The Declarative Approach for Anomaly Detection in Production Rule Bases with Semantic Constraints

Sergey Lukichev

Institute of Informatics, Brandenburg University of Technology at Cottbus
slukichev@canto.de

Abstract. In this paper we present a rule-based (declarative) approach for rule verification. We focus on anomalies, which may appear in rule bases, containing production rules and semantic constraints. The presented approach defines special rules, called verifier rules, which look for anomalies in business rules. The approach is flexible and easy to maintain in the sense that verifier rules can easily be added or modified if new anomalies are found and have to be detected. Examples of verifier rules in Drools syntax are given at the end of the paper.
Keywords: *Verification, integrity constraints, Jena, Drools.*

1 Introduction

It is widely recognized that the process of verification is an important part of quality assurance for rule systems. The common definition of the term "verification" is given in [10]: verification of rules includes checking the knowledge base for logical anomalies such as redundant rules, contradictory rules, and missing rules. Such verification is called anomaly detection. It is important to note that mentioned anomalies are not necessarily errors, but rather potential errors, also known as "attention points ... that might give hints for incorrect rules or wrongly expressed knowledge"[13]. Anomalies may appear in rule bases, for instance, as a result of rules refactoring or due to various communication problems when the knowledge is incorrectly transferred from the business expert to the knowledge engineer. In this paper we focus on two cases of the ambivalence anomaly [10]: The ambivalent rule pair anomaly and the semantic constraint violation by condition and postcondition.

We choose these anomalies because they may appear in rule bases, which contain semantic constraints: special rules, which define an inconsistent state of the knowledge. The main novelty of the work is the use of the declarative approach for anomaly detection. The core of the presented approach are verifier rules, which we introduce in Section 2.2. These verifier rules detect anomalies in production rules, expressed in terms of the production rule metamodel, which is defined in Section 2.3. In Section 3 we define anomalies in production rule bases by means of the model theory. We provide verifier rules, which detect previously defined anomalies in Section 4. In Section 5 we discuss soundness and

completeness of the presented approach. Section 6 references related works on rule verification and concludes the paper.

2 Production Rules and Verifier Rules

Before we start introducing the declarative rule verification approach, we explain what is a knowledge base with production rules and semantic constraints. We define a production rule metamodel (Section 2.3), which is used later in Section 4 on verification as the vocabulary for verifier rules. We introduce the concept of verifier rules and their advantages in Section 2.2.

2.1 Production Rule

According to the OMG Production Rules Representation [3], a production rule is "a statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied" of the form:

if [condition] then [action-list]

We define the condition part of the production rule as a literal conjunction (a conjunction of atoms or negation of atoms) and denote it as $cond(r)$, here r is a production rule. The rule action part defines an ordered list of actions. The OMG PRR [3] defines three state changing actions: Update action, assert action, and retract action. In our approach we consider only two types of actions: Assert and retract. An update action is implemented by a sequence of retract action (remove old facts) and assert action (assert new facts). This consideration with two types of action is common for some rule languages, for instance, Jena 2. The action part of the production rule is also called a postcondition and denoted as $pcond(r)$ where r is a production rule. In this paper a postcondition is a literal conjunction, where a positive atom means the assert action and a negative atom means the retract action.

Definition 1 (Knowledge Base). *A production rule base R together with a vocabulary V of R, semantic constraints C, and a fact set X forms a* knowledge base $KB = \langle X, V, C, R \rangle$.

The fact set X consists of a set of ground facts (variable-free atoms). It is important to note that in our verification approach anomalies are detected by examining the syntax of a rule base and the content of the fact set is not taken into account.

2.2 Verifier Rules for Anomaly Detection

We distinguish between business rules, which have to be verified, and verifier rules, which are used for business rules verification.

A *verifier rule* is a production rule, which is executed when a business rule base contains an anomaly and generates a report about it.

A *rule-based verification approach* employs verifier rules for detecting anomalies in business rules.

The declarative verification approach has a number of advantages such as: *i).* Simplicity of implementation. Anomalies are described by means of special rules, called verifier rules, which are executed by a rule engine. It means that rule-based verification is about writing verifier rules, which is easier than developing and implementing algorithms; *ii).* Easiness of maintenance. When new anomalies are discovered, new verifier rules can be added easily in order to detect them. No additional anomaly-detection programming and algorithm development is required; *iii).* Support for various rule languages. Verifier rules are expressed in terms of a generic rule metamodel, which, if general and flexible enough, can be used for various rule languages.

An example of a verifier rule, expressed informally is "If the body of a rule contains a conjunction of an atom and its negation then this rule contains *unsatisfied condition anomaly*". In contrast to business rules, which are based on terms from a business vocabulary, the vocabulary of verifier rules is based on the rule metamodel. In our example terms "atom", "negation", and "rule body" are from the rule metamodel and not from a business domain.

We express verifier rules using Drools syntax and execute them in the Drools rule engine, however, any other rule language and an engine can be used.

We assume that business rules are verified at the design/development stage [6] and verifier rules are not placed into the system with business rules.

2.3 Production Rule Metamodel

Business rules can be formalized as production rules. Verifier rules, which we present in this paper, are used to find anomalies in production rule bases. In this work we consider a sample metamodel for production rules(Figure 1), which describes the structure of a production rule base with semantic constraints. The metamodel is very close to the metamodel of Jena rules [2], where a rule condition is a conjunction of either triple atoms or built-in atoms. Semantic constraints in Jena can be expressed by means of OWL DL. Therefore, verifier rules, expressed in terms of the metamodel, depicted in Figure 1, can verify Jena rules. Production rule conditions have the expressive power of RDF: to represent the binary existential-conjunctive fragment of predicate calculus.

A rule set consists of rules. A rule has a name and a unique identifier. It has a list of atoms, interpreted as conjunction, as a head and a list of atoms as a body. An abstract class Atom has an identifier and refers to the rule it belongs to.

We distinguish two types of atoms: The BuiltinAtom, which represents various built-ins and TripleAtom. The TripleAtom class has the attribute isNegated and refers to the abstract class Term, which represents subject, predicate and object of the triple atom. Class TripleAtom can represent Jena triples. We interpret a negation as an absence of a fact in the working memory, which is the common interpretation of negation in production rule systems, for instance, in Jena and JBoss Rules.

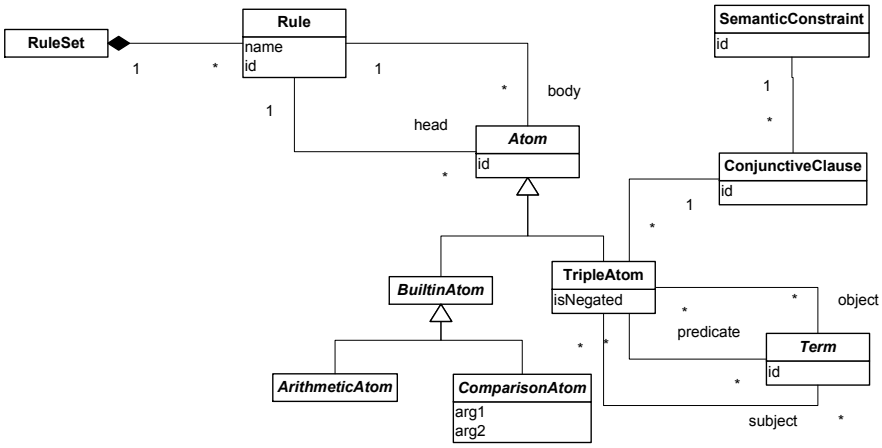

Fig. 1. The generic rule metamodel

A BuiltinAtom is either an ArithemticAtom, which represents various built-ins for arithmetic functions, or a ComparisonAtom, which represents comparison built-ins such as, for instance, greaterThan or lessThan. A SemanticConstraint is a list of ConjunctiveClause's with meaning of disjunction of clauses.

The idea of such structure is to represent constraints in the disjunctive normal form. It allows relatively simple verifier rules for anomaly detection. The practical goal is to verify business rules, expressed in Jena against semantic constraints, expressed in some Description Logic language, for instance, OWL DL. In this paper we consider semantic constraints in the disjunctive normal form and do not explain in details how OWL DL axioms are transformed into logical formulae in DNF. Issues of translating from Description Logic into predicate logic are described in various books and articles ([4], [14]). For instance, in [4] the expressive power of DL is compared against the expressive power of predicate calculus. In particular, it describes the translation from descriptions to predicate calculus, which introduces new variables whenever a new quantifier appears. Here we only give an example of an OWL axiom and corresponding formula in DNF. Let us consider a constraint: *Every eligible driver must have a training certificate*. It is formalized by means of the following OWL DL axiom:

$$\text{Class}(\text{EligibleDriver}, \text{restriction}(\text{hasCert}, \text{someValuesFrom}(\text{TrainingCert})))$$

This DL axiom corresponds to the following predicate logic formula:

$$\forall x(\text{EligibleDriver}(x) \rightarrow \exists y|\text{hasCert}(x, y) \wedge \text{TrainingCert}(y))$$

This implication is equivalent to the following formula in the disjunctive normal form:

$$\forall x \exists y(\neg\text{EligibleDriver}(x) \vee \neg\text{hasCert}(x, y) \vee \neg\text{TrainingCert}(y))$$

A translation of an OWL axiom to the disjunctive normal form is always possible. Existentially quantified variables in the resulting formula in DNF are replaced using skolemization: an existentially quantified variable x is replaced by a function, which takes as parameters all universally quantified variables that precedes x in the formula's quantified variables list.

3 Ambivalence Anomaly

In this section we define the ambivalence anomaly [10], using the model theory. Verifier rules for ambivalence detection are more complex in the sense that they have more atoms in conditions than verifier rules for anomalies, where semantic constraints are not involved (for instance, subsumption, contradictions).

3.1 Semantic Constraint Violation

First, we define a violation of a semantic constraint by a general logical formula. In the following $\models$ denotes the satisfaction relation. $\mathcal{I}_\mathcal{V}$ denotes an interpretation over some valuation.

Definition 2 (Semantic constraint violation by a logical formula). *A formula F* violates semantic constraint *c if the following holds for all interpretations $\mathcal{I}$:*

$$\mathcal{I}_\mathcal{V} \models^t F \text{ then } \mathcal{I}_\mathcal{V} \models^f c$$

In words, if the formula holds then the constraint does not hold. As an example, let us consider a production rule **PR** and a constraint **Constr**:

PR: Eligible(?driver) $\wedge\neg$hasTrainingCertification(?driver, 'true') $\Rightarrow$ HighRiskDriver(?driver)
Constr: Eligible(?driver) $\wedge$ hasTrainingCertification(?driver, 'true')

It is obvious that if the condition of PR holds, then Constr does not hold. The business problem, caused by this anomaly is that the rule with such condition is meaningless since existence of facts, on which it holds, is prohibited by the constraint. Or, in other words, the condition of such rule is never satisfied. It is important to note, that in the example above such variable as ?driver is local at the rule level. The condition of the rule violates the constraint only when the variable is unified with the same value in both the rule and the constraint.

Ambivalent Rule Pair. A rule pair is ambivalent if the condition of one rule subsumes the condition of the other and the conjunction of their postconditions violates the semantic constraint.

Definition 3. *A rule pair $r_1, r_2 \in R$ is* ambivalent *if $cond(r_1)$ subsumes $cond(r_2)$ and there is a semantic constraint $c \in C$ such that the conjunction of postconditions $pcond(r_1) \wedge pcond(r_2)$ violates c.*

The more general case of this anomaly is when conditions of two rules have a non empty intersection, which may lead to firing of both rules for some state.

Let us consider production rules PR1 and PR2 and the semantic constraint Constr1:

PR1 age(?driver, ?x) ∧ ?x> 26 ⇒NormalDriver(?driver)
PR2 age(?driver, ?x) ∧ ?x> 70 ⇒SeniorDriver(?driver)
Constr1 ¬(SeniorDriver(?driver) ∧ NormalDriver(?driver))

The condition of PR1 subsumes the condition of PR2, however, their right-hand sides are different. It means that if these rules are executed, the fact base will contain two facts: "The driver is a normal driver" and "the driver is a senior driver", which is prohibited by the semantic constraint.

Semantic Constraint Violation by Condition and Postcondition A production rule has a semantic constraint violation by condition and postcondition if the state of the fact base after execution of the rule violates a semantic constraint.

Definition 4. *A production rule r has a* semantic constraint violation by condition and postcondition *if exists $c \in C | cond(r) \wedge pcond(r)$ violates c.*

Let us consider the production rule PR3 and the constraint Constr2:

PR3 Provisional(?car) ∧ age(?car, ?x) ∧ ?x> 3 ⇒NotEligible(?car)
Constr2 ¬(NotEligible(?car) ∧ Provisional(?car))

If PR3 is executed then the fact base contains two facts: "a car is provisional" and "a car is not eligible", which is prohibited by the semantic constraint.

4 Verifier Rules for Ambivalence Detection

In this section we define verifier rules in two ways: We give a semi-formal definition of a verifier rule in English, and then express the rule using Drools syntax. Drools syntax ([1]) is technical, but readers, who are familiar with Java programming language and predicate logic can easily understand it. In addition, we give explanations and comment every verifier rule, expressed in Drools syntax. We remind, that semantic constraints are in DNF, i.e. consist of disjunction of conjunctive clauses.

Verifier Rule 1 (Ambivalent rule pair). A rule pair r_1, r_2 is ambivalent if there is a semantic constraint c such that:

1. cond(r_1) subsumes cond(r_2);
2. Every conjunctive clause of c contains an atom, which is oppositely negated to some atom a either from pcond(r_1) or from pcond(r_2);
3. In pcond(r_1) exists an atom, which is in c;
4. In pcond(r_2) exists an atom, which is in c.

Condition 1 is important in order to make sure that rules can be executed on the same facts (See [10] for definition of subsumption). Condition 2 in this rule checks whether constraint c is violated by either atoms from the head of rule 1 or from the head of rule 1. However, this does not guaranty that the conjunction of rule heads violates c, since it is possible that the constraint is violated by the head of just one rule. For instance, if the head of the rule is $A \wedge B$ and constraint is $\neg A \wedge \neg B$ then the constraint is violated no matter of the head of the second rule. Additional conditions 3 and 4 guarantee that each rule head contains at least one oppositely negated atom from c and, therefore, a conjunction of rule heads violates the constraint.

```
rule "Ambivalent rule pair"
    when
        $sc :SemanticConstraint()

        $r1 :Rule()
        $r2 :Rule(id != r1.id)

        # Check that the body of r1 subsumes the body of r2
        forall(
           $atom: Atom(ruleId == $r1.id, body == true)
           DuplicateAtom(
                left == $atom,
                right memberOf $r2.body
           )
        )

        # Check that every conjunctive clause is violated
        # by conjunction of pcond(r1) and pcond(r2)
        forall(
           $clause:ConjunctiveClause(constrId == $sc.id)

           exists(
              $triple:TripleAtom(clauseId == $clause.id)
              or(
                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r1.head
                   )

                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r2.head
                   )
              )
           )
        )

        # At least one atom from the head of r1 and r2
        # must be in some conjunctive clause of sc
        exists(
           $a1:Atom(ruleId == $r1.id, body == false)

           OppositelyNegatedAtoms(
                 left memberOf $sc,
                 right == $a1
           )
        )
        exists(
           $a1:Atom(ruleId == $r2.id, body == false)

           OppositelyNegatedAtoms(
                    left memberOf $sc,
                    right == $a1
           )
        )
```

```
    then insert(new AmbivalentRulePair( $r1, $r2, $sc ));
end
```

We have to explain some predicates, used in the rule above. `DuplicateAtom` checks for atoms with the same subject, predicate and object. Technically, such atoms can be derived by additional rules of the verifier rule base before the verifier rule is executed. We call such rules "supplementary rules" since they do not detect anomalies, but derive intermediate facts, needed by verifier rules. The predicate `OppositelyNegatedAtoms` checks for conjunctions of an atom and its negation (for instance, $p(s, o) \wedge \neg p(s, o)$). The subsumption check in the rule above works for triple atoms and for built-ins it is different.

Verifier Rule 2 (Semantic constraint violation by condition and postcondition). A rule r violates semantic constraint c by condition and postcondition if

1. Each atom of every conjunctive clause of c has an oppositely negated atom either in $\mathrm{cond}(r)$ or in $\mathrm{pcond}(r)$;
2. At least one atom of $\mathrm{pcond}(r)$ is in c;
3. At least one atom of $\mathrm{cond}(r)$ is in c.

First condition checks whether condition or postcondition of the rule violates every conjunctive clause of the constraint, and therefore, violates the constraint. Conditions 2 and 3 check that the conjunction of condition and postcondition violates the constraint. This verifier rule in Drools syntax:

```
rule "semantic constraint violation by condition and postcondition"
    when
        $sc :SemanticConstraint()

        $r :Rule()

        forall(
           $clause:ConjunctiveClause(constrId == $sc.id)
           exists(
              $triple:TripleAtom(clauseId == $clause.id)
              or(
                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r.body
                   )

                   OppositelyNegatedAtoms(
                     left == $triple,
                     right memberOf $r.head
                   )
              )
           )
        )

        # At least one atom from the the body of r and the head of r2
        # must be in some conjunctive clause of sc
        exists(
           $a:Atom(ruleId == $r.id, body == false)

           OppositelyNegatedAtoms(
                 left memberOf $sc,
                 right == $a
           )
        )
```

```
        exists(
           $a1:Atom(ruleId == $r.id, body == true)

           OppositelyNegatedAtoms(
                    left memberOf $sc,
                    right == $a
           )
        )
    then insert( new SemcCVByCondPcond( $r, $sc ));
end
```

5 Soundness and Completeness

The discussion concerning soundness and completeness of the proposed verification solution is about relations between model-theoretic definitions of anomalies (Section 3) and verifier rules, which detect defined anomalies (Section 4). Informally, a solution is sound if it solves the problem for which it is developed. In our case, the solution is a set of verifier rules for the detection of different anomalies. The converse of the soundness property is the completeness property. A solution is complete if its set of verifier rules detects all possible anomalies.

As an example, let us check the soundness of the Verifier Rule 1. Let $A(r_1, r_2, c)$ be an anomaly, detected by Verifier Rule 1 and r_1, r_2 are rules and c is the semantic constraint. We say that this verifier rule is *sound* if the following two conditions hold:

1. r_1 subsumes r_2;
2. $pcond(r_1) \wedge pcond(r_2)$ violates c (as it is defined in Definition 3).

The first condition is based on the definition of rule subsumption. Here we assume that the subsumption of rules is sound. The second condition follows from conditions 2,3,4 of Verifier Rule 1 since these conditions check that $pcond(r_1) \wedge pcond(r_2)$ violates every conjunctive clause of c and, therefore, violates c.

The presented verification approach does not guarantee **completeness**: As it is stated in [8] and further supported in the tutorial and survey on rule verification by O'Keefe [9], "verification, based upon anomaly detection is a heuristic approach, rather than deterministic, for two reasons. First, detected anomalies may not be errors, and errors may exist that are not related to known anomalies. Second, some of the methods for detecting anomalies are themselves heuristic, and thus do not guarantee detection of all identifiable anomalies".

6 Related Works and Conclusions

In this paper we have presented the rule-based approach for production rule verification. The approach is based on verifier rules, which, been expressed in terms of the rule metamodel, analyze the business rule base and derive certain facts if anomalies are detected. We have also demonstrated two verifier rules, which detect two particular types of the ambivalence anomaly: The ambivalent rule pairs anomaly and the semantic constraint violation by condition and postcondition.

There are a lot of works on rule verification, for instance, [5], [11] and others. However, the most related work is the Drools verifier from JBoss [12]. This verifier employs similar declarative approach for anomaly detection. It uses Drools syntax for expressing both verifier rules and production rules. The main difference with the presented approach is that JBoss verifier does not check for anomalies, which include semantic constraints. This is mainly because Drools does not support constraints yet.

The presented verification approach is implemented in the Jena Rule Verifier [7]. The verifier already checks for a number of anomalies and the upcoming further work is towards extensions of its functionality for detection of various anomalies, which include semantic constraints.

References

1. Business Rules Management System Drools. Project homepage. http://www.jboss.org/drools.
2. Jena - A Semantic Web Framework for Java. Project homepage. http://jena.sourceforge.net.
3. Production Rule Representation (PRR). OMG Adopted Specification. http://www.omg.org/spec/PRR/1.0/.
4. Alex Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82:353–367, 1996.
5. G. Castore. Validation and verification for knowledge based control systems. In *Proceedings of the First Annual Workshop on Space Operations, Automation and Robotics, NASA,*, pages 197–202, 1987.
6. Antoni Ligeza. *Logical Foundations for Rule-Based Systems (Studies in Computational Intelligence) (Studies in Computational Intelligence)*. Springer-Verlag New York, Inc., 2006.
7. Sergey Lukichev. The jena rule verifier project, 2009. https://sourceforge.net/projects/jenaruleverifie.
8. L. A. Miller. Dynamic testing of knowledge bases using the heuristic testing approach. *Expert Systems with Applications*, 1(3):249–269, 1990.
9. Robert M. O'Keefe and Daniel E. O'Leary. Expert system verification and validation: a survey and tutorial. *Artificial Intelligence Review*, 7(1):3–42, 1993.
10. A. D. Preece. Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, 9(8):683–701, 1994.
11. A. D. Preece, R. Shinghal, and A. Bakarekh. Verifying expert systems: a logical framework and a practical tool. *Exp. Stst. Appl.*, 5:421–436, 1992.
12. Toni Rikkola. Rule analytics module, 2008. http://www.jboss.org/community/docs/DOC-11890.
13. Silvie Spreewenberg. Using verification and validation techniques for high-quality business rules. *Business Rules Journal*, 4(2), 2003.
14. Pascal Hitzler Rudi Studer and York Sure. Description logic programs: A practical choice for the modelling of ontologies. In *1st WS on Formal Ontologies meet Industry*, 2005.