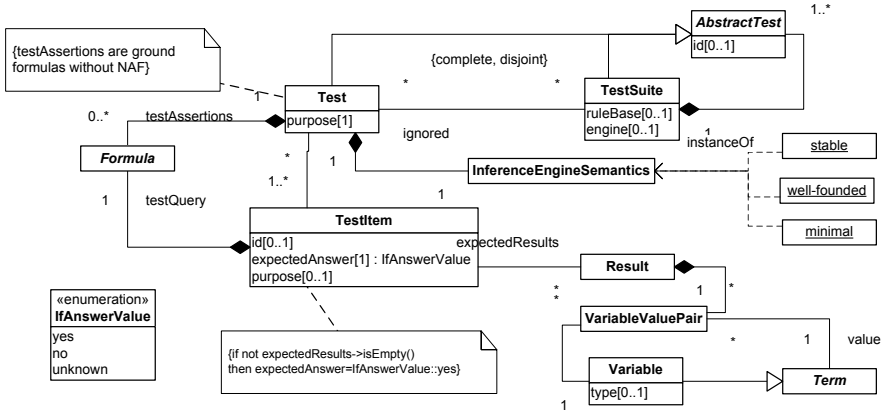

Fig. 1. Rule Test Metamodel

The concrete XML syntax for RuleML can be obtained from the metamodel, depicted on Figure 1, by applying the following principles of the XML Schema development:

Every model class is represented in the schema by an XML element, whose name is the class name, as well as a complex type, the name of which is the class name. If the class is abstract, the corresponding element is also abstract. The corresponding XML element, contains XML attributes for each data-valued property (attribute) from the model class. Our metamodel contains only *optional* or *required* attributes which are mapped to `optional`, respectively `required` attribute in the XML Schema. The metamodel does not contain any object valued attribute.

A functional association is mapped as a part of the content model of the class referencing this association in a form of XML attribute. If a role name is presented, then this name is used as an XML attribute name. If the role name is not presented then referenced class name is used as an XML attribute name. For instance, the functional association between the class *VariableValuePair* and the class *Variable* does not provide a role and therefore the referenced class name in the schema definition is used.

Composite and multivalued properties are always serialized using XML elements. A non-functional association is mapped as a part of the content model of the class referencing this association in a form of XML element. If a role name is provided, then this name is used for the corresponding element name. If a role is undefined, then the referenced class name is used. Let's consider, for example, the association between *Test* class and *Formula* class. Since a role name is provided (i.e. *testAssertions*) an element with this name is defined as a content part of the element `Test`.

Let's consider a sample RuleML rule base, which consists of two rules, defining well-known concepts of a *brother* and an *uncle*. We use Prolog-like syntax to express these rules.

```
brother(X, Y):-parent(Z,X), parent(Z,Y).
uncle(X,Y):-parent(Z,Y),parent(X,Z).
```

The RuleML test suite for testing this rule base is:

```
1 <TestSuite ruleBase="SampleBase.xml">
2  <Test id="ID001" purpose="...">
3    <testAssertions>
4     <RuleMLFormula>
5      <ruleml:And>
6       <ruleml:Atom>
8        <ruleml:Rel>parent</ruleml:Rel>
9        <ruleml:Ind>John</ruleml:Ind>
10       <ruleml:Ind>Mary</ruleml:Ind>
11      </ruleml:Atom>
12      ...
       </ruleml:And>
13    </RuleMLFormula>
14   </testAssertions>
15   <TestItem expectedAnswer="yes">
16    <testQuery>
17     <RuleMLFormula>
18      <ruleml:Atom closure="universal">
21        <ruleml:Rel>uncle</ruleml:Rel>
22        <ruleml:Ind>Mary</ruleml:Ind>
23        <ruleml:Var>Nephew</ruleml:Var>
24      </ruleml:Atom>
25     </RuleMLFormula>
26    </testQuery>
27    <expectedResults>
28     <VariableValuePair>
29      <ruleml:Var>Nephew</ruleml:Var>
30      <ruleml:Ind>Tom</ruleml:Ind>
31     </VariableValuePair>
32     <VariableValuePair>
33      <ruleml:Var>Nephew</ruleml:Var>
34      <ruleml:Ind>Irene</ruleml:Ind>
35     </VariableValuePair>
36    </expectedResults>
37   </TestItem>
38   <InferenceEngineSemantics>minimal
39   </InferenceEngineSemantics>
40  </Test>
41 </TestSuite>
```

This test suite has one test (line 2). The test assertions of the test (lines 3-14) are ground facts for rules in the RuleML rule base in RuleML language. Lines 8-10 describe a ground fact: *John is a parent of Mary*. We assume that the test assertions of the test also contain the following ground facts: *John is a parent of Paul*, *Paul is a parent of Tom*, *Paul is a parent of Irene*. We have omitted representation of these facts in the XML above due to space limitations. The test has one test item (line 15-37) with one test query as a RuleML formula (lines 17-25). The query is *Who are the nephews of Mary?*. Expected results (lines 27-36) of the query consist of two variable value pairs: *Nephew is Tom* (lines 28-31) and *Nephew is Irene* (lines 32-35). The test query from the above XML (lines 16-26) can be expressed using also SWRL:

```
<swrlx:individualPropertyAtom swrlx:property="uncle">
 <owlx:Individual owlx:name="Mary">
 <ruleml:var>Newphew</ruleml:var>
</swrlx:individualPropertyAtom>
```

It is possible to define a test item in the test with a query without expected results. For instance, the query *Who are the nephews of John?* must produce no results when applying rules from the rule base.

```
<TestItem expectedAnswer="no">
 <testQuery>
  <RuleMLFormula>
   <ruleml:Atom closure="universal">
    <ruleml:Rel>uncle</ruleml:Rel>
    <ruleml:Ind>John</ruleml:Ind>
    <ruleml:Var>Nephew</ruleml:Var>
   </ruleml:Atom>
  </RuleMLFormula>
 </testQuery>
</TestItem>
```

As can be seen from these examples, the only rule language specific part is the abstract class *Formula* (Figure 1). This class corresponds to the following definition in the XML Schema:

```
<xs:complexType name="Formula.Type">
 <xs:choice>
  <xs:element ref="RuleMLFormula"/>
   <xs:element ref="SWRLFormula"/>
  </xs:choice>
</xs:complexType>
```

Elements RuleMLFormula and SWRLFormula are defined in the schema as well.

In order to support some other rule language, XML Schema `redefine` instruction should be used to redefine complex type `Formula.Type`. Thus, the presented metamodel can be used for writing test in different rule languages.

5 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. [17]. Criteria for verification and validation range from e.g. structural checks for relevance, redundancy and reachability to semantics tests for completeness and consistency. For a survey see [16]. Several verification and validation methods have been proposed, such as:

Methods based on *operational debugging* [31] 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, these methods presuppose a deep understanding of the inference processes by the user to detect the inconsistencies.

Tabular methods, e.g. [19], 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. [20, 21], 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. [22] which model the rule base as a Petri net and test the complete models starting with all possible initial states, which is very costly.

Methods based on *declarative debugging* [32] which build an abstract model representing the execution trace and elicit feedback from an oracle (e.g. the user) to navigate through the model till the inconsistency/error is reached.
Methods based on *algebraic interpretation*, e.g. [23] 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.
While this approaches mainly focus on monotonic reasoning, there are also some approaches on verifying non-monotonic rule bases such as [24] which analyzes rule bases expressed in default logic or [25] which tests rule bases with production rules. For further details concerning inconsistency checking techniques see e.g. [26]. Much research has been directed at the automated refinement of rule bases, e.g. [28, 27], and on the automatic generation of test cases, e.g. [29]. For an overview on rule base debugging tools see e.g. [30]. Test coverage of imperative programs has been intensively investigated in the past decades [7], but there are only a few attempts addressing test coverage measurement for test cases of backward-reasoning rule based programs [33, 36, 35] or forward-reasoning production rule systems [34].

6 Conclusion

Rules are often being used as a declarative programming language to describe real-world decision logic and create production systems upon. 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. Test cases for V&V of rule bases are particular well-suited when rule applications grow larger and more complex and are maintained (possibly distributed) by different people. They help to capture rule engineer's intended meaning of a rule-based program 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 high level decision logic, on the other hand. Although various V&V approaches of rule bases have been introduced in literature, most of these approaches have a rather practical nature and are depending on the syntactical structure of a particular rule representation language which constrains them to carry out only partial tests of the properties of the rule base. In this paper we have introduced an abstract conceptual framework, in the style of Tarski, which allows treating a wide range of model theoretic semantics and developed a test-driven validation methodology upon, which treats test cases as constraints on the set of possible models, i.e. they describe an approximation of the intended model(s) which can be automatically validated against the derived models (the rule programs' output) via a set of inference operations. This leads to a well-defined and more general semantical basis, independent of the particular syntactical representation of the various rule base properties and their logical relationships that can be tested. We have given a concrete instantiation of these unifying abstract framework in logic programming using LP-based test cases based on a set of test queries and have elaborated on a LP-based test coverage measure for determining the quality of a test case. Testing methodologies delivering quality data for rule based reliability models have not been investigated very often. The coverage notion defined 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. Finally, we have present an abstract syntax for rule tests as a MOF metamodel and a concrete RuleML-based syntax, semantics and implementation of the metamodel. Although a lot of work still needs to be done, the findings about the application of test-driven development for self-validating rule bases and the proposed testing methodologies and techniques give enough reason to make a relevant contribution and simultaneously motivate further research in this field.

References

1. 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.
2. K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
3. Virginia E. Barker, Dennis E. O'Connor, J. Bachant, and E. Soloway. Expert systems for configuration at digital: Xcon and beyond. *Commun. ACM*, 32(3):298–318, 1989.
4. 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.
5. M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. 1995.
7. Zhu, H., Hall, P. A., and May, J. H. 1997. *Software unit test coverage and adequacy*. ACM Comput. Surv. 29, 4 (Dec. 1997), 366-427. DOI= http://doi.acm.org/10.1145/267580.267590
8. E. Harold. An early look at JUnit 4. `http://www-128.ibm.com/developerworks/java/library/j-junit4.html`.
9. S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2):167–207, 1990.
10. L. 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.
11. D. 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.
12. 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.
13. J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
14. G.D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5, 1970.

15. A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
16. A. Preece. Evaluating verification and validation methods in knowledge engineering. University of Aberdeen, 2001.
17. G. Antoniou, F.v. Harmelen, R. Plant, J. Vanthienen. Verification and validation of knowledge-based systems - report on two 1997 events. AI Magazine, 19(3):123126.
18. 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.
19. 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.
20. 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.
21. M. Ramaswamy, S. Sarjar, C. Ye Sho. Using directed hypergraphs to verify rule-based expert systems. *IEEE TKDE*, 9(2), 1997, p. 221-237.
22. 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.
23. 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.
24. G. Antoniou. Verification and Correctness Issues for Nonmonotonic Knowledge Bases. *Int. J. of Intelligent Systems*, 12(10), 1997, p. 725-739.
25. C.H. Wu, S.J. Le. Knowledge Verification with an Enhanced High-Level Petri-Net Model. *IEEE Expert*, 1997, p. 73-80.
26. F.P. Coenen, T. Bench-Capon. Maintenance of Knowledge-Based Systems: Theory, Techniques and Tools. Academic Press, London, 1993.
27. S. Craw, D. Sleeman. Automating the Refinement of KBS. *Proceedings ECAI'90*, 1990.
28. 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.
29. 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.
30. R.T. Plant. Tools for Validation&Verification of Knowledge-Based Systems 1985-1995. Internet Source.
31. L. Byrd. Understanding the control flow of prolog programs. Proceedings of the Workshop on Logic Programming, 1980.
32. E.Y. Shapiro. Algorithmic program debugging. MIT Press, May 1982.
33. R. Denney. Test-Case Generation from Prolog-Based Specifications, IEEE Software, vol. 8, no. 2, pp. 49-57, Mar/Apr, 1991.
34. G. Antoniou, O. Jack. Testing Production System Programs, p. 214, The Ninth International Symposium on Software Reliability Engineering, 1998.
35. O. Jack. Software Testing for Conventional and Logic Programming, vol. 10 of Programming Complex Systems, F. Belli (ed.), Walter de Gruyter & Co., Berlin, New York, 1996.
36. G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In Int. Symp. on Software Reliability Enginnering, pages 104113, 1992.