

A Logic-based Service for Verifying Use Case Models

Fernando Bautista, Carlos Cares

Computer Science and Informatics Department, University of La Frontera (UFRO)
Temuco, Chile

Email: fernandobautis@gmail.com, carlos.cares@ceisufro.cl

Abstract—Use cases are a modeling means to specify the required use of software systems. As part of UML (Unified Modeling Language), it has become the *de facto* standard for functional specifications, mainly in object-oriented design and development. In order to check these models, we propose a theoretical solution by adapting a general quality of models framework (SEQUAL), and, following our approach, a rule-based solution that includes both expert-based and definition-based rules. In order to promote a distributed set of quality assessment services, a Web service has been developed. It works on XMI (XML Metadata Interchange) files which are parsed and verified by Prolog clauses.

Keywords—Rule-based quality; UseCase verification; Logic-based services; XMI; Prolog.

I. INTRODUCTION

In Software Engineering, use cases are a means to specify the required uses of software systems. Typically, they are used to represent what the system is supposed to do. Use cases are part of UML (Unified Modeling Language) specifications and they have become so wide spread that they are now considered the *de facto* standard for requirements specification of object-oriented software systems [1].

Quality assurance of use cases is a common topic in Software Engineering. For example, some heuristics including UML use cases, have been proposed for model revision [2]. Moreover, preparing good use cases for connection with other static and dynamic models remains important as they are a key representation for verification and validation [3].

Other studies have tried to assist in the semi-automatic verification / validation of use cases. Kotb and Katayama [4] present a novel approach to check the verification of the use case diagrams. Shinkawa [5] proposes a formal verification process model for UML use case, and Gruner [6] details a meta model of possible relations between use cases, which may, in the future, be implemented in Prolog. However, these proposals assume that the set of steps in order to implement it (known as basic course or basic flow) is always part of the use case specification. However, this assumption is not broadly true, and, moreover, it is shown that the way of this narrative presents ambiguity [7]. From a formal perspective, an interesting summary is found in [8], where different lapses are identified for the analyzed approaches.

The objective is to show a tool for supporting a quality assessment process of use case diagrams, even, when some of these use cases have no proper narrative inside of them. The theoretical base is given by SEQUAL (SEmiotic QUALity) [9]. It is a highly spread quality framework for models and it addresses different kinds of qualities including syntactic, semantic, pragmatic and social qualities which make it very complex to include all these quality perspectives from the scratch. Under this assumption, a rule-based system is a

modular and scalable solution in order to initially implement some types of verifications and then another group of them under an incremental development.

In this paper, we present a first Prolog prototype, as proof of concept, of a tool that can aid the quality assessment of a use case diagram. Moreover, we have implemented it as a Web service in order to illustrate that logic-based solutions can also be part of key quality assessment process in cloud-based software development environment.

In order to check use case models and their application, in Section 2 we present a set of verification syntactic and semantic rules and how they have been derived from SEQUAL conceptual framework. In Section 3, we show how the derived rules have been implemented in Prolog clauses into our first version of the Use Case Checker (UC2). In the Conclusion section, we summarize the contributions and we outline the future work on two tracks: technological improvement of UC2 and broadening the scope of the quality assessment approach.

II. WHERE DO VERIFICATION RULES COME FROM

In order to derive verification rules, we have used the SEQUAL conceptual framework [9]. Although it was proposed more than 20 years ago, it has had great influence on verifying the quality of multiple kinds of models [10]. However, it has never been used to verify use case models. The SEQUAL framework is based on semiotic, hence it includes syntactic, semantic and pragmatic qualities where interpretations and subjectivity of modelers and users are also considered. We have specialized the SEQUAL framework as it is illustrated in Figure 1. Therefore, from a theoretical point of view, this specialization is our quality of models theory's contribution, and, our rule-based solution, may also be considered a proof of concept to it.

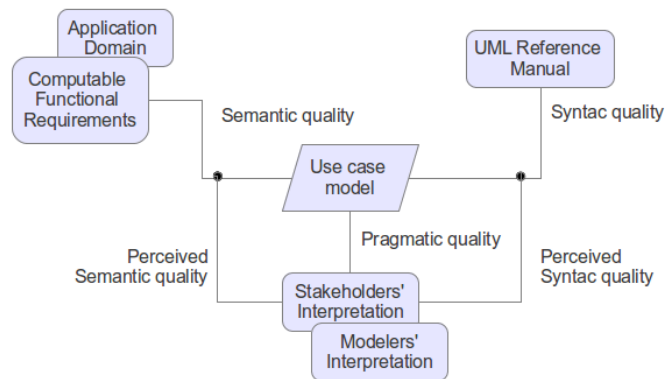


Figure 1. SEQUAL specialization for use case models

Following our SEQUAL specialization, we have generated rules to principally verify syntactic quality, but also we have included semantic quality rules in order to illustrate its implementation.

A use case model may contain several use case diagrams, and the quality application developed here is applied only to one use case diagram. Therefore, the process of verifying a use case model implies verifying all its containing diagrams.

The following rules were generated from the definition of OMGs (Object Management Group), applicable to UML use cases (specified in the UML superstructure version 2.4.1). These rules mainly verify the syntactic quality of a use case diagram, as defined in the quality model described above. Later, some of these rules will be verified automatically by the software prototype developed.

The generated rules are listed and described by adding an identification number. However, this numbering is arbitrary and does not represent any kind of hierarchy. In each case, we have illustrated the right case in opposition to a wrong case.

A. Rule 1 - There must be clear system boundaries

A use case diagram represents the interaction of an actor with the system. The UML Reference Manual says that the subject is the system under consideration to which the use cases apply [1]. Thus, it does not make any sense to model a use case diagram without a subject or system boundary, which must be represented either by a package or a classifier. Given that the existence of a system boundary may be verified in an explicit diagram structure, we classify this rule as part of the syntactic verification of a use case model. It is illustrated in Figure 2.

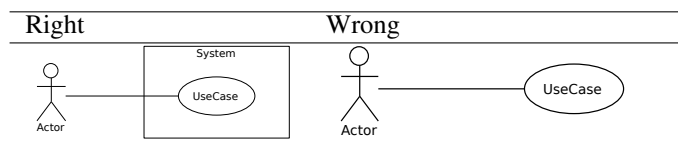


Figure 2. Example of Rule 1

B. Rule 2 - Actors must not be isolated

The UML Reference Manual says that an actor specifies a role played by a user or any other system that interacts with the subject [1]. Modeling an isolated actor does not make any sense. Although the diagram shows a system boundary and use cases in it, there is not possible to inference that the actor interacts with all or some of the present use cases. Given that the existence of an isolated actor may be verified in an explicit diagram structure, we classify this rule as part of the syntactic verification of a use case model. It is illustrated in Figure 3.

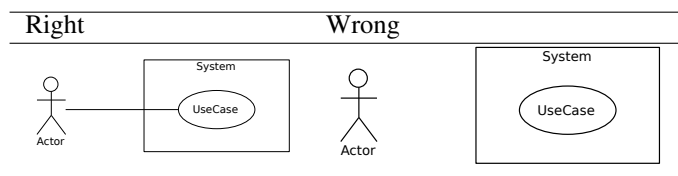


Figure 3. Example of Rule 2

C. Rule 3 - Use cases must not be isolated/inaccessible

The UML Reference Manual says that a use case represents a behavior of the system in which an actor or another system interact with it. Therefore, an isolated use case can never be executed. Here we refer not only to the fact of being isolated from actors interactions, but also of other dependencies coming from its interaction with other use cases. Given that the existence of an isolated actor may be verified in an explicit diagram structure, we classify this rule as part of the syntactic verification of a use case model. It is illustrated in Figure 4.

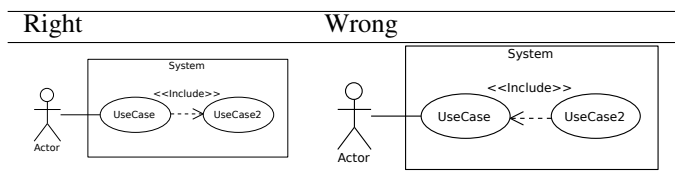


Figure 4. Example of Rule 3

D. Rule 4 - Actors must not be inside the system

The UML Reference Manual says that an actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is external to the subject [1]. Therefore, the actor should not be inside the system or subsystem being modeling. This kind of diagram does not have nested representation of involved system or subsystems. Therefore, there is no case in which it can be possible. Given that an actor being inside the system boundary may be verified in an explicit diagram structure, we classify this rule as part of the syntactic verification of a use case model. It is illustrated in Figure 5.

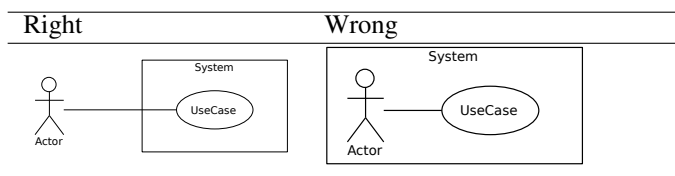


Figure 5. Example of Rule 4

E. Rule 5 - Use cases should be within system boundaries

The UML Reference Manual says that a use case is the specification of a set of actions performed by a system, which yields an observable result [1]. We, therefore, created a rule that a use case should not be outside the system boundary because it is necessary to specify which system is the owner of that behaviour. Given that a use case is part of a system, i.e., that it belongs to an existing system, then it may be verified by parsing the diagram structure, then we classify this rule as part of the syntactic verification of a use case model. It is illustrated in Figure 6.

F. Rule 6 - Use cases must start with a verb

The UML Reference Manual says that a use case is the specification of a set of actions performed by a system [1]. Therefore, its description requires, at least, a verb, i.e., the verb that specifies that action; additionally, as a quality description,

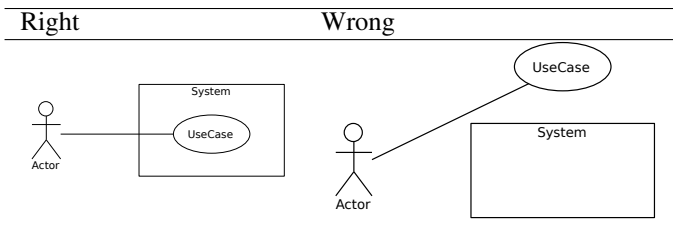


Figure 6. Example of Rule 5

we would ask for that verb to appear in the first place of the sentence that describes the use case behaviour.

In Spanish language, in which we worked, the verb should properly be used in the imperative form, but given that this mode is not commonly used, we recommend the use of verbs in its infinitive form. This last recommendation is irrelevant in English where these two forms are identical.

Given that verifying the verb form and its position in a sentence, i.e., that it is necessary to look for external references beyond UML structure, we classify this rule as part of the semantic verification of a use case model. It is illustrated in Figure 7.

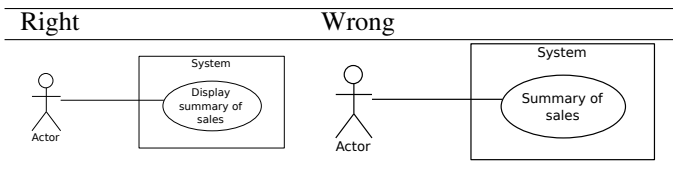


Figure 7. Example of Rule 6

G. Rule 7 - Use cases must represent an observable behavior of the system

The UML Reference Manual says that a use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system [1]. We therefore created a rule that a use case should represent actions of the system, which are observable to the actor, written from the perspective of the system.

In order to make this possible we have described a set of typical verbs of a system behaviour and other that can be warnings in the redaction of the use case descriptions. For example, a common mistake in use cases is the use of non-observable verbs like “To save” or “To register”. Also a common mistake is the use of a human actions like entering data or including high level behaviours like selecting or managing. In order to verify this feature, we have used additional list of non-observable actions (verbs) of classical system behaviours and another of classical human behaviours under a system interaction in order to give warning about them. Given that these lists are external to the own nature of UML structures, we classify this rule as part of the semantic verification of a use case model. It is illustrated in Figure 8.

H. Rule 8 - Actors names should be singular

UML superstructure says: A single physical instance may play the role of several different actors [1]. Class modeling

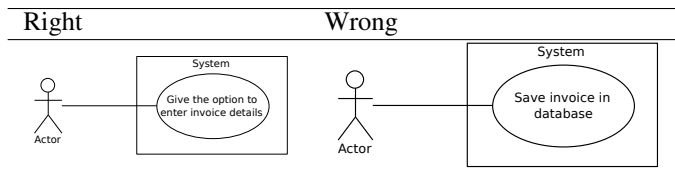


Figure 8. Example of Rule 7

assumes that actors are classes, which are required to follow the standard class nomination, including its expression in the singular case. Due to this verification goes beyond the UML structure, i.e. it exceeds syntax, we have classified this rule as part of the semantic verification of a use case model. It is illustrated in Figure 9.

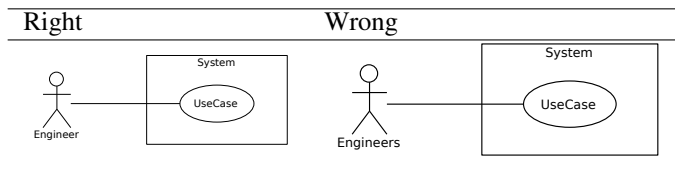


Figure 9. Example of Rule 8

I. Rule 9 - Computable verbs

The verb represents the behavior of the system, therefore the action represented by the verb must be unambiguous and capable of being implemented into a computer system. Therefore, using also a reference list of verbs we can warn about the use of a verb that is not part of “computable verbs”. Given that the existence of non computable verbs may not be verified in an explicit diagram structure, we classify this rule as part of the semantic verification of a use case model. It is illustrated in Figure 10.

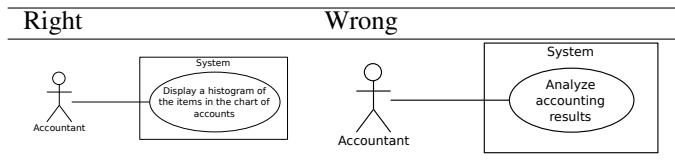


Figure 10. Example of Rule 9

III. PROLOG CLAUSES FOR USE CASES VERIFICATION

The system consists of software for analyzing a use case diagram (XMI file), applying the rules generated and subsequently delivering the result of the application in JSON (JavaScript Object Notation) format. This software will be implemented in Web technologies as a service. In Figure 11, the inputs and outputs of the system are represented.

A. Involved technologies

The XMI (XML Metadata Interchange) standard was defined for the exchange of UML diagrams. XMI is an XML-based integration framework for the exchange of models, and any kind of XML data. Thus XMI is used in the integration of tools, repositories, and model-related applications in general.

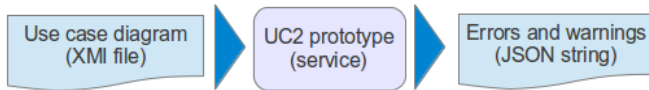
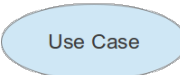
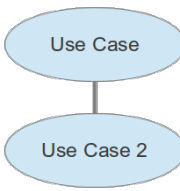
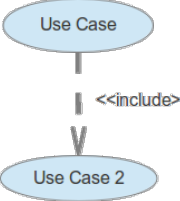


Figure 11. System view of Use Case Checker (UC2) Prototype

The framework defines rules for generating XML schemas from a metamodel based on the Metaobject Facility (MOF). Although XMI is most frequently used as an interchange format for UML, it can be used with any MOF-compliant language [11].

Table I shows the XMI representation of some elements of a use case diagram.

TABLE I. XMI REPRESENTATION OF POSSIBLE USE CASE EXPRESSIONS

Element	XMI Representation
Use Case 	<pre><packagedElement name="{NAME}" ↪ xmi:id="{ID}" xmi:type="uml:UseCase"> </packagedElement></pre>
Association 	<pre><packagedElement xmi:id="{ID}" ↪ xmi:type="uml:Association"> <ownedEnd aggregation="none" ↪ association="{ID ELEMENTO ↪ ORIGEN}" xmi:id="{ID}" ↪ xmi:type="uml:Property"> </ownedEnd> <ownedEnd aggregation="none" ↪ association="{ID ELEMENTO ↪ DESTINO}" xmi:id="{ID}" ↪ xmi:type="uml:Property"> </ownedEnd> </packagedElement></pre>
Include 	<pre><!-- An include is nested in a ↪ packageElement Use Case, this ↪ use case correspond to the UC ↪ arrow source. --> <include addition="{ID TARGET UC}" ↪ xmi:id="{ID}" ↪ xmi:type="uml:Include"> </include></pre>

SWI-Prolog is a portable implementation of the Prolog programming language. SWI-Prolog aims to be a robust, scalable implementation supporting a wide range of applications, providing interfaces to other languages and providing support for parsing XML and RDF (Resource Description Framework) documents. The system is particularly suited for server applications due to its support for multithreading and HTTP server libraries [12].

To develop UC2 we have used SWI-Prolog v 7.2.3, which provides some improvements over version 6.x.x in the creation and reading of JSON structures.

This component is responsible for implementing the pre-

viously generated rules. For this proof of concept, only some of these rules are shown.

For the implementation of the rules that verify the quality of a use case diagram, we defined a set of logical Prolog rules which allow us: 1) to identify the elements of a use case diagram within a file XMI; and 2) to see if these elements comply with the quality rules generated.

Figure 12 shows the code that identifies part of the elements within an XMI file.

```
%useCase(XML, ID, NAME)
useCase(XML, ID, NAME) :-
  xpath(XML,
    ↪ //packagedElement(@'xmi:type'=
    ↪ 'uml:UseCase'), A),
  xpath(A, /self(@'xmi:id'), ID),
  xpath(A, /self(@'name'), NAME).

%association(XML, ArrowSource,
  ↪ ArrowTarget)
association(XML, Source, Target) :-
  xpath(XML,
    ↪ //packagedElement(@'xmi:type'=
    ↪ 'uml:Association'), A),
  xpath(A, ownedEnd(1), O),
  xpath(O, /self(@'type'), Source),
  xpath(A, ownedEnd(2), U),
  xpath(U, /self(@'type'), Target).

%extend(XML, ArrowSource, ArrowTarget)
extend(XML, Source, Target) :-
  xpath(XML, //packagedElement, A),
  xpath(A, /self(@'xmi:id'), Source),
  xpath(A, extend, E),
  xpath(E,
    ↪ /self(@'extendedCase'), Target).
```

Figure 12. Prolog clauses for parsing XMI input files

When the above-defined rules are implemented, they must pass as a text parameter in the XMI file (coded as a variable named "XML").

When the Prolog Rules have been identified through the elements of a use case diagram, the hierarchical relationship between them must be known. To do this, we defined the Prolog Rule shown in Figure 13.

```
%in(XML, ChildElement, ParentElement)
in(XML, Child, Parent) :-
  xpath(XML, //packagedElement, A),
  xpath(A, /self(@'xmi:id'), Parent),
  xpath(A, packagedElement, E),
  xpath(E, /self(@'xmi:id'), Child)
```

Figure 13. Prolog clause for getting container-content relationships

B. Service Functions

In this section, we describe some of the previous rules in order to illustrate the high cohesion and low coupling reached by this Prolog implementation which has been implemented

as a REST service.

Unrelated Actor. This rule is the implementation in SWI-Prolog of Quality Rule 2 “Actors must not be isolated”. Figure 14 shows the diagram in the input file and its corresponding JSON output.

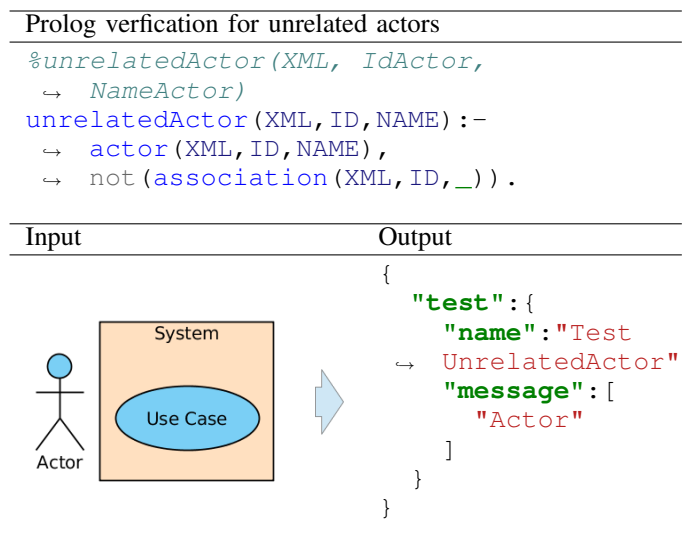


Figure 14. Implementation and sample for Rule 2: “Actors must not be isolated”

Isolated Use Case. This rule is the implementation in SWI-Prolog of Quality Rule 3 “Isolated/Inaccessible use case”. Figure 15 shows the diagram in the input file and its corresponding JSON output.

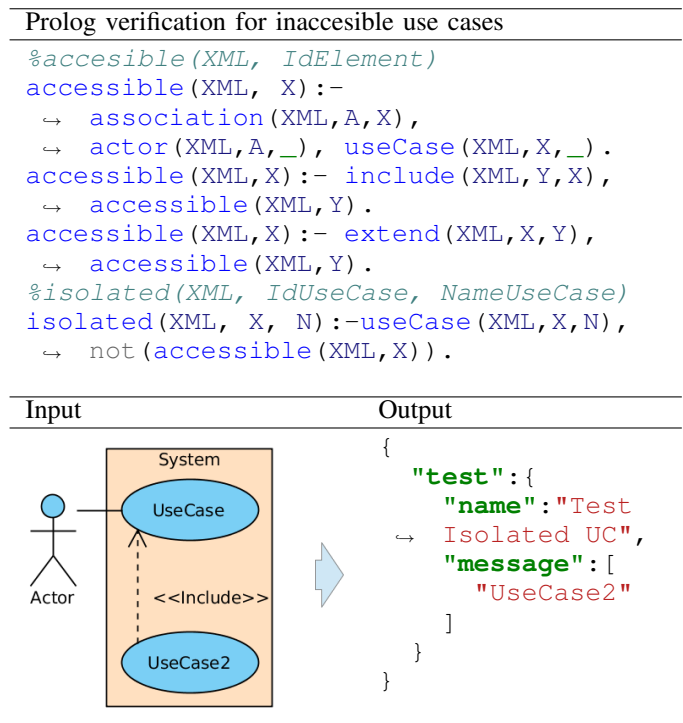


Figure 15. Implementation and sample for Rule 3: “Use cases must not be isolated/inaccessible”

Actor Inside the System. This rule is the implementation in SWI-Prolog of Quality Rule 4 “Actors must not be inside the system”. Figure 16 shows the diagram in the input file and its corresponding JSON output.

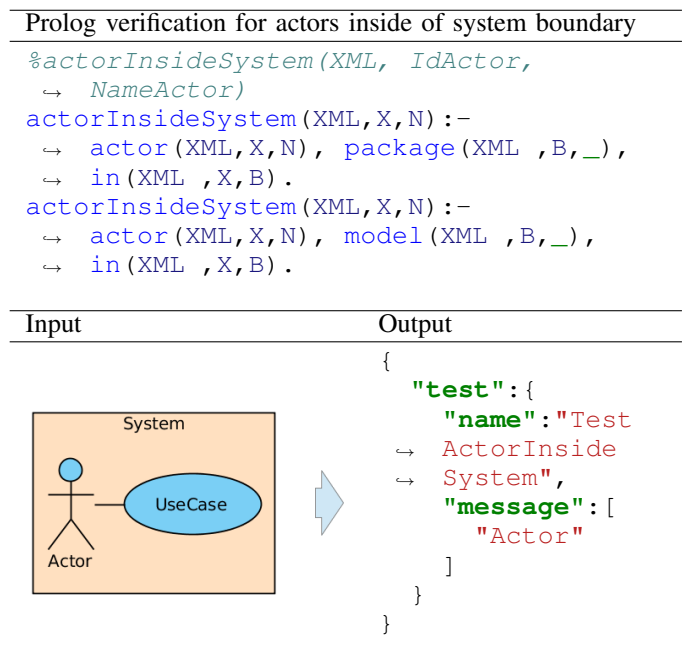


Figure 16. Implementation and sample for Rule 4: “Actors must not be inside the system”

Use Case Outside of System Boundary. This rule is the implementation in SWI-Prolog of Quality Rule 5 “Use cases should be within system boundaries”. Figure 17 shows the diagram in the input file and its corresponding JSON output.

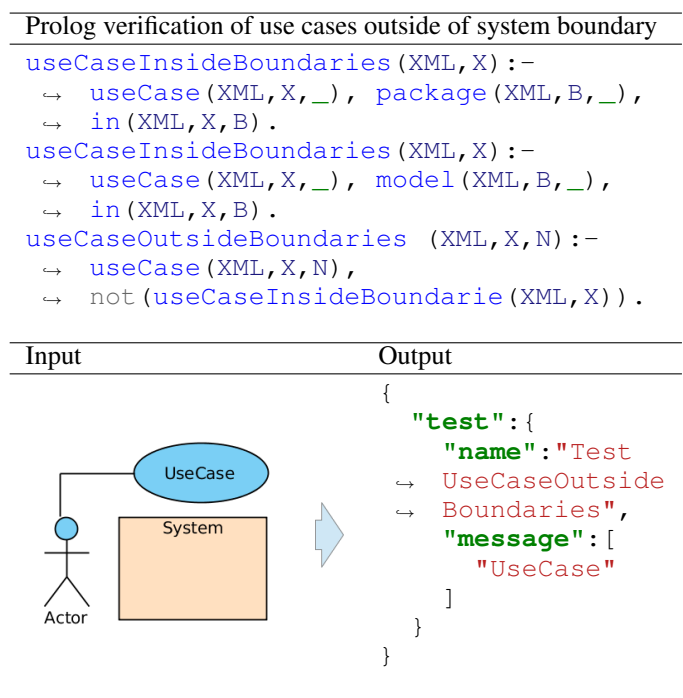


Figure 17. Implementation and sample for Rule 5 “Use cases should be within system boundaries”

Finally, as a sample of a Web site that puts together the above rules, we show the corresponding Prolog calls in Figure 18.

```
load_xml(Target, XML, []),
  findall(F, isolated(XML,_,F),
  ↪ Isolated),
  findall(F,
  ↪ useCaseOutsideBoundarie(XML,_,F),
  ↪ UseCaseOutsideBoundarie),
  findall(F, actorInsideSystem(XML,_,F),
  ↪ ActorInsideSystem),
  findall(F, unrelatedActor(XML,_,F),
  ↪ unrelatedActor),
  ↪ reply_json(json([
  ↪ test=json([name='Test Isolated',
  ↪ message=Isolated]),

  ↪ test=json([name='Test
  ↪ UseCaseOutsideBoundarie',
  ↪ message=UseCaseOutsideBoundarie]),

  ↪ test=json([name='Test
  ↪ ActorInsideSystem',
  ↪ message=ActorInsideSystem]),

  ↪ test=json([name='Test UnrelatedActor',
  ↪ message=UnrelatedActor])
  ]))
```

Figure 18. Prolog clauses for return the result of all the rules as a JSON response.

Additional expressions like extends or the representation in XMI of include are not explicitly represented in this document. The full UC2 source code is available at [13].

IV. CONCLUSION

Use cases are a common choice to specify software requirements. In spite of existing theoretical approach to use case verification, there are no known implementations that put together a theoretical background about quality of models and a derived implementation. In this paper, we propose a general quality framework and we show an initial implementation for verifying use case model, represented on demonstrations that cover the part of our system realized to the moment. This rule-based approach besides the chosen architecture, shows that a declarative approach is not only effective but also a solution presenting two main characteristics: low coupling and high cohesion which allows an easy maintenance and high scalability. Moreover, the implemented rules allow to report both errors and warnings, which can be used in a software process to improve quality of use cases due to, even being a partial approach as is, these results are measurable and repeatable ones, which makes it a valuable under the perspective of software engineering at any label, as part of a computer-aided step in a traditional development approach, or as a part into a model driven development approach.

Future work is related to limitations of the current approach. Firstly, the integration of SWI-Prolog as part of a Web service impose a limitation because it does not provide a good integration to classical Web services as Apache. Thus, we need

to review the technology behind the current solution approach. However, the most relevant challenge is to add pragmatics rules and to work with perceived quality. Classical features of a set of requirements, i.e., completeness and consistency, can not be validated without considering the context to which the system has been conceived to work, i.e., the set of stakeholders' expectations. However, this approach seems to be a way to reach it.

ACKNOWLEDGMENT

The authors would like to thank DIUFRO project DI13-0068 from the Vice-rectory of Research and Development and the Master Program of Informatics Engineering both from University of La Frontera, by supporting different aspects of this work.

REFERENCES

- [1] O. UML, "2.4. 1 superstructure specification," document formal/2011-08-06. Technical report, OMG, Tech. Rep., 2011.
- [2] B. Berenbach, "The evaluation of large, complex UML analysis and design models," in *Proceedings. 26th International Conference on Software Engineering*. Institute of Electrical & Electronics Engineers (IEEE), 2004.
- [3] G. Kösters, H.-W. Six, and M. Winter, "Coupling use cases and class models as a means for validation and verification of requirements specifications," *Requirements Engineering*, vol. 6, no. 1, pp. 3–17, Feb 2001.
- [4] Y. Kotb and T. Katayama, "A novel technique to verify the uml use case diagrams," in *IASTED Conf. on Software Engineering*, 2006, pp. 300–305.
- [5] Y. Shinkawa, "Model checking for UML use cases," in *Software Engineering Research, Management and Applications*. Springer Science Business Media, 2008, pp. 233–246.
- [6] S. Gruner, "From use cases to test cases via meta model-based reasoning," *Innovations Syst Softw Eng*, vol. 4, no. 3, pp. 223–231, Aug 2008.
- [7] S. Tena, D. Díez, P. Díaz, and I. Aedo, "Standardizing the narrative of use cases: A controlled vocabulary of web user tasks," *Information and Software Technology*, vol. 55, no. 9, pp. 1580–1589, 2013.
- [8] M. Oliveira Jr, L. Ribeiro, É. Cota, L. M. Duarte, I. Nunes, and F. Reis, "Use case analysis based on formal methods: An empirical study," in *International Workshop on Algebraic Development Techniques*. Springer, 2015, pp. 110–130.
- [9] J. Krogstie, O. I. Lindland, and G. Sindre, "Towards a deeper understanding of quality in requirements engineering," in *Advanced Information Systems Engineering*. Springer Science + Business Media, 1995, pp. 82–95.
- [10] J. Krogstie, G. Sindre, and O. I. Lindland, "20 years of quality of models," in *Seminal Contributions to Information Systems Engineering*. Springer, 2013, pp. 103–107.
- [11] M. Weiss, "Xml metadata interchange," in *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, pp. 3597–3597.
- [12] J. Wielemaker, S. Ss, and I. Ii, "Swi-prolog 7.2.3-reference manual," 2015.
- [13] F. Bautista and C. Cares. Uc2: A prolog checker for use cases. [Http://dci.ufro.cl/fileadmin/Software/UC2.zip](http://dci.ufro.cl/fileadmin/Software/UC2.zip), [retrieved: January, 2017].