I1-D7

Verification and Validation of R2ML Rule Bases

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Cottbus/I1-D7/D/PU/b1
Responsible editors:	Sergey Lukichev
Reviewers:	Dragan Gasevic
Contributing participants:	Cottbus
Contributing workpackages:	I1
Contractual date of deliverable:	September 29, 2006
Actual submission date:	September 29, 2006

Abstract
In this report we give preliminary definitions of anomalies in rule bases for different types of rules. In addition, we describe a test-driven approach to rule-base validation.

Keyword List
Rules, Validation, Verification, R2ML, Rule Markup Language, Semantic Web

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Verification and Validation of R2ML Rule Bases

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

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

September 29, 2006

Abstract
In this report we give preliminary definitions of anomalies in rule bases for different types of rules. In addition, we describe a test-driven approach to rule-base validation.

Keyword List
Rules, Validation, Verification, R2ML, Rule Markup Language, Semantic Web

Contents

Chapter 1

Introduction

In this report we give preliminary definitions of anomalies in rule bases for different types of rules. Definitions are given in the Chapter 2.

Chapter 3 describes a test-driven approach to rule bases validation. We describe general principles of test-based validation, define a metamodel for tests and describe how XML Schema can be derived from in in order to encode rule tests. We consider four rule types: derivation rules, production rules, reaction rules and integrity rules, explain how tests can be created for each rule type and give test examples.

Chapter 2

Verification of R2ML Rulebases

Rule base verification commonly means to analyse a rule base for checking if it contains any *anomalies* that would indicate probable errors. Verification is an important quality assurance measure. The following definitions of anomalies in R2ML rule bases are based on [PS94] and [Ant97].

2.1 R2ML Rule Bases

We consider three cases of R2ML rule bases:

1. A *derivation rule base* consists of a vocabulary, one or more derivation rule sets, and zero or more integrity rule sets.
2. A *production rule base* consists of a vocabulary, one or more production rule sets, and zero or more integrity rule sets.
3. A *reaction rule base* consists of a vocabulary, one or more reaction rule sets, zero or more derivation rule sets, and zero or more integrity rule sets.

A rule base R together with a fact base X over the vocabulary of R forms a knowledge base $\langle X, R\langle$.

An R2ML sentence literal is either an atom (a pos-literal), a negated atom (a not-literal), a strongly negated atom (a neg-literal), a negation-as-failure-negated atom (a naf-literal), or a negation-as-failure-negated strongly negated atom (a nafneg-literal). Rule conditions can be transformed into a disjunctive normal form that is a disjunction of conjunctions of R2ML sentence literals.

An *incompatibility constraint* is an integrity rule of the form $\forall x, y, \ldots \not(A_1 \wedge A_2)$, which requires for two atoms A_1 and A_2 that they never hold at the same time.

2.2 Anomalies Applying to All Three Kinds of Rule

2.2.1 Redundancy: Unsatisfiable Condition

A rule in an R2ML rule base has an unsatisfiable condition if each conjunction of the disjunctive normal form of the condition contains a literal whose predicate is neither included in the rule

base vocabulary nor does it occur in the conclusion of any derivation rule.

2.3 Anomalies in Derivation Rules

2.3.1 Redundancy

2.3.1.1 Redundancy: Unsatisfiable Condition

See 2.2.1.

2.3.1.2 Redundancy: Subsumed Rule

A derivation rule r' is subsumed by another derivation rule r if the conditions of r' are subsumed by the conditions of r, and the conclusion of r' subsumes the conclusion of r. A formula F subsumes another formula F' if for each interpretation $\mathcal{I}$ and substitution σ such that $\mathcal{I} \models \mathcal{F}\sigma$ there is a more specific substitution σ' (extending σ) such that $\mathcal{I} \models \mathcal{F}'\sigma'$.

2.3.1.3 Redundancy: Redundant Rule

A derivation rule r is redundant in a rule base R if the consequence set of the knowledge base $\langle X, R\langle$ is the same as that of $\langle X, R - \{r\}\langle$ for any admissible fact base X.

2.3.2 Circularity

A derivation rule r is circular in a rule base, if for some ground instance $r\sigma = F\sigma \leftarrow G\sigma$, the conclusion $F\sigma$ depends on itself, that is, it contains a literal which is contained in all 'prerequisite expansion sets' obtained by following the arcs in the dependency graph.

2.3.3 Inconsistency

2.3.3.1 Inconsistency: Incompatible Rule Pair

Two derivation rules are incompatible in a rule base, if the conditions of one of them subsume the conditions of the other, and their conclusions violate an incompatibility constraint.

2.3.3.2 Other Patterns of Inconsistency

It is an open issue, if there are other patterns of inconsistent rule bases that would be relevant for anomaly detection.

2.4 Anomalies in Production Rules

2.4.1 Redundancy

2.4.1.1 Redundancy: Unsatisfiable Condition

See 2.2.1.

2.4.1.2 Redundancy: Subsumed Rule

A production rule r' is subsumed by another production rule r if the conditions of r' are subsumed by the conditions of r, and the action expression of r includes the action expression of r' as a subaction. An action expression A includes another action expression A' as a subaction if A can be rewritten as a sequence of actions such that one of them is A'.

2.4.1.3 Redundancy: Redundant Rule

A production rule r is redundant in a rule base R if the knowledge base $\langle X, R\rangle$ has the same semantics as $\langle X, R - \{r\}\rangle$ for any admissible fact base X.

2.4.2 Circularity

A notion of circularity for reaction rules still has to be elaborated. We just describe here the basic idea.

A production rule r is circular in a rule base, if for some ground instance $r\sigma = C\sigma \rightarrow A\sigma$, the produced action $C\sigma$ affects the validity of the condition $C\sigma$.

2.4.3 Inconsistency

2.4.3.1 Inconsistency: Incompatible Rule Pair

Two production rules are incompatible in a rule base, if the condition of one of them subsumes the condition of the other, and their action expressions and/or post-conditions are incompatible with each other.

2.4.3.2 Other Patterns of Inconsistency

It is an open issue, if there are other patterns of inconsistent production rule bases that would be relevant for anomaly detection.

2.5 Anomalies in Reaction Rules

2.5.1 Redundancy

2.5.1.1 Redundancy: Unsatisfiable Condition

See 2.2.1.

2.5.1.2 Redundancy: Subsumed Rule

A reaction rule r' is subsumed by another reaction rule r if

1. the triggering event expression of r' is subsumed by the triggering event expression of r,
2. the conditions of r' are subsumed by the conditions of r,
3. the postcondition of r is subsumed by the postcondition of r', and
4. the action expression of r includes the action expression of r' as a subaction.

2.5.1.3 Redundancy: Redundant Rule

A reaction rule r is redundant in a rule base R if the knowledge base $\langle X, R\rangle$ has the same semantics as $\langle X, R - \{r\}\rangle$ for any admissible fact base X.

2.5.2 Circularity

A notion of circularity for reaction rules still has to be elaborated. We just describe here the basic idea.

A reaction rule r is circular in a rule base, if for some ground instance $r\sigma = E\sigma, C\sigma \rightarrow P\sigma, A\sigma$, the postcondition $P\sigma$ or the triggered action $C\sigma$ affects the validity of the condition $C\sigma$ or of the triggering event $E\sigma$.

2.5.3 Inconsistency

2.5.3.1 Inconsistency: Incompatible Rule Pair

Two reaction rules r_1 and r_2 are incompatible in a rule base, if the condition of r_1 subsumes the condition of r_2, the event expression of r_1 subsumes the event expression of r_2, and their action expressions and/or post-conditions are incompatible with each other.

2.5.3.2 Other Patterns of Inconsistency

It is an open issue, if there are other patterns of inconsistent reaction rule bases that would be relevant for anomaly detection.

Chapter 3

Test-Driven Validation of Rule Bases

A rule base validation commonly means a process that aims for the detection of incorrect results or undesired behavior. In the context of rule bases validation, validation is often done by testing the application and assessing the results, or comparing the results with previous results that were believed correct. In this chapter we describe a test-driven validation of rule bases in general and explain how R2ML rule bases can be validated by means of tests.

3.1 Introduction

Extreme programming [Bec00, Bec99] 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 [Lay04]. 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 [GHJV95]. 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 specification-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 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.

3.2 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 by users (both end users or 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 [VEBS89]. 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 [McC76], 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[McC80, Mak89, SK90]. 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:

$M_0 \models_{TC} (X, A, +)$ iff $\forall m \in M_0 : m \in \Sigma(Mod(X), R) \Rightarrow m \in Mod(A)$
$M_0 \models_{TC} (X, A, -)$ 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 [DP05]. In analogy to refactoring in software engineering [Fow99], 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 [PP96].

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).

3.3 Rule-Base Validator

The rule-base validator, based on principles, described in this section, works as follows:

- The validator loads a specified test suite, including tests, test assertions and expected results;
- The validator resolves a rule base for testing;
- The validator passes the rule base, test assertions and test queries into the rule engine for execution;
- The rule engine performs test queries on test assertions, executing rules from the rule base;
- The validator receives execution results from the rule engine and compares them against expected results, specified in the test suite.

The structure of a test suite is described in the subsequent sections for each type of rules: derivation rules, production rules, integrity rules, reaction rules.

3.4 General Test Metamodel for Rules

In this section we propose an abstract syntax for a test suite as a MOF metamodel 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 derivation rule base tests in R2ML and RuleML in Section 3.5.3. Production

rules are discussed in the Section 3.7. Integrity rules are discussed in the Section 3.9. Reaction rules are discussed in the Section 3.8.

An abstract syntax for rule test is depicted on Figure 3.1. 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. This URI can be used by the validator in order to discover a remote rule base, which is going to be tested.

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 (formulas without variables) without Negation As Failure in a hosting rule language, message event expression as test events (used for testing of reaction rules) 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".

The concrete XML syntax for a specific rule language can be obtained from the metamodel, depicted on Figure 3.1, by applying the following principles of the XML Schema development:

1. Every model class is represented in the schema by an XML element, whose name is the class name, as well as a complex type, which name 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.

2. 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.

3. 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

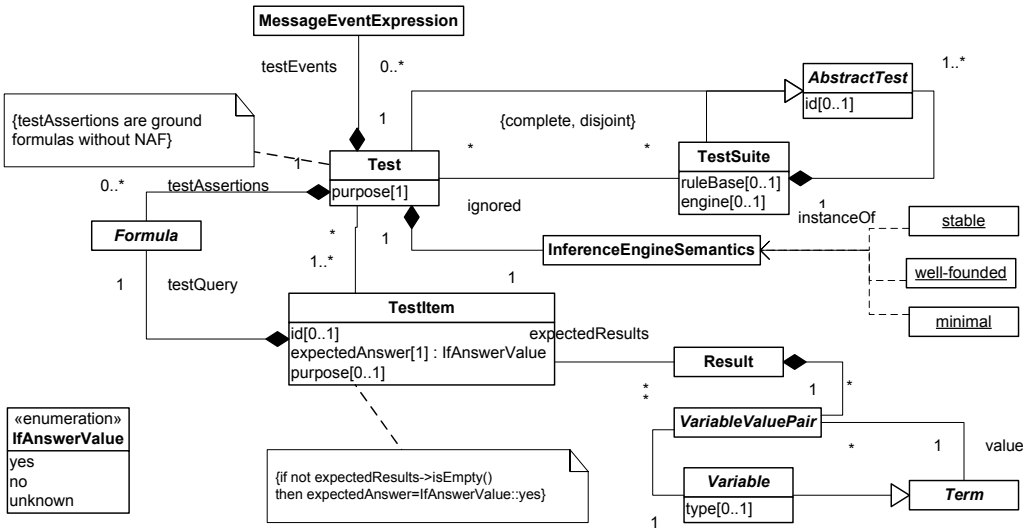

Figure 3.1: Rule Test Metamodel

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`.

The rule language specific parts of the metamodel are classes *Formula* and *VariableValuePair* (Figure 3.1). Class *Formula* corresponds to the following type definition in the XML Schema:

```
<xs:complexType name="Formula.Type">
 <xs:choice>
  <xs:element ref="R2MLFormula"/>
  <xs:element ref="RuleMLFormula"/>
  <xs:element ref="SWRLFormula"/>
  </xs:choice>
</xs:complexType>
```

Elements R2MLFormula, RuleMLFormula, and SWRLFormula are defined in the schema as well. For instance, the type of R2MLFormula element is defined as following:

```
<xs:complexType name="R2MLFormula.Type" abstract="false">
  <xs:choice>
    <xs:element ref="r2ml:LogicalFormula"/>
  </xs:choice>
</xs:complexType>
```

Class *VariableValuePair* corresponds to the following type definition in the XML Schema:

```
<xs:complexType name="VariableValuePair.Type" abstract="false">
 <xs:sequence>
```

```
  <xs:choice>
   <xs:element ref="R2MLVariableValuePair" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="RuleMLVariableValuePair" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="SWRLVariableValuePair" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
 </xs:sequence>
</xs:complexType>
```

It means that the expected result can be expressed in R2ML, RuleML or SWRL language. Element *R2MLVariableValuePair* is of the following XML complex type:

```
<xs:complexType name="R2MLVariableValuePair.Type" abstract="false">
 <xs:sequence>
  <xs:element ref="r2ml:GenericVariable"/>
  <xs:choice>
   <xs:element ref="r2ml:GenericVariable"/>
   <xs:element ref="r2ml:PlainLiteral"/>
   <xs:element ref="r2ml:TypedLiteral"/>
   <xs:element ref="r2ml:ObjectVariable"/>
  </xs:choice>
 </xs:sequence>
</xs:complexType>
```

3.5 Testing Derivation Rules

Testing of derivation rules is based on some assertions usually obtained with respect of the vocabulary elements used in the conditions part of a rule and using test queries obtained on the base of the vocabulary from the conclusion part of the rule. The test suite refers to a specific rule base.

3.5.1 Creating test assertions

Test assertions can be obtained from the conditions part of a rule. As it is specified by the metamodel (Figure 3.4), a test assertion is a ground formula (a formula without variables). Therefore a test assertion can be obtained from a rule condition by substituting condition variables by constants. For example, the following R2ML formula in the condition part of a rule:

```
<r2ml:GenericAtom r2ml:predicateID="parent">
 <r2ml:arguments>
  <r2ml:GenericVariable r2ml:name="X"/>
  <r2ml:GenericVariable r2ml:name="Y"/>
 </r2ml:arguments>
</r2ml:GenericAtom>
```

corresponds to the following test assertion:

```
<r2ml:GenericAtom r2ml:predicateID="parent">
 <r2ml:arguments>
```

```
  <r2ml:PlainLiteral r2ml:lexicalValue="John" r2ml:languageTag="en"/>
  <r2ml:PlainLiteral r2ml:lexicalValue="Mary" r2ml:languageTag="en"/>
 </r2ml:arguments>
</r2ml:GenericAtom>
```

3.5.2 Creating test queries

A test query can be obtained from the rule conclusion by substituting variables with constants. If a test query contains only ground facts, then the expected answer is "yes". It means that the formula is true on its ground facts. If the test query contains variables, a list of results is returned and compared against expected results, specified in the result set. For example, the following atom from the rule conclusion:

```
<r2ml:GenericAtom r2ml:predicateID="uncle">
 <r2ml:arguments>
  <r2ml:GenericVariable r2ml:name="X"/>
  <r2ml:GenericVariable r2ml:name="Neph1"/>
 </r2ml:arguments>
</r2ml:GenericAtom>
```

may correspond to the following R2ML query:

```
<r2ml:GenericAtom r2ml:predicateID="uncle">
 <r2ml:arguments>
  <r2ml:PlainLiteral r2ml:lexicalValue="Mary" r2ml:languageTag="en"/>
  <r2ml:GenericVariable r2ml:name="Neph1"/>
 </r2ml:arguments>
</r2ml:GenericAtom>
```

After the query execution, the variable `Neph1` is initialized with a list of values, which are then compared against expected results.

3.5.3 A Sample R2ML Test Suite for Derivation Rules

The XSML Schema, derived from the metamodel in Section 3.4, can be used to create test suites for rule bases, expressed in different rule markup languages. In this section we give an example of an R2ML test suite for testing an R2ML rule base. Let's consider a sample R2ML rule base, which consists of two rules, defining well-known concepts of a *brother* and an *uncle*. We use Prolog syntax to express these rules.

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

The R2ML test suite for this rule base is:

```
1  <TestSuite ruleBase="ExampleRuleBase.xml">
2   <Test id="ID001" purpose="">
3    <testAssertions>
4     <R2MLFormula>
5      <r2ml:qf.Conjunction>
```

```
6       <r2ml:GenericAtom r2ml:predicateID="parent">
7        <r2ml:arguments>
8         <r2ml:PlainLiteral r2ml:lexicalValue="John" r2ml:languageTag="en"/>
9         <r2ml:PlainLiteral r2ml:lexicalValue="Mary" r2ml:languageTag="en"/>
10       </r2ml:arguments>
11      </r2ml:GenericAtom>
12      ...
13     </r2ml:qf.Conjunction>
14    </R2MLFormula>
15   </testAssertions>
16   <TestItem expectedAnswer="yes">
17    <testQuery>
18     <R2MLFormula>
21       <r2ml:GenericAtom r2ml:predicateID="uncle">
22        <r2ml:arguments>
23         <r2ml:PlainLiteral r2ml:lexicalValue="Mary" r2ml:languageTag="en"/>
24         <r2ml:GenericVariable r2ml:name="Nephew"/>
25        </r2ml:arguments>
26       </r2ml:GenericAtom>
28     </R2MLFormula>
29    </testQuery>
30    <expectedResults>
31     <R2MLVariableValuePair>
32      <r2ml:GenericVariable r2ml:name="Nephew"/>
33      <r2ml:PlainLiteral r2ml:lexicalValue="Tom" r2ml:languageTag="en"/>
34     </R2MLVariableValuePair>
35     <R2MLVariableValuePair>
36      <r2ml:GenericVariable r2ml:name="Nephew"/>
37      <r2ml:PlainLiteral r2ml:lexicalValue="Irene" r2ml:languageTag="en"/>
38     </R2MLVariableValuePair>
39    </expectedResults>
40   </TestItem>
41   <InferenceEngineSemantics>minimal</InferenceEngineSemantics>
42  </Test>
43 </TestSuite>
```

This test suite has one test (line 2). The test assertions of the test (lines 3-15) are ground facts for rules in the R2ML rule base in R2ML language. Lines 6-11 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 16-40) with one test query as an R2ML logical formula (lines 18-28). The query is *Who are the nephews of Mary?*. Expected results (lines 30-39) of the query consist of two variable value pairs: *Nephew is Tom* (lines 31-34) and *Nephew is Irene* (lines 35-38).

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>
  <R2MLFormula>
   <r2ml:GenericAtom r2ml:predicateID="uncle">
    <r2ml:arguments>
     <r2ml:PlainLiteral r2ml:lexicalValue="John" r2ml:languageTag="en"/>
     <r2ml:GenericVariable r2ml:name="Nephew"/>
    </r2ml:arguments>
   </r2ml:GenericAtom>
  </R2MLFormula>
 </testQuery>
</TestItem>
```

3.5.4 A Sample RuleML Test Suite

In this section we give an example of a RuleML test suite for testing a RuleML rule base. 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).

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>
```

3.6 Measuring the Quality of Tests for Rules

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 semantics of backward-reasoning (resp. forward-reasoning rule systems such as production rule system) and the common imperative (Object-oriented) 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. These methods are not directly applicable in backward-reasoning rule systems, such as logic programming inference engines (e.g. Prolog), which typically make a goal-driven refutation attempt applying a resolution algorithm with backtracking and unification, since here 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, a coverage measure for such backward-reasoning engines should take this unification based execution model into account where test goals (queries) are used to instantiate the rules, leading to specializations of the rules. Using the goal reductions in the body part of a derivation rules as sub-goals for further derivations leads to more specific rule specializations on the next level and by repetition of this process to a rule specialization order $(H \leftarrow B) \geq (H \leftarrow B)' \geq ...$, whereas $\geq$ denotes the level relation "more general as". Accordingly, to adequately test the rules of a derivation rule base which follows the declarative programming paradigm the test queries should investigate the full scope of the terms of all rules in the rule base by specializing these rules via unification with the test queries. In other words, the coverage level of a test case is determined by the level of how much of the general information represented by the rules' terms is tested by the test goals. Obviously, to fully investigate a rule set, i.e. to cover it, only a least general set of test queries is needed, which has to be determined.

Inductively deriving general knowledge from specific knowledge is a task which is approached by inductive logic programming (ILP) techniques which allow computing the least general generalization (lgg) (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. 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 $r \subseteq r'$ (see e.g. [Plo70]). Based on the concepts of θ-subsumption and least general generalization it is possible to determine the level of coverage by generalizing the specializations of the rules, which are instantiated by the test goals, under θ-subsumption ordering, i.e. computing the lggs of all successful specializations, and attempt a reconstruction of the original rule base. The number of successful "recoverings" then gives the level of test coverage, i.e. the level determines those statements (rules) in a rule base that have been executed/investigated through a test run and those which have not. In particular, if the complete rule base can be reconstructed via generalization of the specialization then the test fully covers the rule base. Formally we express this as follows:
Let T be a test with a set of test queries $T := \{Q_1?, .., Q_n?\}$ for a rule base 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 rule base 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 rule base or not. The coverage measure for a rule base 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 rule base P with the 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. 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 rule base 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 rule base, 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.

Although, the defined coverage measure is based on the central concept of unification and uses ILP techniques for generalization of the derived specializations of the rule base, it is worth noting, that the measure might be applied also in the context of forward-directed reactive rules such as ECA rules or production rules. There are several approaches in the active database domain which transform active rules into LP derivation rules, in order to exploit the formal declarative semantics of logic programs to overcome confluence and termination problems of active rule execution sequences, where the actions are input events of further active rules ([Zan95], [SF98], [BL96]). For such transformed declarative rule bases consisting of LP derivation rules test cases can be written and the coverage can be computed as described above. The combination of deductive and active rules has been also investigated in different approaches mainly based on the simulation of active rules by means of deductive rules ([Lud98], [Lau98], [Zan94]). Moreover, there are approaches which directly build reactive rules on top of LP derivation rules such as the Event Condition Action Logic Programming language (ECA-LP) which enables a homogeneous representation of ECA rules and derivation rules ([Pas05b], [Pas05a]). Closely related are also logical update languages such as transaction logics and in particular serial Horn programs, where the serial Horn rule body is a sequential execution of actions in combination with standard Horn pre-/post conditions. [BK93] These serial rules can be processed top-down or bottom-up and hence are closely related to the production rules style of *condition* $\rightarrow$ *update action*. This partial relation between backward reasoning LP derivation rules and forward reasoning production rules which enables transformations of production rule bases into logic programs has been also shown for a subclass of production rules, the stratified production rules. Hence, these class of production rules also qualifies for our goal-driven testing approach and unification based test coverage measure ([Ras92], [DM02]).

3.7 Testing Production Rules

A production rule consists of conditions, a produced action and a postcondition. R2ML supports four action types: assign action, create action, delete action and invoke action. Each of these rule actions changes the state of the knowledge base. A test query of a test item (see Figure 3.4) returns values, which are compared against expected results. For each action type we describe a simple mapping from R2ML ActionExpression into test query in R2ML.

3.7.1 Creating test assertions

Test assertions are created similar to that of derivation rules. See Section 3.5.1 for details.

3.7.2 Creating test queries from actions

Each of the possible R2ML action expression is a source of test action effects. The engineer creates test action effects by using the vocabulary elements of these actions.

3.7.2.1 Assign Action

An R2ML AssignActionExpression assigns a value to the attribute of a context object. An AssignActionExpression has the following template:

```
<r2ml:AssignActionExpression r2ml:propertyID=$p>
 <r2ml:contextArgument>
  $ObjectTerm
 </r2ml:contextArgument>
  $Term
</r2ml:AssignActionExpression>
```

If the property `$p` denotes an object, then this assign action expression corresponds to the following template of a test query:

```
<r2ml:ReferencePropertyAtom r2ml:referencePropertyId=$p>
 <r2ml:subject>
  $ObjectTerm
 </r2ml:subject>
 <r2ml:objectValue>
  $Term
 </r2ml:objectValue>
</r2ml:ReferencePropertyAtom>
```

In this template and in the followings for other action types, `$ObjectTerm` and `$Term` should be substituted by variables or values, depending what kind of testing is required. If terms are substituted with values, it is a ground query, which returns a *yes/no/unknown* answer. If some term is a variable, and other is a value, the query returns a list of results, which are compared against expected results, specified in the test suite.

If the property p is an object attribute, then the assign action expression corresponds to the following template of a test query:

```
<r2ml:AttributionAtom r2ml:attributeID=p>
 <r2ml:subject>
  $ObjectTerm
 </r2ml:subject>
 <r2ml:dataValue>
  $Term
 </r2ml:dataValue>
</r2ml:AttributionAtom>
```

For instance, the expected state change after the assign action *set the color of the rental car car101 to "red"* is queried by the following R2ML query:

```
<r2ml:Attributionatom r2ml:attributeID="cars:color">
 <r2ml:subject>
  <r2ml:ObjectName r2ml:objectID="cars:car101"/>
 </r2ml:subject>
 <r2ml:dataValue>
   <r2ml:TypedLiteral r2ml:datatypeID="xs:string" r2ml:lexicalValue="red"/>
```

```
 </r2ml:dataValue>
</r2ml:Attributionatom>
```

This corresponds to the following assignment of the metavariables: `$ObjectTerm` is the element `<r2ml:ObjectName r2ml:objectID="cars:car101"/>` and the `$Term` variable is bounded to the element `<r2ml:TypedLiteral r2ml:datatypeID="xs:string" r2ml:lexi`

3.7.2.2 Create Action

An R2ML CreateActionExpression creates an object from the list of arguments. A CreateActionExpression has the following template:

```
<r2ml:CreateActionExpression r2ml:classID=$c>
 <r2ml:ObjectName r2ml:objectID=$name/>
 $ObjectSlot1
 ...
 $DataSlot1
 ...
</r2ml:CreateActionExpression>
```

It corresponds to the following R2ML query template:

```
<r2ml:ObjectDescriptionAtom r2ml:classID=$c>
 <r2ml:subject>
  <r2ml:ObjectName r2ml:objectID=$name/>
 </r2ml:subject>
 $ObjectSlot1
 ...
 $DataSlot1
 ...
</r2ml:ObjectDescriptionAtom>
```

For instance, the expected state change after the create action *create Person X with name "John"* can be queried by the following R2ML formula:

```
<r2ml:ObjectDescriptionAtom r2ml:classID="Person">
 <r2ml:subject>
  <r2ml:ObjectName r2ml:objectID="person101"/>
 </r2ml:subject>
 <r2ml:DataSlot r2ml:attributeID="name">
  <r2ml:value>
   <r2ml:TypedLiteral r2ml:datatypeID="xs:string" r2ml:lexicalValue="John"/>
  </r2ml:value>
 </r2ml:DataSlot>
</r2ml:ObjectDescriptionAtom>
```

3.7.2.3 Delete Action

The R2ML DeleteActionExpression deletes an object. A DeleteActionExpression has the following template:

```
<r2ml:DeleteActionExpression r2ml:classID=$c>
 $Term
</r2ml:DeleteActionExpression>
```

It corresponds to the following R2ML query template:

```
<r2ml:qf.NegationAsFailure>
 <r2ml:ObjectClassificationAtom r2ml:classID=$c>
  $Term
 </r2ml:ObjectClassificationAtom>
</r2ml:qf.NegationAsFailure>
```

For instance, the expected state change after the delete action *delete Person o1* can be queried by the following R2ML formula:

```
<r2ml:qf.NegationAsFailure>
 <r2ml:ObjectClassificationAtom r2ml:classID="Person">
  <r2ml:ObjectName r2ml:objectID="o1"/>
 </r2ml:ObjectClassificationAtom>
</r2ml:qf.NegationAsFailure>
```

3.7.2.4 Invoke Action

The R2ML InvokeActionExpression invokes an operation with a list of parameter arguments and changes the state of the context object. We consider invoke actions, which change the state of the context object. For instance, the expected state change after the invoke action *change the name of Person o101 to "John"* can be queried by the following R2ML formula:

```
<r2ml:ObjectDescriptionAtom r2ml:classID="Person">
 <r2ml:subject>
  <r2ml:ObjectName r2ml:objectID="o101"/>
 </r2ml:subject>
 <r2ml:DataSlot r2ml:attributeID="name">
  <r2ml:value>
   <r2ml:TypedLiteral r2ml:datatypeID="xs:string" r2ml:lexicalValue="John"/>
  </r2ml:value>
 </r2ml:DataSlot>
</r2ml:ObjectDescriptionAtom>
```

Those production rules, for which the action effect is not known, can be validated only if there is a postcondition specified. If the action effect is not known, it is also not known which state change of the KB has occurred. In order to resolve the state change, a postcondition for the rule must be specified. A postcondition is a logical formula, which must hold after the execution of the action. The generation of test queries according with rules postconditions is similar with the generation of test queries for conclusions in the case of derivation rules.

3.8 Testing Reaction Rules

This section is devoted to the reaction rules validation. Since RuleML does not provide support for reaction rules, the validation concerns the reaction rule set of the R2ML language. The

validation of reaction rules is similar to the validation of production rules. The difference is that a test consists of test events in addition to test assertions. The validator passes test assertions and the rule base to the reaction rule engine and generates triggering events from the test events.

This report concerns just reaction rules triggered by message events which are atomic i.e. they have no duration. Such an event have the following structure:

- The *eventType* property specifying the event type;
- The attribute *startDateTime* which encodes the start date and time of the event;
- The *sender* property encoding the sender of the event;
- A set of *arguments*, elements from the vocabulary. These arguments are passed to the reaction rule conditions part by the rule engine.

More complex events like AndNotEventExpression or SequenceEventExpression (see [WGL06] for all supported events) will require the same model of validation but more capabilities of the validator to generate these complex events.

As in the case of production rules the validator generates a set of test queries corresponding to the action patterns from the head of the reaction rules we have in the rule base. Therefore a reaction rules test suite is a tuple (TE, CA, QA, QP, R) where:

- TE denotes a set of *test events* according to the vocabulary of events used in the reaction rule base;
- CA denotes a set of *test assertions* according to the vocabulary of conditions used in the reaction rules rulebase;
- QA denotes a set of *test actions* according to the vocabulary of actions used in the rule base;
- QP denotes a set of *test queries* according to the vocabulary of postconditions used in the reaction rules rulebase.
- R denotes a set of *expected results* in the form of a collection of variable-value pairs.

3.8.1 Obtaining test events

A test engineer inspects the rule base and must create test events according to the event types, which appear in the body of reaction rules. Reaction rules are triggered by the inference engine on the base of these test events. Below is an example of an R2ML Message Event which can appear in the body of a reaction rule i.e. the triggered event.

Example 3.1 (A returnCar event)

```
<r2ml:MessageEventExpression
         xmlns:eurent="http://www.eurent.org"
         xmlns:events="http://www.eurent.org/events"
         r2ml:eventType="events:returnCar"
         r2ml:startTime="2006-09-21T09:00:00"
```

```
         r2ml:duration="P0Y0M0DT0H0M0S"
         r2ml:sender="http://www.eurent.org">
 <r2ml:arguments>
  <r2ml:ObjectVariable r2ml:name="car" r2ml:classID="eurent:RentalCar"/>
  <r2ml:ObjectVariable r2ml:name="customer" r2ml:classID="eurent:Customer"/>
 </r2ml:arguments>
</r2ml:MessageEventExpression>
```

An example of a corresponding test event is:

```
<TestEvent id="E001">
<r2ml:MessageEventExpression
         xmlns:eurent="http://www.eurent.org"
         xmlns:events="http://www.eurent.org/events"
         xmlns:cars="http://www.eurent.org/cars"
         xmlns:customers="http://www.eurent.org/customers"
         r2ml:eventType="events:returnCar"
         r2ml:startTime="2006-09-21T09:00:00"
         r2ml:duration="P0Y0M0DT0H0M0S"
         r2ml:sender="http://www.mywebsite.org">
 <r2ml:arguments>
  <r2ml:ObjectName r2ml:objectID="cars:car10110"
                   r2ml:classID="eurent:RentalCar"/>
  <r2ml:ObjectName r2ml:name="customers:customer1001"
                   r2ml:classID="eurent:Customer"/>
 </r2ml:arguments>
</r2ml:MessageEventExpression>
</TestEvent>
```

On the base of this test event the validator generates an event of the type returnCar. The inference engine triggers the rule according with this event and binds the existent variables to specific values. In the above example, the object variable *car* is bound to the object name *cars:car10110*.

The general template is simpler i.e. for any event type from the vocabulary (in the above example, a returnCar event) one or more test assertions are generated according to the following pattern:

```
<TestEvent id=$idValue>
<r2ml:MessageEventExpression
  <!--
  Necessary namespaces declarations for example
   xmlns:eurent="http://www.eurent.org"
   xmlns:events="http://www.eurent.org/events"
  -->
  [Namespaces]
  <!-- the event type name for example, returnCar -->
   r2ml:eventType=$eventTypeName
  <!-- the start time attribute, for example
       "2006-09-21T09:00:00" i.e. Sept. 21, 2006 at 9 am -->
```

```
   r2ml:startTime=$startTimeValue
  <!-- Since is a message event i.e. an atomic one then duration is always 0 -->
   r2ml:duration="P0Y0M0DT0H0M0S"
  <!-- The sender value in a form of an URI, as in
       the above example http://www.mywebsite.org -->
         r2ml:sender=$senderValue
 <r2ml:arguments>
  <!-- a number of terms of the corresponding types according
       with the vocabulary schema of the event type -->
   $Term1
   $Term2
    ....
 </r2ml:arguments>
</r2ml:MessageEventExpression>
</TestEvent>
```

Everywhere above `$XXX` denotes a metavariable encoding the desired value. For example, if we consider that: `$idValue:="E001"`
`$eventTypeName:="returnCar"`
`$senderValue="http://www.eurent.org"`
`$Term1` denotes the element
`<r2ml:ObjectName r2ml:objectID="cars:car10110" r2ml:classID="eurent:RentalCar"/>`
and
`$Term2` denotes the element
`<r2ml:ObjectName r2ml:name="customers:customer1001" r2ml:classID="eurent:Customer"/>`
we obtain the test event from the Example 3.1.

3.8.2 Obtaining test actions

See the Section 3.7.2.

3.8.3 Obtaining test queries from postconditions

The generation of test queries according with rules postconditions is similar with the generation of test queries for conclusions in the case of derivation rules.

3.8.4 A Sample R2ML Test Suite for Reaction Rules

Consider the following reaction rule in R2ML XML syntax:

Example 3.2 (Sending cars to service) *"If a customer returns a car and the car has more than 5000km since the last service then send the car to the service..."*

```
<r2ml:ReactionRule r2ml:ruleID="ECA001"
          xmlns:eurent="http://www.eurent.org"
         xmlns:events="http://www.eurent.org/events"
         xmlns:cars="http://www.eurent.org/cars"
         xmlns:customers="http://www.eurent.org/customers">
<r2ml:triggeringEvent>
```

```
 <r2ml:MessageEventExpression r2ml:eventType="events:returnCar"
         r2ml:startTime="20060321T09:00:00"
         r2ml:duration="P0Y0M0DT0H0M0S"
         r2ml:sender="http://www.eurent.org">
  <r2ml:arguments>
   <r2ml:ObjectVariable r2ml:name="car" r2ml:classID="cars:RentalCar"/>
   <r2ml:ObjectVariable r2ml:name="customer" r2ml:classID="customers:Customer"/>
  </r2ml:arguments>
 </r2ml:MessageEventExpression>
</r2ml:triggeringEvent>
<r2ml:conditions>
 <r2ml:DatatypePredicateAtom r2ml:datatypePredicateID="swrlb:greaterThan">
  <r2ml:dataArguments>
   <r2ml:AttributeFunctionTerm r2ml:attributeID="cars:lastservice">
    <r2ml:contextArgument>
     <r2ml:ObjectVariable r2ml:name="car" r2ml:classID="cars:RentalCar"/>
    </r2ml:contextArgument>
   </r2ml:AttributeFunctionTerm>
   <r2ml:AttributeFunctionTerm r2ml:attributeID="cars:odometer_reading">
    <r2ml:contextArgument>
      <r2ml:ObjectVariable r2ml:name="car" r2ml:classID="cars:RentalCar"/>
    </r2ml:contextArgument>
   </r2ml:AttributeFunctionTerm>
   <r2ml:TypedLiteral r2ml:datatypeID="xs:positiveInteger" r2ml:lexicalValue="5000"/>
  </r2ml:dataArguments>
 </r2ml:DatatypePredicateAtom>
</r2ml:conditions>
<r2ml:producedAction>
 <r2ml:InvokeActionExpression r2ml:operationID="cars:service">
  <r2ml:contextArgument>
   <r2ml:ObjectVariable r2ml:name="car" r2ml:classID="cars:RentalCar"/>
  </r2ml:contextArgument>
 </r2ml:InvokeActionExpression>
</r2ml:producedAction>
<r2ml:postcondition>
 <r2ml:ObjectClassificationAtom r2ml:classID="cars:RentalCarScheduledForService">
  <r2ml:ObjectVariable r2ml:name="car"/>
 </r2ml:ObjectClassificationAtom>
</r2ml:postcondition>
</r2ml:ReactionRule>
```

Possible test events are:

Example 3.3 (Test events)

```
<TestEvents>
<TestEvent id="E001">
<r2ml:MessageEventExpression
```

```
         r2ml:eventType="events:returnCar"
         r2ml:startTime="2006-09-21T09:00:00"
         r2ml:duration="P0Y0M0DT0H0M0S"
         r2ml:sender="http://www.mywebsite.org">
 <r2ml:arguments>
  <r2ml:ObjectName r2ml:objectID="cars:car10110"
                   r2ml:classID="eurent:RentalCar"/>
  <r2ml:ObjectName r2ml:name="customers:customer1001"
                   r2ml:classID="eurent:Customer"/>
 </r2ml:arguments>
</r2ml:MessageEventExpression>
</TestEvent>
<TestEvent id="E002">
<r2ml:MessageEventExpression
         r2ml:eventType="events:requestCar"
         r2ml:startTime="2006-09-22T08:15:00"
         r2ml:duration="P0Y0M0DT0H0M0S"
         r2ml:sender="http://www.mywebsite.org">
 <r2ml:arguments>
  <r2ml:ObjectName r2ml:objectID="cars:car10111"
                   r2ml:classID="eurent:RentalCar"/>
  <r2ml:ObjectName r2ml:name="customers:customer1001"
                   r2ml:classID="eurent:Customer"/>
 </r2ml:arguments>
</r2ml:MessageEventExpression>
</TestEvent>
</TestEvents>
```

The first one is expected to generate an event which triggers the reaction rule from Example 3.2 and the second one to generate an event, which will not trigger the rule, since the type of the event from the test event is different than the type of the event from the rule event.

Below we provide a possible simple test assertions for the rule from Example 3.2.

Example 3.4 (Test assertions)

```
<testAssertions>
 <R2MLFormula id="A001">
  <r2ml:DatatypePredicateAtom r2ml:datatypePredicateID="swrlb:greaterThan">
   <r2ml:dataArguments>
    <r2ml:AttributeFunctionTerm r2ml:attributeID="cars:lastservice">
     <r2ml:contextArgument>
      <r2ml:ObjectName r2ml:objectID="car10110" r2ml:classID="cars:RentalCar"/>
     </r2ml:contextArgument>
    </r2ml:AttributeFunctionTerm>
    <r2ml:AttributeFunctionTerm r2ml:attributeID="cars:odometer_reading">
     <r2ml:contextArgument>
      <r2ml:ObjectName r2ml:objectID="car10110" r2ml:classID="cars:RentalCar"/>
     </r2ml:contextArgument>
```

```
    </r2ml:AttributeFunctionTerm>
    <r2ml:TypedLiteral r2ml:datatypeID="xs:positiveInteger" r2ml:lexicalValue="5000"/>
   </r2ml:dataArguments>
  </r2ml:DatatypePredicateAtom>
 </R2MLFormula>
 <R2MLFormula id="A002">
  <r2ml:DatatypePredicateAtom r2ml:datatypePredicateID="swrlb:greaterThan">
   <r2ml:dataArguments>
    <r2ml:AttributeFunctionTerm r2ml:attributeID="cars:lastservice">
     <r2ml:contextArgument>
      <r2ml:ObjectName r2ml:objectID="car10110" r2ml:classID="cars:RentalCar"/>
     </r2ml:contextArgument>
    </r2ml:AttributeFunctionTerm>
    <r2ml:AttributeFunctionTerm r2ml:attributeID="cars:odometer_reading">
     <r2ml:contextArgument>
      <r2ml:ObjectName r2ml:objectID="car10110"/>
     </r2ml:contextArgument>
    </r2ml:AttributeFunctionTerm>
    <r2ml:TypedLiteral r2ml:datatypeID="xs:positiveInteger" r2ml:lexicalValue="1000"/>
   </r2ml:dataArguments>
  </r2ml:DatatypePredicateAtom>
 </R2MLFormula>
 <R2mlFormula id="A003">
  <r2ml:ObjectClassificationAtom r2ml:classID="cars:RentalCarScheduledForService">
   <r2ml:ObjectName r2ml:objectID="car10110"/>
  </r2ml:ObjectClassificationAtom>
 </R2mlFormula>
 </testAssertions>
```

The reader may notice that the first assertion will fit with the rule condition contrary to the second one which will not. The last assertion concerns the postcondition of the rule from Example 3.2.

The query part of our validation test regards the test action effects related to the actions and with test queries related to our postconditions.

Example 3.5 (Test action effects and test queries for postconditions)

```
<TestItem expectedAnswer="yes">
 <testActionEffect>
  <R2MLFormula>
   <r2ml:ObjectDescriptionAtom r2ml:classID="cars:RentalCar">
    <r2ml:subject>
     <r2ml:ObjectName r2ml:objectID="car10110"/>
    </r2ml:subject>
    <r2ml:DataSlot r2ml:attribute="cars:inService">
     <r2ml:dataValue>
      <r2ml:TypedLityeral r2ml:lexicalValue="true" r2ml:datatypeID="xs:boolean"/>
     </r2ml:dataValue>
```

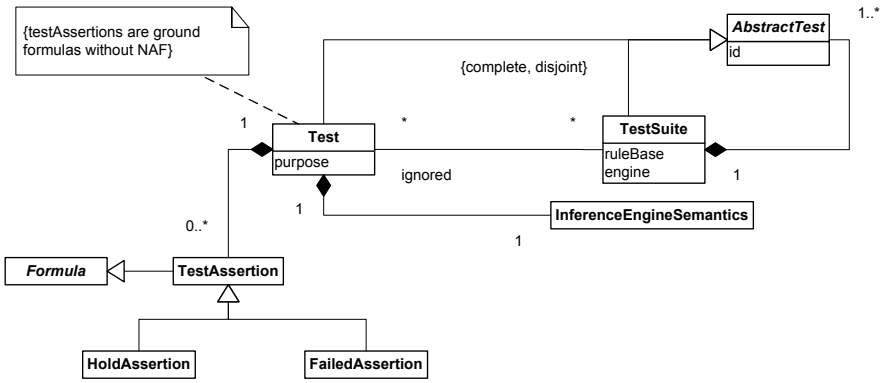

Figure 3.2: Test metamodel for integrity rules

```
    </r2ml:DataSlot>
   </r2ml:ObjectDescriptionAtom>
  </R2MLFormula>
 </testActionEffect>
 <testQuery>
  <R2MLFormula>
   <r2ml:ObjectClassificationAtom r2ml:classID="cars:RentalCarScheduledForService">
    <r2ml:ObjectName r2ml:objectID="car10110"/>
   </r2ml:ObjectClassificationAtom>
  </R2MLFormula>
 </testQuery>
</TestItem>
```

The scenario of validation consists of the following main steps:

1. The validator passes the test and the rule base to the rule engine;
2. The validator generates events on the base of the test events;
3. According to the specific events occurrence reaction rules are triggered and variables are bounded according to event data;
4. The validator performs test queries and test action effects after the rule engine execution;

3.9 Testing Integrity Rules

The test metamodel for integrity rules (Figure 3.2) is different from the derivation and production rules in a way there are no test queries and expected results, but there are two test assertion types: *HoldAssertion* and *FailedAssertion*.

We say that an integrity rule is valid if it holds on HoldAssertion's and does not hold on the other FailedAssertion's.

The validator for integrity rule bases works as follows. It passes the integrity rule base with a list of test assertions to the rule engine. If rules are valid, the rule engine returns that they hold on HoldAssertion's and fail on FailedAssertion's.

Bibliography

[Ant97] G. Antoniou. Verification and correctness issues for nonmonotonic knowledge bases. *International Journal of Intelligent Systems*, 12:725–738, 1997.

[Bec99] K. Beck. Extreme programming. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 411. IEEE Computer, 1999.

[Bec00] K. Beck. *Extreme programming explained: embrace change*. Addison Wesley, Boston, MA, USA, 2000.

[BK93] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In *International Conference on Logic Programming*, pages 257–279, 1993.

[BL96] Chitta Baral and Jorge Lobo. Formal characterization of active databases. In *Logic in Databases*, pages 175–195, 1996.

[DM02] P.M. Dung and P. Mancarella. Production systems with negation as failure. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):336–352, 2002.

[DP05] Jens Dietrich and Adrian 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.

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[GHJV95] 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.

[Lau98] B. Ludascher W. May Lausen, G. On logical foundations of active databases. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 389–422. Kluwer Academic Publishers, 1998.

[Lay04] 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.

[Lud98] B. Ludascher. *Integration of Active and Deductive Database Rules*. PhD thesis, University of Freiburg, Germany, 1998.

[Mak89] 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.

[McC76] 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.

[McC80] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *AI*, 13(1+2):27–39+171–172, 1980.

[Pas05a] A. Paschke. Eca-lp: A homogeneous event-condition-action logic programming language. Technical report, Internet-based Information Systems, Technical University Munich, November 2005. http://ibis.in.tum.de/staff/paschke/rbsla/.

[Pas05b] A. Paschke. Eca-ruleml: An approach combining eca rules with interval-based event logics. Technical report, Internet-based Information Systems, Technical University Munich, November 2005. http://ibis.in.tum.de/staff/paschke/rbsla/.

[Plo70] G.D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5, 1970.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[PS94] A.D. Preece and R. Shinghal. Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, 9:683–701, 1994.

[Ras92] Louiqa Raschid. A semantics for a class of stratified production system programs. Technical report, College Park, MD, USA, 1992.

[SF98] S. Greco S. Flesca. Declarative semantics for active rules. In *Proc. of Database and Expert System Applications*. Springer, 1998.

[SK90] M. Magidor S. Kraus, D. Lehmann. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44:167–207, 1990.

[VEBS89] J. Bachant Virginia E. Barker, Dennis E. O'Connor and E. Soloway. Expert systems for configuration at digital: Xcon and beyond. *Communication of the ACM*, 32:292–318, 1989.

[WGL06] G. Wagner, A. Giurca, and S. Lukichev. Rewerse i1 deliverable d8, language improvements and extentions. Technical report, 2006.

[Zan94] C. Zaniolo. A Unified Semantics for Active and Deductive Databases. In N. W. Paton and M. H. Williams, editors, *Proceedings of the 1st International Workshop on Rules in Database Systems*, pages 271–287. Springer, 1994.

[Zan95] Carlo Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *Deductive and Object-Oriented Databases*, pages 55–72, 1995.