Inferring Business Rules from Natural Language Expressions

Giovanni Aiello, Roberto Di Bernardo, Martino Maggio
Research and Development Laboratory
Engineering Ingegneria Informatica S.p.A
Palermo, Italy
{giovanni.aiello,roberto.dibernardo,martino.maggio}@eng.it

Daniele Di Bona, Giuseppe Lo Re
University of Palermo
Palermo, Italy
danieledibona@gmail.com, giuseppe.lore@unipa.it

Abstract—This paper proposes a mapping technique for automatically translating rules expressed in a format based on natural language, i.e. Semantics of Business Vocabulary and Business Rules (SBVR) standard, into production rules that can be executed by a computer (i.e. rule engine). The proposed approach achieves a twofold purpose: on the one hand non IT skilled people (i.e. domain expert) can effectively focus on business rules definition by using statements in natural language, and on the other hand the IT staff will have to manage business rules in a format ready to be executed by a rule engine. The main goal is to overcome some weaknesses in the software development process that could produce inconsistencies between the domain requirements identification and the implemented software functionalities. An exhaustive analysis of the mapping technique is provided and a real case study is presented in order to prove the validity of our work.

Keywords — Business Rules, SBVR, Natural Language, Drools

I. INTRODUCTION

In software engineering, the formalization of domain requirements towards code implementation involves two main kinds of actors: a) Domain experts – they have deep knowledge about the domain context, they are not required to have specific IT skills; b) IT developers – they have technical skills concerning software development, but, typically, they do not have specific knowledge of the application domain. This situation generates some weakness points in software development, because domain requirements could be erroneously interpreted and, also, implemented functionalities could not reflect the designed ones. In respect to this, an automatic interpretation and implementation of requirements can make the overall process more straightforward. To start addressing this aspect, in a previous work [1] we proposed a methodology capable of graphically modeling business rules, so giving non IT experts a tool to define production rules by using standards notation such as Business Process Modeling Notation (BPMN) 1.

Within this work, we propose a technique that is able to automatically translate business rules expressed in a structured natural language method, i.e. Semantics of Business Vocabulary and Rules (SBVR) standard [2], into executable production rules that can be interpreted by a rule engine (i.e. JBoss Drools 2 rules). Our approach can be synthetized as follows:

1) The definition of domain rules and vocabulary expressed in SBVR Structured English 3 by using an ontological approach similar to those reported in [3] and [4];
2) The automatic generation of executable production rules by applying our fast context-free engine-independent grammar over the set of those SBVR rules;
3) Firing off the executable production rules into a specific rule engine.

The rest of the paper is structured as follows: in section II we introduce the techniques for expressing business rules in a format based on natural language, i.e. SBVR; in section III we present our mapping approach for translating SBVR rules into production rules that can be executed by a specific rule engine (i.e. JBoss Drools). In section IV we describe a case study related to MyOpenGov research project [5], where rules describing a health process, expressed in natural language format, constitute the input for the automatic generation of production rules. Finally, in section V we report conclusions and propose possible directions to be followed in future works.

II. NATURAL LANGUAGE EXPRESSIONS FOR DEFINING BUSINESS RULES

SBVR is a specification of the Object Management Group (OMG) useful to formalize complex business rules and business vocabularies. The objective of SBVR is to capture specifications in a formal language similar to the natural one, but, as confirmed in [2] and [6], the SBVR specification is not intended to describe production rules that can be processed by rule engines. SBVR captures declarative rules expressed from a business perspective. Transition from the fact models to mechanized rules in an automated system is outside the scope of SBVR. However, starting from SBVR rules, it is possible to define software artifacts that can be executed by a rule engine. To achieve this goal, a mapping methodology for translating SBVR rules into production rules is necessary. Literature

2 JBoss Drools, http://www.jboss.org/drools
3 In this paper we assume SBVR Structured English as the language with a mapping to SBVR structures of meaning.
reports few approaches to automatically transform SBVR rules into executable production rules. In [7] authors use SBVR to attach business rules descriptions in a business process expressed in Business Process Modeling Notation (BPMN). Nevertheless, no techniques to make executable SBVR business rules are reported. In [8] authors use SBVR as mathematical meta-model for translating business rules into executable models, such as UML/ocl constrains and SQL statements, but no production rules are generated. In [9] a technique to transform SBVR business rules into executable production rules is proposed. Although the work is a good step to reduce the gap between business/domain statements and IT artifacts, it requires manual steps performed by IT staff. In [10] a user friendly Domain Specific Language (DSL) is defined in order to map SBVR rule formats into a format ready to be translated in production rules. However, the approach proposed by authors lacks of an automatic generation of those production rules that can be executed by a rule engine.

III. MAPPING METHODOLOGY FOR TRANSLATING SBVR EXPRESSIONS INTO PRODUCTION RULES

We propose a mapping technique for automatically translating rules expressed in SBVR in production rules that can be processed by a rule engine. Our mapping technique mainly focuses on two steps:

• The mapping of SBVR vocabulary concepts into Java beans;
• The translation of SBVR rule set into JBoss Drools production rules.

A. Mapping SBVR vocabulary into Java beans

The SBVR standard is defined as a "fact-oriented" language [11]. A SBVR vocabulary is configured as a set of descriptions that explain and define the concepts that shape the domain of interest. The SBVR vocabulary includes:

• Noun concepts that define general concepts (terms) and individual concepts typically used to represent proper nouns (names);
• Verb-concepts that describe the structural, behavioral or relational aspects of the business itself, as the connection of two or more noun concepts.

In our methodology the SBVR vocabulary is mapped into a library of Java beans as shown in Fig. 1.

In particular:

• Each complex term (see “man” term in Fig. 1) is mapped as a simple Java class without attributes; the value of the “General Concept” property establishes the inheritance relationship between two generated Java classes;
• Each verb concept is mapped as a definition of an attribute for a Java class. In Fig. 1 the subject of the verb concept is “man” while the object is "age”, so Age Java class became an attribute of the Man Java class.

In our methodology, names, as instantiations of terms, are modeled as instantiated Java beans, which will be injected in the working memory of the target rule engine. Business rules, based on verb-concepts, will fire against existing Java beans (names) in the working memory, according to the inheritance relationship between terms and names.

B. Translation of SBVR rule set into production JBoss Drools production rules

SBVR supports first-order logic with equality, restricted higher-order logic, limited deontic and alethic modal logic, set theory and mathematical logic. SBVR rules can be of two types: structural or operational [11]. A structural rule makes use of an alethic modal operator such as "it is necessary that" or "it is possible that". An operational rule follows a deontic modal operator such as "it is obligatory that", "it is permitted that", "it is forbidden that" or "it is impossible that". SBVR rule syntax is represented in Fig. 2.

![Fig. 2. Syntactic diagram of an SBVR rule](image)

A rule is formed by an optional prefix, consisting of an alethic or deontic operator, followed by an "if-then" expression (e.g. “if a person is adult then the person is 18 years old at least”) or a series of expressions in logical and/or between them ("or_exp"). Each basic expression, represented by the "exp" element (shown in Fig. 3), is based on the existence of a "verb-concept wording" in the vocabulary (e.g. “an enterprise has more than 500 employed employees”) that links the subject term "subject" (e.g. enterprise) to the object term "object" (e.g. employed employees) through the verb-symbol "verb" (e.g. has).

![Fig. 3. Syntactic diagram of a basic SBVR expression](image)

The business constraint is expressed through the use of appropriate logical and quantification operators, in order to make explicit the relationship between subject and object of the verb-concept, declared in the vocabulary and used here. Furthermore SBVR allows a term to be followed by a

---

Fig. 1. SBVR vocabulary to Java bean mapping
subordinate clause, in order to add additional constraints to the referring term. The standard allows to apply such kind of formulas to both subject and object of an expression (e.g. “a man, who has more than 100 cars or has more than one yacht, is rich”). Thus, elements (subjects and objects) can be syntactically modeled as shown in Fig. 4 and Fig. 5.

![Fig. 4. Syntactic diagram of subject and object elements](image)

As stated in [11], not all business rules can be translated into executable production rules. For instance, operational business rules including modal operators like "it is possible that", "it is probably that", "it is permitted that", and "it is non-necessary that" cannot be translated into artifacts executable by rule engines, since they express only a possibility, or the probability, or no-need that some fact happens or not. Given their lack of determinism, in fact, a rule engine cannot determine which actions should be carried out. On the contrary, there are some operators, such as "it is impossible that", "it is not permitted that", "it is prohibited that", "it is forbidden that" that can be directly translated, describing violations within the domain of interest and leaving no doubt on their interpretation. Furthermore, in a rule it is possible to use a modal operator such "it is necessary that" and "it is obligatory that" in order to express a set of actions to be performed when certain conditions are satisfied or that some requirements must always be checked.

SBVR allows if-then expressions (see Fig. 2). These formulations, although syntactically and semantically correct, are not often suitable for being executed into a rule engine. Namely, using "it is necessary that" prefix, the set of expressions that follow the keyword "then" represent the actions to take when the rule is activated. Since these actions are alternatives to each other, even if the rule has a perfect meaning from a business point of view, a rule engine cannot determine which actions should be performed between those that constitute the "or_exp" of the "then" branch. This implies a specific limitation. If we use the "it is impossible that" prefix, the set of expressions that follow the keyword "then" are facts that cannot occur simultaneously with those declared in the "if" statement. Therefore these events should be managed through the production of alert messages and/or deleted from the working memory of the rule engine.

Typically, a generic production rule consists of a Left Hand Side (LHS), containing the rule’s firing conditions, and a Right Hand Side (RHS), indicating the actions to be done after rule has fired. For computational cost reasons, conditional comparisons are not executed in RHS. Therefore, in case of if-then formulation with "it is impossible that" prefix, violations (that, in logical or, make up the then branch of the rule) must be properly broken down and evaluated in the LHS, so every rule engine can check them out efficiently. This aspect implies a specific translation mapping.

C. A grammar for semantic parsing of SBVR rules

In this section our grammar to reduce4, process, and translate a SBVR Rule Set into production rules is described. In figures included in this section, the elements in blue capital letters are known as tokens or terminals. Non-terminal symbols are represented in orange lower case letters.

![Fig. 6. Syntactic diagram of non-terminals rule, or_cond_exp, nec_or_cond_exp, and_cond_exp](image)

The single SBVR Rule, reduced by a non-terminal <rule> (Fig. 6) consists of two parts. The first part can be expressed in two different ways: a) as a requirement (“it is necessary that” or “it is obligatory that”), b) as a prohibition (e.g. “it is forbidden that”). The second part can be expressed in two ways:

1) "if-then" format: the rule is composed by an if branch with conditional statements combined in logical and/or (i.e. non-terminals <or_cond_exp>, <and_cond_exp>), followed by a then branch, composed by statements only in logical and (i.e. <and_action_exp>, <action_exp>)

2) Declarative format: the rule is represented by a chain of propositions in logical and/or (i.e. <nec_or_cond_exp>, <nec_and_cond_exp> for obligation or necessity, <or_cond_exp> and <and_cond_exp> for prohibitions).

Non-terminals included into non-terminal <rule> are shown in Fig. 6. Non-terminal <cond_exp> (Fig. 7) reduces sentences in the following forms: a) "There is a term[t]", b) "A term[t] <verb> <cond_object>". The former verifies the existence of an instance of term within the rule engine working memory; the use of identifier “t” for terms allows business

---

4 In this paper we use the specific terminology of LR Parsers
expressions. From the specific target language, the grammar is able to reach a normalized executable in and actions that can be mapped to subject "term[t]" in logical valorizations over a action_object Fig. 8. Through non-terminal <and_action_exp> (Fig. 8) it is possible to express one or more affirmative statements, reduced by the non-terminal <action_exp>. Statements that can be reduced through <action_exp> are:

1) "A term[t] <verb> <action_object>": it expresses a valorization action on a property of "term[t]" (the valorization action, which may be numeric, literal or boolean, is reduced by non-terminal <action_object>, shown in Fig. 8);
2) "There is a term[t]": it describes an action of creating and inserting a new instance of a term (Java class) into the working memory of the rule engine;
3) "It will not be anymore a term[t]": it implies to remove the instance (Java bean) "term[t]" from the working memory of the rule engine.

The non-terminal <action_orThatExpr> is adopted to perform valorizations over a set of instances-properties related to subject "term[t]" in logical and (the whole description has been omitted for brevity).

The defined grammar allows us to capture all conditions and actions that can be mapped into skeletons of artifacts executable in specific production rules. In this sense, our grammar is able to reach a normalized rule format independent from the specific target language.

D. Generating Drools rules

Jboss Drools focuses on a representation of the knowledge through declarative, concise, unambiguous, first-order logic expressions. Each expression is formed by a "Left Hand Side" (LHS), which contains activation conditions of the rule, and by a "Right Hand Side" (RHS), which contains all actions to perform at the firing of the rule. When some facts are asserted or modified in the working memory, the rule engine searches matches of facts among all the available production rules by evaluating their LHS, according to the pattern matching algorithm ReteOO [12]. When there are some positive matches, the engine proceeds with the activation of the verified rules by executing actions contained in the RHS. The actions are executed over matched Java objects (facts) included into the working memory. Drools also includes some keywords to operate into the working memory such as a) "insert" to add a new fact (i.e. Java object) into the working memory; b) "retract" to delete a fact from working memory; c) "update" to update the status of a matched object in working memory.

Starting from the normalized format of parsed, as described in section C, the mapping of SBVR business rules into production rules expressed in Drools code is shown from Fig. 9 to Fig. 12. The detailed description of the Drools code generation starting from a SBVR rule expressed in the form shown in Fig. 10 is reported in section IV.

The non-terminals <cond_exp> and <nee-cond_exp> make use of <cond_object> (here not described for brevity).

Fig. 7. Syntactic diagram of non-terminals cond_exp and cond_object.

Fig. 8. Syntactic diagram of non-terminals and action_exp, action_exp and action_object.

Fig. 9. Mapping of a SBVR rule into Drools (first form)

Fig. 10. Mapping of a SBVR rule into Drools (second form)

Fig. 11. Mapping of a SBVR rule into Drools (third form)

Fig. 12. Mapping of a SBVR rule into Drools (forth form)
IV. INFERRING BUSINESS RULES OF A HEALTH PROCESS: A CASE STUDY

In order to validate our approach, a real eHealth scenario is here reported. It has been defined within MyOpenGov research project [5], which involves several Health Authorities in Italy. MyOpenGov project aims at developing a social collaboration platform, based on Open source and Open service innovation approaches, which is able to integrate services, by means of the bus (called Service Innovation Decoder) capable to connect different data coming from legacy systems and web services provided by Public Administrations. Our approach has been implemented into a parser, which is able to generate both Java beans from the SBVR vocabulary and JBoss Drools code starting from the SBVR business rules.

In Fig. 13 the health process includes business rules written by a non IT skilled person in natural language by means of SBVR standard and the proposed grammar. The same person has also modeled the business process by using the Business Process Modeling Notation (BPMN) standard. In particular, the process in Fig. 13 helps the citizen to book a medical exam, reported in a doctor’s prescription, identifying the Health Authority in Campania Region that could supply it in the fastest way. Several SBVR rules are associated to some tasks in order to associate decisions criteria.

In particular:

1) In “Process citizen prescription” activity, performed by MyOpenGov bus, the associated SBVR rule verifies if the medical prescription is valid (i.e. it is not expired):

   **It is necessary that a prescription is not expired**

2) In “Get availability” activity, performed by a Campania Health Authority (CHA), three SBVR rules are associated in order to:

   a) determine the possible dates on which a machinery for Computerized Tomography (CT) is available. The available date list contains the available dates of all machineries for Computerized Tomography;

   If a machinery has equals ‘CT’ as name and has an availability [CtDateList] and there is a available date list [AvDateList], then put [CtDateList] in [AvDateList]

   b) access the priority waiting list if the citizen has made a booking request in “private regime”;

   If there is a priorityWaitingList [pwl] and a bookingRequest [br] has ‘private’ regime [pr] then put [br] in [pwl]

   c) access the normal waiting list if the booking request of citizen has been made in “public regime”. The “Get availability” activity is processed for each Health Authority in Campania Region (Italy) near the residence of the requesting citizen.

   If there is a normalWaitingList [nwl] and a bookingRequest [br] has ‘public’ regime [pr] then put [br] in [nwl]
included into the SBVR vocabulary. Then, the parser finds the term and the corresponding "Prescription" Java class.

4) "is not" is interpreted as a verb in the negative form;
5) "expired" is parsed as <cond_object>, so the translator semantically checks if the term "expired" is included into the SBVR vocabulary. The parser finds the term and realizes that the corresponding Java class type is "Boolean".

6) As the <nec_cond_exp> has been recognized, the translator checks if the verb-concept "prescription is expired" is included into the SBVR vocabulary; a semantic rule associated to such a non-terminal lets the translator generate the corresponding DRL condition fragment, as a part of the LHS, related to the Java bean "Prescription".

7) As the <nec_cond_exp> is the last part of the rule, it is also reduced as both <nec_and_cond_exp> and <nec_or_cond_exp>; this way, the translator realizes that no other condition fragment has to be generated over the "prescription" term.

8) The modal terminal "IS NECESSARY" followed by the non-terminal <nec_or_cond_exp> is parsed as a <rule>, so both the RHS and LHS of the production rule expressed in Drools can be generated. In particular, the generated LHS is the following: "$Prescription : Prescription( expired!=false )". As we parsed a barring rule, if such a condition is matched for some facts, it would be a violation, so a "retract" action over the "Prescription", included into the working memory, is generated into the RHS. The complete generated DRL rule is shown in Fig. 15.

Through our mapping technique and the grammar described in section III, our parser generated the drools code ready to be processed by a Drools engine, as shown in Fig. 15.

![Fig. 15. Production rules in Drools language generated by the translator](image-url)