Formal Rule Representation and Verification from Natural Language Requirements Using an Ontology

Driss Sadoun, Catherine Dubois, Yacine Ghamri-Doudane, Brigitte Grau

Formal Rule Representation and Verification from Natural Language Requirements Using an Ontology

D. Sadoun[1,2], C. Dubois[3,4], Y. Ghamri-Doudane[5], and B. Grau[1,3]

[1] LIMSI/CNRS, France
[2] University Paris-Sud, France
[3] ENSIIE, France
[4] CEDRIC/CNAM, France
[5] University of La Rochelle/L3i Lab, France

Abstract

The development of a system is usually based on shared and accepted requirements. Hence, to be largely understood by the stakeholders, requirements are often written in natural language (NL). However, checking requirements completeness and consistency requires having them in a formal form. In this article, we focus on user requirements describing a system behaviour, i.e. its behavioural rules. We show how to transform behavioural rules identified from NL requirements and represented within an OWL ontology into the formal specification language Maude. The OWL ontology represents the generic behaviour of a system and allow us to bridge the gap between informal and formal languages and to automate the transformation of NL rules into a Maude specification.

Keywords

Knowledge representation, OWL ontology, NL requirements, formal verification;

1 Introduction

Requirements correspond to a specification of what should be implemented. Among other, they describe how a system should behave. Stakeholders of a system development often use natural language (NL) for a broader understanding, which may lead to various interpretations, as NL texts can contain semantic ambiguities or implicit information and be incoherent. Thus, requirements have to be checked and this requires them to be represented in a formal language. A transformation of NL requirements into formal specifications is usually costly in human and material resources and would benefit of an automatic method. A direct transformation is difficult, if not impossible [5], which leads to the need of an intermediate representation to reduce the gap between the two formalisms. Both works of [5] and [9] propose a first step in the formalization process by transforming NL specifications into SBVR. Similarly, in [7], the authors use SBVR as

an intermediate representation to transform NL business rules into semi-formal models such as UML. The tool NL2Alloy [1] also uses SBVR as a pivot representation to generate Alloy[6] code from NL constraints. To our knowledge, only NL2Alloy proposes a complete chain of transformation from NL to formal specifications, but it does not perform formal verifications on the intermediate representations to validate it. Indeed, verifying extracted information needs formal knowledge representation and inference mechanisms. However, controlled natural languages as SBVR or semi-formal representation models as UML often lack validation mechanisms and inference engines. These shortcomings have led many researchers to explore the transformation of SBVR or UML into languages such as OWL and SWRL [6, 10] or as Maude [3].

We propose an OWL-DL ontology based on description logics as an intermediate representation. We use this ontology to guide the automatic identification of behavioural rules from NL requirements analysis and to represent them formally [8]. Behavioural rules are represented in the ontology in order to be transformed into a formal specification language. Indeed, OWL allows us to check the consistency and the completeness of the modelled rules. However, it cannot represent state evolution or sequential rules application. Hence, to simulate and validate the whole system behaviour, we propose to transform the ontology model into a formal specification Maude. In this article, we focus on the ontology conception choices and the transformation process that enable us to automate the production of formal specifications and to maintain the link between NL requirements and their formal representation.

This work has been done in the framework of the project *ENVIE VERTE*[7] which aims to allow a user to configure her own smart space by describing her requirements in natural language. A smart space is a set of communicating objects (sensors, actuators and control processes) that may influence, under well defined conditions, the behaviour of the smart space devices (physical processes). The behavioural rules determine desired component interactions.

2 Ontology of a system behaviour

2.1 Conceptualisation choices

An ontology defines concepts ($\mathbb{C}$), properties ($\mathbb{P}$) and individuals ($\mathbb{I}$) of a domain. Concepts and properties of an ontology are defined by terminological axioms ($\mathbb{A}$). We represent an ontology $\mathbb{O}$ as a tuple $< \mathbb{C}, \mathbb{P}, \mathbb{A}, \mathbb{I}, \mathfrak{I}^{\mathbb{C}}, \mathfrak{I}^{\mathbb{P}} >$ where:
- $\mathbb{C}$ is a set of concepts;
- $\mathbb{P}$ is a set of binary properties;
- $\mathbb{A}$ is a set of terminological axioms;
- $\mathbb{I}$ is a set of individuals;
- $\mathfrak{I}^{\mathbb{C}}$ is a function that associates to each concept a set of individuals;

[6] A language and tool for relational model verification. http://alloy.mit.edu/alloy/
[7] Funded by *DIGITEO*, projet DIM LSC 2010.

- $\mathfrak{I}^{\mathbb{P}}$ is a function that associates to each property a set of couples of individuals or of couples individual/value.

The ontology of a system behaviour has to define the components of the system, their characteristics and the way they behave. In this framework, it is important to highlight a distinction between two kinds of individuals within ontologies: 1) individuals representing entities; 2) individuals representing a type characterizing entities, which lead us to distinguish two sorts of concepts: *individual concepts* and *generic concepts*. This distinction is pertinent for both NL requirement analysis and the automatic ontology translation into the formal language Maude. Based on that, we define two high level concepts to represent a system behaviour: *Component* ($\mathbb{C}_C \subseteq \mathbb{C}$) and *Type* ($\mathbb{C}_T \subseteq \mathbb{C}$) (cf. figure 1).

1. each sub-concept of *Component* is an *individual concept* defining sets of individuals representing entities of the domain (physical components, software components, phenomena, ...);
2. each sub-concept of *Type* is a *generic concept* defining specific types (color, model, brand, ...) of the domain. It extends predefined data types (integer, real, boolean, string, ...), used to characterize the components of the system.

Representing the system behaviour requires taking into account the dynamic aspects of its operation. Thus, we modelled two super-properties in the ontology: 1) *Relation* for describing an interaction between two components of the system; 2) *Attribute* for describing a characteristic of a component, defined as follows:

1. sub-properties of *Relation* are defined exclusively between two sub-concepts of *Component*. Within OWL, each property is defined as an *ObjectProperty*. Formally $\mathbb{P}_R$ is the set of properties P of type *Relation* such that $D \triangleleft P \triangleright R$[8] with $D \subseteq \mathbb{C}_C$ and $R \subseteq \mathbb{C}_C$ et $\mathfrak{I}^{\mathbb{P}}(P) \subseteq \mathfrak{I}^{\mathbb{C}}[\mathbb{C}_C] \times \mathfrak{I}^{\mathbb{C}}[\mathbb{C}_C]$[9].
2. sub-properties of *Attribute* are defined between a sub-concept of *Component* or *Type* and an OWL type. Within OWL, each property is defined as *ObjectProperty* between sub-concepts of *Component* and sub-concepts of *Component* or *Type*, or as a *DataProperty* between *Component* or *Type* and an OWL type. Formally $\mathbb{P}_{\mathbb{A}}$ is the set of properties P of type *Attribute* such that $D \triangleleft P \triangleright R$[8] with $D \subseteq \mathbb{C}_C \cup \mathbb{C}_T$ and $R \subseteq \mathbb{C}_C \cup \mathbb{C}_T \cup \mathbb{T}$ and $\mathfrak{I}^{\mathbb{P}}(P) \subseteq (\mathfrak{I}^{\mathbb{C}}[\mathbb{C}_C] \cup \mathfrak{I}^{\mathbb{C}}[\mathbb{C}_T]) \times (\mathfrak{I}^{\mathbb{C}}[\mathbb{C}_C] \cup \mathfrak{I}^{\mathbb{C}}[\mathbb{C}_T] \cup \mathbb{V})$. We also distinguish two types of attributes:
   - *dynamic attribute* whose value may evolve over the time, as the balance of a bank account;
   - *static attribute* whose value is not set to change, such as a bank account ID. This last kind of attribute corresponds to definitional properties of a concept that can be used to identify and distinguish its individuals.

The result of our conceptualisation choices is the ontology illustrated in Figure 1. The ontology is divided in two parts: the upper level ontology models a

[8] We note $D \triangleleft P \triangleright R$ to define for each property P its *domain* D and its *range* R.

[9] $\mathfrak{I}^{\mathbb{C}}[\mathbb{C}_C]$ represents the ranges of all the elements of $\mathbb{C}_C$ by $\mathfrak{I}^{\mathbb{C}}$ ($\mathfrak{I}^{\mathbb{C}}[\mathbb{C}_C] = \bigcup_{c \in \mathbb{C}} \mathfrak{I}^{\mathbb{C}}(C)$).

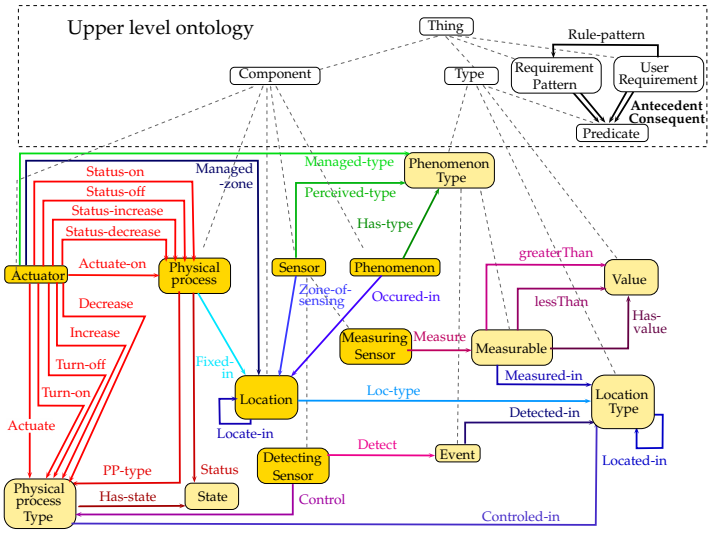

Fig. 1. The ontology of smart space behaviour

generic system behaviour and the domain specific ontology models a smart space behaviour. This specific part contains fourteen concepts : seven sub-concepts of *Component*, and seven sub-concepts of *Type*. The properties are represented by oriented arrows linking concepts of their domain and range. We only figure properties corresponding to *ObjectProperty*, they are thirty one. Dotted Arrows represents subsumption relations.

2.2 Behavioural rules

As concepts and properties, *Behavioural rules* participate to the domain definition, by modelling its dynamic aspects. They are formed as *antecedent* $\rightarrow$ *consequent*. The *antecedent* defines conditions under which the rule applies. The *consequent* defines the result of its application. Each of them corresponds to a conjunction of predicates denoting instances of a property $P(i_x, i_y)$ with $(i_x, i_y) \in \mathfrak{I}^{\mathbb{P}}(P)$, since, in our approach, rule identification is guided by property instance identification [8]. Within the ontology, we model a behavioural rule as two sets of predicates $P_k(i_x, i_y)$ with P_k a binary predicate referring to a property instance and i_x an individual, a literal (value of a basic data type) or a variable. We defined a concept *Predicate* as a sub-concept of the concept *Type* (cf. figure 1), associated to the two properties *Antecedent* & *Consquent* (cf. figure 1) on which the behavioural rules are constituted.

We distinguish two types of behavioural rules: 1) rules describing the general behaviour of the system that is independent of the user needs; 2) rules specific to the user requirements. We propose to model within the ontology the two

concepts *Requirement-Pattern* and *User-Requirement*. *Requirement-Pattern* is a set of different generic patterns of rules. Its individuals are defined by an expert of the domain to guide the NL requirement analysis. *User-Requirement* is a set of behavioural rules specified by a user. Its individuals are created automatically from NL requirements analysis and linked to their model pattern by the property *Rule-pattern* (cf. figure 1). Within the ontology five requirement patterns have been defined for guiding the identification of behavioural rules of a smart space.

2.3 Population of the ontology

In [8], we proposed an approach for ontology population based on the identification of property instances in sentences which leads to recognize triples of individuals. Instance property recognition enables to resolve some ambiguities and to infer implicit individuals. The creation of *User-Requirement* individuals exploits these property instances and depends on two verifications based on the use of OWL reasoning and SQWRL queries. First, for each requirement pattern represented in the ontology, we check that all the predicates (i.e. property instances) specializing it have been recognized and do not introduce any inconsistency in the ontology, then, that the resulting rule, i.e. the individual of *User-Requirement* is correctly formed. If this two verifications hold, an instance of the concept *User-Requirement* is created. During the ontology population process, several instances of *User-Requirement* can be associated to an instance of *Requirement-Pattern* via the property Rule-Pattern (cf. Figure 1). Each of them is associated with the sentence number it is extracted from. It enables to keep the link between textual requirements and formal rules.

We collected user requirements of a smart space behaviour configuration via a platform available on the web[10]. We collected about hundred sentences[11] (2171 words). Figure 2 presents an example of an individual of *User-Requirement* that specializes an instance of *Requirement-Pattern*. It was created automatically from the NL requirement analysis and was identified from the sentence number 1 "*When I enter a room the door opens automatically.*" of the analysed user requirements. Right elements in bold are instances identified from user requirements analysis. Elements preceded by a question mark '?' correspond to variables. The left property in bold is a super-property[12] that determines the type of property to identify from user requirements analysis.

Within the hundred sentences, 62 were manually annotated as containing a behavioural rule. From user requirements analysis, a total of 28 rules were completely identified and created in the ontology and 34 rules were partially recognized. During the ontology reasoning, two rules among the 28 were rejected, being inconsistent with two existing rules and 3 were identified as containing an additional (incorrect) predicate. As within the ontology, identified individuals are linked to the sentence they were extracted, a precise feedback is returned to the

[10] http://perso.limsi.fr/sadoun/Application/en/SmartHome.php
[11] A rule is extracted from a sentence
[12] *Actuate* is the super-property of *Turn-on*.

An individual of *Requirement-Pattern* (a generic rule).	**R-1 : When I enter a room the door opens automatically**.
Detected-in(**t**,**l**)	*Detected-in*(**movement-in**,**room**)
Controlled-in(**p**,**l**)	*Controlled-in*(**door**,**room**)
Has-type(?ph,**t**)	*Has-type*(?s,**movement-in**)
Occurred-in(?ph,?loc)	*Occurred-in*(?s,?I1-445)
Perceived-type(?s,**t**)	*Perceived-type*(?I1-280,**movement-in**)
Zone-of-sensing(?s,?loc)	*Zone-of-sensing*(?I1-280,?I1-445)
Managed-type(?a,**t**)	*Managed-type*(?I1-8,**movement-in**)
Managed-zone(?a,?loc)	*Managed-zone*(?I1-8,?I1-445)
Loc-type(?loc,**l**)	*Loc-type*(?I1-445,**room**)
⇒ **Actuate**(?a,**p**)	⇒ **Turn-on**(?I1-8,**door**)

Fig. 2. A user requirement created from NL requirement analysis

user, highlighting missing and incorrect information in order to let her correct or complete the concerned requirement. Once all the necessary checks have been performed successfully, the validated rules are transformed into Maude.

3 From the ontology to the Maude formal specifications

3.1 The formal specification language Maude

Maude[13] enables to describe the dynamic of a system, i.e. its state changes, and provides different tools for checking it. The state space of a system is represented by a signature Σ that defines sorts (i.e. types) of constants and variables manipulated by Maude and operators that will act upon the manipulated data and by a set of equations $\mathcal{E}$ built between terms using the signature. Within Maude, the evolution of the system state is described by *rewriting rules* of the form $R : t \rightarrow t^{'}$, where t and $t^{'}$ are terms formed on the signature. Rewriting rules rewrite each term of the left hand side of the rule into a term of the right hand side. The rewriting mechanism allows for specification animation and verification of certain properties as the reachability or the non-reachability of particular states.

Maude defines an *object-oriented module* that offers an object-oriented syntax which is well adapted for concurrent systems, using sets of objects, and a communication mechanism based on message transmission between objects. We use it as a target module for the transformation of the ontology model.

In an object-oriented module, objects are of the form $<O : C|a_1 : v_1, ..., a_n : v_n>$ with O the object identifier, C the object class, a_i $(i \in 1..n)$ its attribute names and v_i $(i \in 1..n)$ the corresponding attribute values. Messages represent the dynamic interaction between objects. They have the form $msg\ Mes : Oid, T_1, ..., T_k \rightarrow Msg$. with msg a keyword, Mes the message name, Oid the type of the recipient object and T_i $(i \in 1..k)$ the types of the message arguments. The state of a system, called configuration, corresponds to a multiset of objects and messages. It is defined using a Maude equation of the form: $eq\ Conf = Ob_1\ ...\ Ob_m\ Mes_1\ ...\ Mes_n$. with eq a keyword, $Conf$ the configuration name, Ob_i and Mes_i the objects and messages of the state system.

We represent a Maude object oriented model as a tuple $<\mathcal{C},\mathcal{M},\Sigma,\mathcal{E},\mathcal{R}>$ with:

[13] http://maude.cs.uiuc.edu/

– $\mathcal{C}$ is the set of class names with, for each class, its set of pairs (attribute, type);
– $\mathcal{M}$ denotes the set of message names;
– Σ corresponds to the typing environment. Each element (constant or variable) is associated to its type;
– $\mathcal{E}$ corresponds to the set of equations representing the state of the system (its configuration) with $\mathcal{E} = \mathcal{E}_{\mathcal{O}} \cup \mathcal{E}_{\mathcal{M}}$ such that:
  - $\mathcal{E}_{\mathcal{O}}$: the set of configurations-objects pairs;
  - $\mathcal{E}_{\mathcal{M}}$: the set of configurations-messages pairs.
  - $\mathcal{R}$ contains the rewriting rules.

3.2 Transformation approach

In this section, we propose a mapping between the ontological elements and the object-oriented Maude elements for an automatic translation. Ontological elements to translate are those contributing to the representation of the system state evolution. They correspond to *User-Requirement* instances and the elements necessary for their definition: concepts *Component* and *Type*, properties (attributes and relations), individuals and their property values. Figure 3 illustrates this mapping. The set of relations $\mathbb{P}_R$ is represented in Maude by a set of messages $\mathcal{M}$ between two objects as they represent evolving relations. The set of attributes $\mathbb{P}_A$ is translated as object attributes. Finally, instances of *User-requirement* are translated as rewriting rules with an antecedent and a consequent built on objects, messages, attributes, literals (i.e. values of basic types) and variables.

The dynamic evolution of a rewriting rule depends on messages and dynamic attributes (cf. section 2.1). When a rule applies, messages of the antecedent are not rewritten and some new messages may appear in the consequent, also *dynamic attributes* values may change and new attributes may appear in the consequent as in Figure 4, which illustrates a *rewriting rule* created from the user requirement *R-1* (cf. Figure 2) and extracted from the sentence number 1 "*When I enter a room the door opens automatically.*" the dynamic attribute *Turn-on* of the object *Actuator* is created in the consequent part.

OWL Ontology	object oriented model Maude
Individual of the concept *Component* ($\in I_C$)	Object ($\in \mathcal{E}$)
Individual of the concept *Type* ($\in I_T$)	Attribute value ($\in \mathcal{E}$)
Sub-concept of the concept *Component* ($\in C_C$)	Class ($\in \mathcal{C}$)
Sub-concept of the concept *Type* ($\in C_T$)	Sort *Oid* ($\in \Sigma$)
Relation ($\in \mathbb{P}_R$)	Message ($\in \mathcal{M}$)
Attribute (static & dynamic) ($\in \mathbb{P}_A$)	Attribute ($\in \Sigma$)
Instance of *User-Requirement* ($\in I_{RU}$)	Rewriting rule ($\in \mathcal{R}$)

Fig. 3. Correspondence between our ontology model and Maude model

rl [R-1] : < **door** : **Physical-process-Type** | Controlled-in : **room** >
< I1-445-8 : **Location** | Loc-type : **room** >
< I1-326-6 : **Phenomenon** | Has-type : movement-in, Occurred-in : I1-445-8 >
< I1-280-7 : **Sensor** | Perceived-type : movement-in, Zone-of-sensing : I1-445-8 >
< **room** : **Location-Type** | >
< smoke : **Event** | Detected-in : **room** >
< I1-8-2 : **Actuator** | Managed-type : movement-in, Managed-zone : I1-445-8 >
⟶
< **door** : **Physical-process-Type** | Controlled-in : **room** >
< I1-445-8 : **Location** | Loc-type : **room** >
< I1-326-6 : **Phenomenon** | Has-type : movement-in, Occurred-in : I1-445-8 >
< I1-280-7 : **Sensor** | Perceived-type : movement-in, Zone-of-sensing : I1-445-8 >
< room : **Location-Type** | >
< smoke : **Event** | Detected-in : **room** >
< I1-8-2 : **Actuator** | Managed-type : movement-in, Managed-zone : I1-445-8, **Turn-on** : **door** > .

Fig. 4. A Maude rewriting rule translated from the behavioural rule R-1

3.3 Automatic translation of the ontology into Maude specifications

Following the mapping of Figure 3, we implemented the translation function $Trad_O$ which exploits *getter-functions* (prefixed by *get-*) issued from the *Java APIs OWL* and *Jess* or implemented by us to query the ontological elements. $Trad_O$ takes the ontology model $(\mathbb{C}, \mathbb{P}, \mathbb{A}, \mathbb{I}, \mathfrak{I}^{\mathbb{C}}, \mathfrak{I}^{\mathbb{P}})$ as input and calls four translation functions (cf. Algorithm $Trad_O$): $Trad_C, Trad_M, Trad_E$ and $Trad_R$. Each of these functions takes as input a subset of the ontology model and translates it into a sub-set of the Maude model. In order to generate Maude specifications from the resulting Maude model, we implemented *pretty-printing* functions (prefixed by *pp-*) that generate portions of Maude code. Their application results in the creation of a Maude specification file. The main function *pp-generation-of-code-Maude* takes as input the output result of $Trad_O$ ($<\mathcal{C},\mathcal{M},\Sigma,\mathcal{E},\mathcal{R}>$) and produces a Maude specification file. It calls eight *pretty-printing* functions (cf. Algorithm *pp-generation-of-code-Maude*) that writes each a sub-set of Maude specifications. The operator $\twoheadleftarrow$ denotes the automatic Maude code generation into the specification document *Spec-Maude*.

Input: $\mathbb{C}, \mathbb{P}, \mathbb{A}, \mathbb{I}, \mathfrak{I}^{\mathbb{C}}, \mathfrak{I}^{\mathbb{P}}$;
Output: $< \mathcal{C},\mathcal{M},\Sigma,\mathcal{E},\mathcal{R} >$;
$\mathbb{C}_{\mathbb{C}} \leftarrow$ *get-ConceptSubClasses*$(\mathbb{A}, Composant)$;
$\mathbb{C}_{\mathbb{T}} \leftarrow$ *get-ConceptSubClasses*$(\mathbb{A}, Type)$;
$\mathbb{C}_{\mathbb{RU}} \leftarrow$ *get-SubConcepts*$(\mathbb{A}, User\text{-}requirement)$;
$\mathbb{I}_{RU} \leftarrow$ *get-ConceptIndividuals*$(\mathbb{C}_{RU}, \mathfrak{I}^{\mathbb{C}})$;
$\mathbb{P}_{\mathbb{R}} \leftarrow$ *get-OntologyRelations*$(\mathbb{A})$;
$\mathcal{C} \leftarrow Trad_C(\mathbb{C}_C, \mathbb{C}_{\mathbb{T}}, \mathbb{A})$;
$\mathcal{M} \leftarrow Trad_M(\mathbb{P}_R)$;
$<\mathcal{E},\Sigma_0> \leftarrow Trad_E(\mathbb{C}_C, \mathbb{P}_R, \mathbb{A}, \mathbb{I}, \mathfrak{I}^{\mathbb{C}}, \mathfrak{I}^{\mathbb{P}})$;
$<\mathcal{R},\Sigma> \leftarrow Trad_R(\mathbb{I}_{RU}, \mathfrak{I}^{\mathbb{P}}, \mathbb{P}_R, \mathbb{A}, \Sigma_0)$;

Algorithm $Trad_O$

Input: $< \mathcal{C},\mathcal{M},\Sigma,\mathcal{E},\mathcal{R} >$, *Spec-Maude*;
Output: *Spec-Maude*;
Spec-Maude $\twoheadleftarrow$ *pp-declareClass*$(\mathcal{C})$;
Spec-Maude $\twoheadleftarrow$ *pp-declareMessage*$(\mathcal{M})$;
Spec-Maude $\twoheadleftarrow$ *pp-declareObject*(Σ);
Spec-Maude $\twoheadleftarrow$ *pp-declareVariables*(Σ);
Spec-Maude $\twoheadleftarrow$ *pp-declareObjectConfiguration*(Σ);
Spec-Maude $\twoheadleftarrow$ *pp-createObjectConfiguration*$(\mathcal{E})$;
Spec-Maude $\twoheadleftarrow$ *pp-createMsgConfiguration*$(\mathcal{E})$;
Spec-Maude $\twoheadleftarrow$ *pp-createRules*$(\mathcal{R})$;

Algorithm *pp-generation-of-code-Maude*

The algorithm 1 details the function $Trad_R$ (cf. Algorithm $Trad_O$) that translates the *user requirements* ($\mathbb{I}_{RU}$) modelled in the ontology into rewriting rules describing the system behaviour within Maude. These rules are formed by binary predicates representing ontology properties. Each predicate may have as argument individuals, literals or variables. Existing objects have been declared in Σ_0 and created in $\mathcal{E}$ within the function $Trad_E$ (cf. Algorithm $Trad_O$). Variables and

literals still need to be declared. For each predicate of the properties *Antecedent* and *Consequent*, *getter-functions* are called to get its name (the property to which it refers) and its domain and range values. These values are inputs of the function *updateObjects* that creates objects or updates their values if they already exist. For example, during the creation of the rewriting rule R-1 (cf. Figure 4) the object *Actuator* has been created from the predicate *Managed-type*, then updated by the predicate *Managed-zone* and finally updated in the *consequent* of the rule by the predicate *Turn-on* that represents a dynamic attribute.

3.4 User requirements verification in Maude

Maude incorporates a variety of validation and verification tools [2] including a model checker [4]. A model-checker enables the model exploration. From an initial configuration, it explores the possible states of the represented system based on rewriting rules application. The model-checking allows us to check undesirable state reachability as states resulting from the simultaneous application of rules in contradiction i.e. that can be triggered at the same time and contains in their consequents predicates in opposition (as *Turn-on* and *Turn-off*) on the same object. Then we say that the rules are inconsistent. Hence, the rule created from the sentence 88 " *When a sensor detects a hot temperature in any*

Input: $\mathbb{I}_{RU}$,$\mathfrak{I}^{\mathbb{P}}$, $\mathbb{P}_R$, $\mathbb{A}$, Σ_0
Output: $\mathcal{R}$,Σ
$\mathcal{R} \leftarrow \emptyset$; $\Sigma \leftarrow \Sigma_0$; *Objs-Antecedent* $\leftarrow \emptyset$; *Objs-Consequent* $\leftarrow \emptyset$;
Msg-Antecedent $\leftarrow \emptyset$; *Msg-Consequent* $\leftarrow \emptyset$; *Class-Attributes-Values* $\leftarrow \emptyset$;
for *each* i_{RU} *in* $\mathbb{I}_{RU}$ **do**
  A-predicates $\leftarrow$ *get-RangeValue*(i_{RU},*Antecedent*,$\mathfrak{I}^{\mathbb{P}}$) ;
  C-predicates $\leftarrow$ *get-RangeValue*(i_{RU}, *Consequent*,$\mathfrak{I}^{\mathbb{P}}$) ;
  for *each a-predicate in A-predicates* **do** //predicates of the antecedent
  $p \leftarrow$ *get-PredicateName*(*a-predicate*) ;
  $v_D \leftarrow$ *get-PredicateDomainValue*(*a-predicate*) ;
  $v_R \leftarrow$ *get-PredicateRangeValue*(*a-predicate*) ;
  $t_D \leftarrow$ *get-Domain*(p,$\mathbb{A}$); $t_R \leftarrow$ *get-Range*(p,$\mathbb{A}$) ;
  if *isVariableOrLitteral*(v_D) **then**
    $\Sigma \leftarrow \Sigma \cup \{(v_D,t_D)\}$;
  if *isVariableOrLitteral*(v_R) **then**
    $\Sigma \leftarrow \Sigma \cup \{(v_R,t_R)\}$;
  if $p \in P_R$ **then** //p is a relation
  Msg-Antecedent $\leftarrow$ *Msg-Antecedent* $\cup \{(p, v_D, v_R)\}$;
  Objs-Antecedent $\leftarrow$ *updateObjects*(*Objs-Antecedent*,$\{(v_D,t_D,\emptyset,\emptyset)\}$);
  Objs-Antecedent $\leftarrow$ *updateObjects*(*Objs-Antecedent*,$\{(v_R,t_R,\emptyset,\emptyset)\}$);
  else //p is an attribute
  Objs-Antecedent $\leftarrow$ *updateObjects*(*Objs-Antecedent*,$\{(v_D,t_D,p,v_R)\}$);
  Objs-Consequent $\leftarrow$ *Objs-Antecedent*;
  for *each c-predicate in C-predicates* **do** //predicates of the consequent
  $p \leftarrow$ *get-PredicateName*(*c-predicate*) ;
  $v_D \leftarrow$ *get-PredicateDomainValue*(*c-predicate*) ;
  $v_R \leftarrow$ *get-PredicateRangeValue*(*c-predicate*) ;
  $t_D \leftarrow$ *get-Domain*(p,$\mathbb{A}$); $t_R \leftarrow$ *get-Range*(p,$\mathbb{A}$) ;
  if *isDynamic*(p,$\mathbb{A}$) **then** //p is an dynamic attribute
  Objs-Consequent $\leftarrow$ *updateObjects*(*Objs-Consequent*,$\{(v_D,t_D,p,v_R)\}$) **else**
    if $p \in P_R$ **then** //p is a relation
    Msg-Consequent $\leftarrow$ *Msg-Consequent* $\cup \{(p, v_D, v_R)\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{$(*Objs-Antecedent*,*Objs- Consequent*,Msg-Antecedent,Msg- Consequent)$\}$;

Algorithm 1: Type declaration and rewriting rules creation

room combined with smoke in this room, close all the doors and windows." was identified as inconsistent with the rule number 1. Model checking also allows us to check the completeness of the specified system by checking the reachability of desirable states. For example, in the framework of a smart space, it is necessary to check if all physical processes can reach the states *on* and *off* at least once. Thus, a message can be returned to the user. As it was the case for the lack of a rule that *turns off* the physical process *light-bathroom*.

4 Conclusion

We proposed an approach for behavioural rules representation and formalization from user requirements written in natural language. The core of this approach is an OWL-DL ontology that encompasses the general behaviour of a system. The ontology is used as a pivot representation as it defines a framework for guiding the identification of behavioural rules and allows us to implement an automated transformation of them into a formal specification in Maude. We described an application of our approach on the domain of smart spaces and showed how representing the behaviour of smart space by a Maude specification enabled us to check its consistency and completeness.

References

1. Bajwa, I.S., Bordbar, B., Lee, M., Anastasakis, K.: Nl2alloy: A tool to generate alloy from nl constraints. JDIM 10(6) (2012)
2. Clavel, M., Duràn, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.: The maude formal tool environment. In: CALCO, vol. 4624, pp. 173–178 (2007)
3. Durán, F., Gogolla, M., Roldán, M.: Tracing properties of uml and ocl models with maude. AMMSE (2011)
4. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude {LTL} model checker. ENTCS 71, 162 – 187 (2004)
5. Guissé, A., Lévy, F., Nazarenko, A.: From regulatory texts to brms: how to guide the acquisition of business rules? In: RuleML. pp. 77–91 (2012)
6. Karpovic, J., Nemuraite, L., Stankeviciene, M.: Requirements for semantic business vocabularies and rules for transforming them into consistent owl2 ontologies. In: Information and Software Technologies, vol. 319, pp. 420–435 (2012)
7. Njonko, P., El Abed, W.: From natural language business requirements to executable models via sbvr. In: ICSAI (2012)
8. Sadoun, D., Dubois, C., Ghamri-Doudane, Y., Grau, B.: From natural language requirements to formal specification using an ontology. In: ICTAI (2013)
9. Selway, M., Grossmann, G., Mayer, W., Stumptner, M.: Formalising natural language specifications using a cognitive linguistics/configuration based approach. In: EDOC. pp. 59–68 (2013)
10. Sukys, A., Nemuraite, L., Paradauskas, B., Sinkevicius, E.: Transformation framework for sbvr based semantic queries in business information systems. In: BUSTECH. pp. 19–24 (2012)