CLEI ELECTRONIC JOURNAL, VOLUME 22, NUMBER 1, PAPER 2, APRIL 2019

An Ontology for Specifying and Tracing Requirements Engineering Artifacts and Test Artifacts

M. Luciana Roldán, Marcela Vegetti, Silvio Gonnet, Horacio Leone
Instituto de Desarrollo y Diseño (Conicet/UTN),
Santa Fe, Argentina, 3000
{lroldan, mvegetti, sgonnet, hleone}@santafe-conicet.gov.ar

and

Marcelo Marciszack
Centro de Investigación, Desarrollo y Transferencia de Sistemas de Información (CIDS),
Córdoba, Argentina, 5000
marciszack@gmail.com

Abstract

Nowadays, modern software development processes follow an iterative approach, which makes possible to start the testing of a system at early stages. This approach requires recording the requirements artifacts that specify the functionality or characteristics required by the system, and the test cases that are derived from each requirement artifact. Frequently, software development organizations employ supporting tools to create and maintain these artifacts. There exist numerous tools for supporting requirements specification activities, as well as the definition and execution of test cases. These separate tools have their own databases and metamodels. The lack of integration between these tools leads to difficulties in tracing related artifacts and obtaining useful knowledge to manage the developing process. It is necessary to understand without ambiguities the concepts used by the different tools to allow them to interoperate. This paper proposes an ontology that defines and integrates the concepts included by the metamodels of different Requirements Engineering and Testing Management supporting tools. The formalization of these concepts and their relationships in an ontology language prevents ambiguity of the concepts and permit to the tools involved to interoperate with each other, to achieve semantic consistency and the tracing of artifacts. The proposed ontology used in conjunction with a reasoner provides capabilities to infer traces that are not explicit, which makes it possible to easily maintain artifacts and associations between them. The approach facilitates backward tracing from test cases to use cases and functional requirements artifacts, obtain knowledge about the causes of a defect or a poor specification, and enable impact analysis.

Keywords: Functional requirements, Artifacts tracing, Ontologies, Traceability, Knowledge inference, Test cases, Requirements engineering.

1 Introduction

There exists a close relationship among the activities involved in Requirements Engineering and software intensive systems testing, which is reciprocal and positive. On the one hand, a complete, consistent and legible specification of requirements allows a good definition of test cases [1]. On the other hand, the testing activities that are performed to verify a system also contribute to the verification and improvement of requirements artifacts, since the test cases specification activities have a positive effect on the improvement on quality of the requirements specification. Testing activities require a validated and released set of requirements. Therefore, having a complete, consistent and legible set of fully specified requirements is beneficial for the definition of high-quality test cases.

Nowadays, most employed modern development processes follow an iterative approach, where it is possible to start testing the system at early stages. In this line, the present proposal promotes the use of requirements artifacts (use cases specifications) to generate tests case artifacts [2]. Use cases [3] are key for the testing process, to derive test artifacts, to identify what conditions should be taken into account when developing the test cases, and to verify the proper implementation of that all the requirements. Frequently, software development organizations employ supporting tools to create and maintain these artifacts. Therefore, it is not only important to record the requirements artifacts that specify the functionality or characteristics required by certain systems (textual requirements specifications, use cases, use case scenarios, etc.), but also to know which test cases were derived from each requirement artifact. Besides, it is important to facilitate backward tracing from test cases to use cases and functional requirements artifacts, thus to obtain knowledge about the causes of a defect or a poor specification, and to enable impact analysis.

Numerous tools for supporting requirements specification activities, as well as for supporting the definition and execution of test cases exist. These separate tools have their own databases and metamodels. The lack of integration among these tools leads to difficulties in tracing related artifacts and obtaining useful knowledge to manage the developing process. It is necessary to understand without ambiguities the concepts used by the different tools to allow them to interoperate.

In this paper, an ontology is developed, which defines the concepts involved in requirements engineering and testing activities, and the relationships among them, with the aim of achieving the required semantic interoperability. In computer science, an ontology is a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. Ontologies are used to describe the domain and to reason about the properties of that domain. Ontologies provide a shared vocabulary, which can be used to model a domain in terms of the type of objects that intervene, and/or concepts that exist, and their properties and relations [4].

In a previous work [5], we introduced a preliminary ontology of functional requirements and use cases, with the aim of supporting the automatic derivation of test cases from use cases. In this paper, the ontology is enriched with additional concepts, relationships, properties, and rules taking advantage of the capabilities to represent complex knowledge that ontologies offer. Besides, the potential of ontologies for knowledge retrieval and inference makes possible to trace the artifacts generated at different software development process stages and with different supporting tools (requirements artifacts, tests artifacts, and intermediate artifacts if any). Besides, the main methodologic steps followed for the ontology development are introduced.

The paper is organized as follows. In section 2, we introduce some general topics about ontologies, traceability issues, and derivation of test cases from use cases. In section 3, we outline the main methodologic steps followed for the ontology development. In section 4, the purpose and scope of the ontology are defined by means of competency questions, and the conceptual model that underlies the ontology is described. In section 5, the implementation of the proposal is detailed and illustrated with a scenario that shows how traceability links between artifacts are retrieved and inferred. In section 6, related works are mentioned. Finally, the conclusion is provided in section 7.

2 Background

2.1 Ontologies

In the context of software developing, numerous studies have been proposed where ontologies play a major role. Particularly, in Requirements Engineering, the concept of ontology has been used to minimize or resolve different types of problems. For instance, to improve the completeness of requirements specification and to help to disambiguate its meaning, to perform consistency analysis in requirements, to assist in explicitly modeling domain knowledge, to manage requirements knowledge and requirements changes and so on [6].

In computer science, ontology is a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain and may be used to describe the domain [4]. In theory, ontology is a "formal, explicit specification of a shared conceptualization". Ontology provides a shared vocabulary, which can be used to model a domain that is, the type of objects, and/or concepts that exist, and their properties and relations [7]. Ontologies make possible the hierarchical classification of interrelated domain concepts and can be formalized using an RDF[1] schema or the OWL[2] language. These languages enable ontologies to be read both by humans and machines and obtain knowledge by means of queries and reasoning to infer knowledge that is not explicitly stated. Additionally, semantic web languages like RDF and OWL facilitates interoperability. They provide a social structure and technical framework for reusing available ontologies, and formal mechanisms to express logical equivalence among classes and properties that belong to different ontologies.

[1] https://www.w3.org/TR/2004/REC-rdf-primer-20040210/
[2] https://www.w3.org/TR/owl2-overview/

2.2 Traceability

According to IEEE St. 610-12-1990 [8] traceability denotes, in general, the degree to which a relationship can be established between different development artifacts, especially artifacts having a predecessor-successor or master-subordinated relationship to one another. A requirement is considered being traceable if the origin of the requirement as well as its further use in the development process can be traced. As stated by Gotel and Finkelstein [9], Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction.

Traceability relations can be documented in different ways [1]. The simplest way of documenting a traceability relationship is to include the identifier of the target artifact as a textual annotation to the source artifact. Another way is by using hyperlinks. To document a traceability relationship a hyperlink from the source artifact to the target artifact is created. Different types of traceability relationships can be documented by defining different hyperlinks types. The traceability relationship types can be defined and structured in a traceability information model. Such traceability model defines, for a specific project, the traceability artifact types and the traceability relationship types between artifacts to be used in this project. Furthermore, such a model can define cardinalities for the relationship types as well as specializations relationships between different types and artifact types.

Besides the traceability documentation approach employed, there are different ways of presenting (visualizing) traceability information. A way of doing it is by using traceability matrices. The rows of a traceability matrix represent the source artifacts considered in the matrix, and the columns represent the target artifacts. An artifact can be represented in the matrix by its identifier. If a traceability relationship exists between the source artifact of row i and the target artifact of column j, cell (i, j) in the traceability matrix is marked. A traceability matrix can be used for representing a single relationship type. In practice, due to the large numbers of requirements, the use of traceability matrices for visualizing traceability information is limited.

Another kind of traceability visualization is traceability graphs. In a traceability graph, the nodes represent artifacts and the edges represent traceability relationships between the artifacts. In addition, to distinguish the different artifact types and/or different traceability relationship types, different type of nodes or edges can be defined. Alternatively, an attribute can be assigned to each node and/or edge of the graph to denote the artifact or relationship type. During traceability analysis, traceability graphs are often analyzed along specific paths consisting of multiple nodes and edges. Traceability paths are particularly important for change management. A limitation of traceability graphs is that is difficult to trace and analyze long paths only visually, so a language is needed to query and retrieve traceability paths efficiently. In addition, if redundant or alternative traceability relationships between the artifacts exist, they become difficult to maintain.

2.3 Derivation of test cases from requirements artifacts

Several authors have proposed strategies for the derivation of test cases based on requirements artifacts [1], [10], [11]. While some of them raise the possibility of deriving test cases directly from the artifacts (such as the System Requirement Specification document, use cases diagrams, or textual use cases) [2], others propose test case derivation from the so-called "test models" (model-based test-case derivation approach) [12], [13]. When deriving test cases directly from requirement artifacts, the test cases are constructed without constructing preliminary artifacts. Since a complete test is not possible, a number of representative test cases must be selected. This selection can be made based on experience, but this approach is no systematic and thus important test cases can be overlooked. Test cases can be determined systematically by means of equivalence class partitioning or using scenarios from requirement engineering as a starting point. In the latter approach, requirement scenarios can be used as test-case scenarios and, based in these test-case scenarios, test cases can be derived. Test cases are derived by defining concrete inputs (and preconditions), expected outputs (and postconditions), and test instructions. The direct derivation of test cases from requirements artifacts provides the advantage that no additional notation or documents have to be introduced. Furthermore, all requirements artifacts can be used as a starting point since the used artifacts do not have to be formalized or interpretable by tools. During the derivation of test cases from requirements scenarios, the quality of the test also strongly depends on the completeness of the requirements specification. This is a problem, as requirements scenarios are generally not created with the objective of defining the system behavior completely. This shortcoming is addressed by model-based test-case derivation.

In the second strategy for the derivation of test cases, a test model serves as the basis for systematically deriving test cases [12]. If during Requirement engineering, model-based artifacts like state machines (finite automata, UML state diagrams, UML activity diagrams), and flowcharts were generated, then these artifacts are suitable for being used as test models. If this kind of (model-based) requirement artifacts do not exist, then a test model has to be created based on the existent requirement artifacts for the sole purpose of supporting testing. In this approach, each path through a test model represents a test-case scenario. By selecting paths systematically, a subset of all possible test-case scenarios can be identified for the test.

3 Methodology for the ontology development

To develop the proposed ontology, an ad-hoc methodology was applied, which is based on principles globally accepted in the ontology engineering domain. It comprises four stages, which are common to most popular ontology development methodologies [4], [14], [15], [16]:

- *Ontology requirements specification*. This stage consists of identifying the purpose for constructing the ontology and set its scope.
- *Conceptualization*. This stage consists of organizing the involved concepts in an informally perceived view of the domain and, then, converting it in a semi-formal UML-based specification.
- *Implementation/Formalization*. This stage implies the codification of the ontology using a formal language.
- *Evaluation*, which allows making a technical judgment of the ontology quality and usefulness with respect to the requirements specification, competency questions and/or the real world

It should be noted that any ontology development is an iterative and incremental process; indeed, these stages are not truly sequential. If some need/weakness is detected during the execution of a stage, it is possible to return to any of the previous ones to make modifications and/or refinements. Some highlights of these methodological steps are given in the remaining of this section.

Regarding the first stage (*Specification*), the approach followed to determine the requirements and scope of the ontology to develop, is by outlining a list of preliminary competency questions, as Gruninger and Fox [14] propose. The list of competency questions does not need to be exhaustive. Later, at the Evaluation stage, they will be used to validate and check the quality of the defined ontology. Besides, competency questions serve to verify if the ontology contains enough information to answer them, and thus to analyze if further information details are required or a portion of knowledge about a particular area that has not been fully addressed.

Then, from the defined competency questions, the relevant terms related to the domain are identified (second column of Table 1). Initially, a list of terms is generated without taking into account if any overlapping exists among the concepts represented by the terms, the possible relationships among them, or the properties that characterize each concept. Then, at the *Conceptualization* stage, each class or concept is defined from the terms, which results in a taxonomy depending on the generalization/specialization relationships that exist among the concepts and their properties [4]. Since the classes by themselves do not provide enough information to answer the competency questions, it is necessary to describe the internal structure of the represented concepts by means of properties. Generally, the remaining terms that do not constitute classes are properties. Then, each of these concepts has to be analyzed in order to decide if it represents an intrinsic property, an extrinsic property, or it is part of a more complex object, or if it constitutes a kind of relationship to another concept. As a result of this stage, a series of conceptual models arise, which are described in section 4.

The *Implementation* stage requires the selection of a formal language suitable for the codification of the concepts that were identified in the previous stage. Based on its ample acceptance, OWL 2, developed by the W3C (World Wide Web Consortium), was chosen. OWL 2 is an ontology language for the Semantic Web with a formally defined meaning. OWL 2 is an extension of OWL designed to facilitate ontology development and sharing via the Web, with the ultimate goal of making Web content more accessible to machines. OWL 2 ontologies provide classes, properties, individuals, and data values and are stored as Semantic Web documents. OWL 2 ontologies can be used along with information written in RDF, and any OWL 2 ontology can also be viewed as an RDF graph (set of RDF triples *subject*, *predicate*, *object*). Protégé[3] tool was employed for editing the ontology. Protégé, besides being an ontology editor, provides deductive classifiers useful for checking model consistency and for inferring new knowledge.

Finally, at the *Evaluation* stage, the activities carried out include populating the ontology with instances gathered from several case studies, formalizing the competency question in the SPARQL[4] language, and executing queries on the knowledge represented in the case studies with the aim of obtaining properly answers to the competency questions. Some of the initial results that were obtained are mentioned in section 5.

The ontology for tracing requirements and test artifacts was developed by following the introduced methodology. During the *Specification* stage, various competency questions were stated, which are partially shown in Table 1. In order to lead the definition of competency questions and to avoid neglecting aspects of concern, several dimensions or categories were regarded, which also enable grouping related competency questions. In Table 1, the *Group* column indicates the questions categories: functional requirements, use cases, test cases, traceability, etc. The list of

[3] https://protege.stanford.edu/

[4] https://www.w3.org/TR/rdf-sparql-query/

competency questions is used in the next stage to identify the relevant terms for specifying the classes, properties, and data values of the ontology under construction. The next sections describe in detail the *Conceptualization*, *Formalization* and *Evaluation* stages.

Table 1: Competency Questions

Competency Questions	Terms	Group
Which are the functional requirements of a system? Which artifacts are created for the system definition?	*Functional Requirement, System, Artifact*	*Functional requirement definition*
Which use cases specifications pertain to a given functional requirement?	*Use Case, Functional Requirement*	*Functional requirement definition*
Which is the goal of a given use case?	*Goal, Use Case*	*Use case definition*
Which are the actors involved in a given use case?	*Actor, Use Case*	*Use case definition*
Which are the preconditions regarded in a given use case?	*Precondition, Use Case*	*Use case definition*
Which is the main scenario of a given use case? Which are the steps of a given alternative scenario? Does exist alternatives or exceptions for a given main scenario?	*Use Case, Main Scenario, Scenario Step*	*Use case definition*
Which are the alternative scenarios of a given use case? Which steps comprise a given alternative scenario?	*Use Case, Alternative Scenario, Scenario Step*	*Use case definition*
Which are the exception scenarios of a given use case? Which steps comprise a given exception scenario?	*Use Case, Exception Scenario, Scenario Step*	*Use case definition*
Which are the postconditions of a given use case?	*Postcondition, Use Case*	*Use case definition*
Given a functional requirement expressed in a use case, which is the deterministic finite automata that models it?	*Functional Requirement, Use Case, Deterministic Finite Automata*	*Traceability*
Which is the alphabet of input symbols handled by certain deterministic finite automata? Which are its states? Which are its transitions?	*Alphabet, Input symbol, State, Transition*	*Behavior/test model definition*
Which are the correspondences or equivalences of the states of a deterministic finite automaton and the steps of the different scenarios within a use case?	*State, Scenario State, Use Case*	*Behavior/test model definition*
Given a bifurcation in a course of action of a use case, which are the corresponding transitions in the transition function of the deterministic finite automaton? Which conditions are verified?	*Transition, Condition*	*Behavior/test model definition*
Which are the possible final states reached by the deterministic finite automata? Which of those final states are produced by error conditions?	*State, Final State, Error Final State*	*Behavior/test model definition*
Which type of test models are used to derive test cases from functional requirements specifications?	*Test Model, Test Case, Functional Requirement*	*Behavior/test model definition*
Which are the test cases scenarios defined for a certain use case?	*Test Case Scenario, Use Case*	*Traceability*
Which test cases are derived from a use case?	*Test Case, Use Case*	*Traceability*
Which test cases scenarios are derived from a test case?	*Test Case, Test Case Scenario*	*Traceability*
Which are the types of input data that are defined for a certain test case scenario?	*Input Data Type, Test Case Scenario*	*Test case definition*
Which are the types of expected results that are defined for a certain test case scenario?	*Expected Result Type, Test Case Scenario*	*Test case definition*
Which criterion of coverage was defined for the test cases scenarios of a certain test model?	*Test Model, Coverage criterion*	*Test case definition*
Which coverage percentage was set for the test cases scenarios of a certain test model?	*Test case scenario, Coverage percentage*	*Test case definition*
Which test flows or sequences of states were selected for the derivation of test cases from a certain test model?	*Test Flow, Sequence of States, Test Model*	*Test case definition*

4 Conceptual Model

4.1 Functional Requirements and Use Cases

Functional requirements are expressed in the form of use cases. Use cases describe a sequence of interactions that an actor can carry out with a system to achieve some valuable result [3], [17]. Pohl defines the use cases as the specification of sequences of actions, including variant sequences, and error sequences, that a system, subsystem, or class can perform by interacting with outside objects to provide a service of value [1]. Besides, the sequence of actions comprised by a use case pursues the satisfaction of a common (set of) goal(s). In other words, a use case is a collection of related usage *scenarios*, which also implies that a scenario is a specific instance of a use case in which it is traversed, a possible course of action.

To overcome the limitations of natural language, textual descriptions based on structured templates that include a series of attributes are used for the specification of use cases. In Fig. 1, a model is presented that describes the main concepts involved in a use case template [1], [17]. The objectives (*Goal*) represent the intentions of the stakeholders (*Actor*). A scenario describes a concrete example of satisfying or failing to satisfy a goal (or set of goals). As indicated in Fig. 1, different kind of scenarios exist in a use case.

A main scenario (*MainScenario*) documents a sequence of interactions or steps (*ScenarioStep*) that is normally executed in order to satisfy a specific set of goals. An alternative scenario (*AlternativeScenario*) documents a sequence of interactions that can be executed instead of a main scenario and that results in the satisfaction of the goals that are associated with the main scenario. On the contrary, an exception scenario (*ExceptionScenario*) documents a sequence of interactions that occur when an exceptional event occurs and as a consequence of it, one or multiple goals associated with the original scenario cannot be satisfied. Thus, an exception scenario documents a sequence of interactions that is executed instead of the interactions documented in another scenario. The use cases (*UseCase*) group a main scenario with cero, one or more corresponding alternative and exception scenarios. Finally, a use case scenario (*UseCaseScenario*) is a valid sequence of interactions from the main, alternative, and exception scenarios defined for the use case, which can lead to a defined termination of the use case (satisfaction of associated goals or abort). The steps (*ScenarioStep*) included in the use case scenarios can represent a concrete action that performs the system or the actor. On the other hand, a scenario step can represent the evaluation of a condition that determines the next step to be executed (thus, it configures a possible bifurcation on the use case). To indicate this, the data property named type is defined. In the former case the value of the type property is "*Sequence*," whereas, in the latter, its value is "*Condition*".

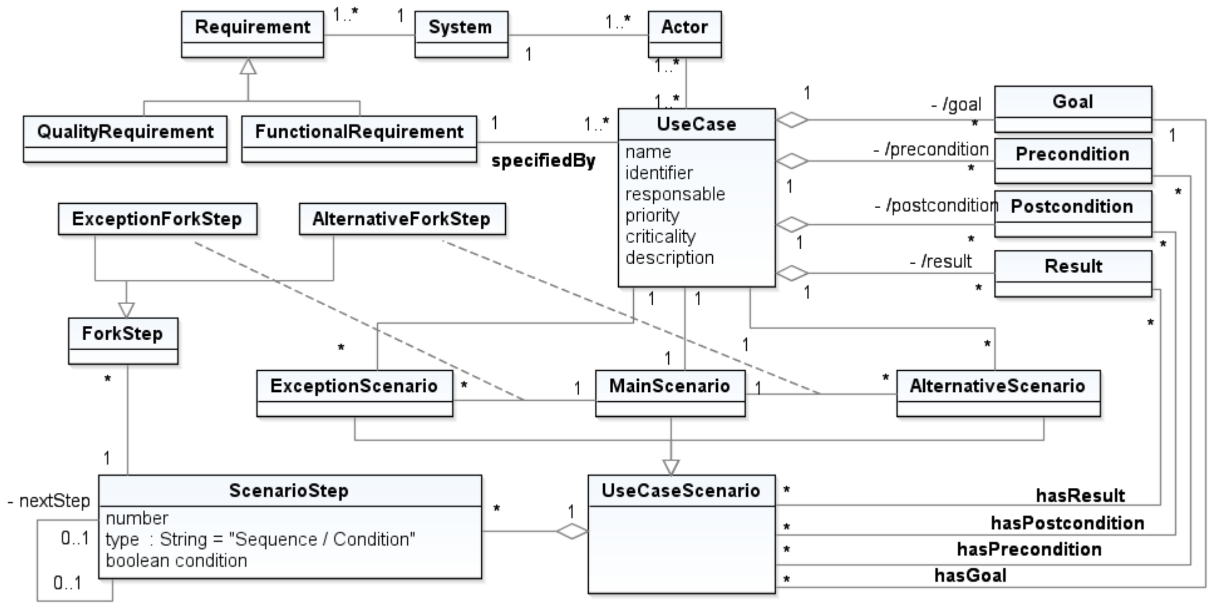


Figure 1: Use Cases and Scenarios Conceptual Model

4.2 Formalization of Use Cases using Deterministic Finite Automata

In a previous work [5], the authors developed a method to formalize the behavior of a use case by means of an equivalent Deterministic Finite Automata (DFA), which is based in the proposal of Marciszack et al. [18]. This proposal makes use of abstract machines for the specification and consistency checking of functional requirements. The method was implemented in a software tool called SIAR (Sistema Integral para la Gestión de Requisitos-Integral System for Requirements Management). SIAR is a tool for representing functional requirements in the form of template-based use cases. One of the functionalities provided by the tool is an automated procedure for consistency analysis of use cases, through the verification of the sequences of actions described in the case of use. With this approach, the system generates a DFA from a use case specification. Then, a simulator of deterministic finite automata carries out an analysis to verify the cohesion and consistency of the scenarios defined by the use case. Deterministic finite automata constitute another artifact type to be defined in the proposed ontology.

Finite deterministic automata [19] are often used to document the reactive behavior of a system. Formally, a deterministic finite automaton is defined as a quintuple ($Q, \Sigma, \delta, q0, F$), where its components are:

1. A finite set of states (Q).
2. A finite set of input symbols (Σ).
3. A transition function that has as arguments a state (belonging to Q) and an input symbol (belonging to Σ) and returns a state (belonging to Q). The transition function is usually referred to as δ. In a graph representation of

the automata (as will be seen later), δ is represented by arcs between the states (or nodes of the graph) and the labels on the arcs. If *q* is a state and *a* is an input symbol, then *δ(q, a)* is the state *p* such that there is an arc labeled as *a* that goes from *q* to *p*.

4. An initial state *q0*, one of the states of *Q*.
5. A set of end states or acceptance states *F*, where *F* is a subset of *Q*.

In addition to the mathematical representation as a quintuple, a *DFA* can be represented as a graph called "transitions diagram", where: a) for each state *q* of *Q*, there is a corresponding node; b) for each state *q* of *Q* and each input symbol *a* of *Σ*, be *δ* (*q*, *a*) = *p*, the transitions diagram has an arc from node *q* to node *p*, labeled with an *a*; c) there is an arrow directed to the initial state *q0* that has no origin in any node, which represents the start state; d) the nodes corresponding to the acceptance states (those belonging to *F*) are marked with a double circle. States that do not belong to *F* have a simple circle.

In [5], the elements that intervene in the *DAF* artifact were depicted in a conceptual model, which is represented in Fig. 2. The concepts *DFA*, *State*, *Transition*, and *InputSymbol* (in Fig. 2) and the relationships between them were included in the ontology proposed in the present work.

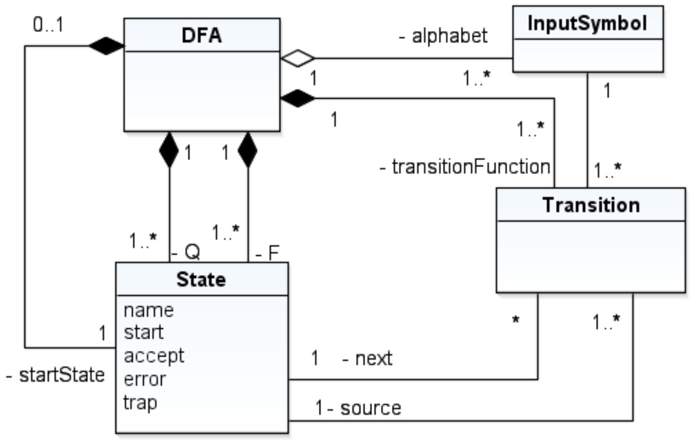


Figure 2. Conceptual model of Deterministic Finite Automata

In order to transform a use case into a deterministic finite automaton, a preliminary set of conversion rules was defined, which allows obtaining an DFA from a use case specification [18]. An example of a template-based use case specification [1][20] is shown in Fig. 3. As it can be viewed, a use case specification comprises one main scenario, cero o more alternative scenarios, and cero o more exception scenarios. Scenarios include a series of steps that indicate the sequence of interactions that take place. Besides, alternative and exception scenarios have a “void” first step that represents the condition that is evaluated before proceeding with the steps in the scenario.

In [5] the conversion rules were refined and enhanced to overcome some limitations that the original approach had, which was not adequate for the transformation of cases of uses with bifurcations due to multiple alternative scenarios. The transformation rules are listed below:

Rule 1. Each step of a course of action of a use case (in any of its scenarios) corresponds to a state node in the *DAF*. The conditions (initial step of an alternative scenario or exception scenario) are also considered steps that have their corresponding state node in the *DFA*.

Rule 2. Every use case has an initial step, which is the first step of all possible courses of action (or flows) and is given by the first step of the main scenario. This step constitutes the initial state/node of the generated *DFA*.

Rule 3. Every use case has a final step in the normal course of action given by the main scenario. This last step constitutes one of the possible final states of the *DFA* (belongs to the set *F*). Reaching such a final state means the satisfaction of the goals of the use case.

Rule 4. A course of action within a use case may not have, have one, or have several final states by mistake, each of these being the last step of an exception scenario. These steps are also final states/nodes of the *DFA*.

Rule 5. Any possible sequence from a scenario step of the use case to another scenario step corresponds to a transition from the *DFA*.

Rule 6. If in a certain step of the main scenario of the use case, there is a possible bifurcation towards an alternative scenario, an outgoing transition will be generated from the state corresponding to that step to the state that represents the first step of the corresponding alternative scenario. This means that starting from a given source state / node it is possible to reach a single destination state / node, or two destination nodes / states.

Use Case ID	CU-01	
Name	Login to the system	
Version	V.1.0	
Description	A user wants to access the system, which means that he / she must "login" with their username and password.	
Actors	User	
Goal	Enter the system and obtain credentials	
Preconditions	The user needs to be registered in the system	
Postconditions	The user is logged in the system. The user's access was recorded in the activity table.	
Results	The user enters the system and can perform actions on it until explicitly leaving the system or until some time of inactivity has passed.	
Main Scenario	1	The user accesses the system through a browser by entering the URL
	2	The user selects the menu option to enter the system
	3	The user provides his/her username and password
	4	The system validates the user's credentials and welcomes him/her
Alternative Scenarios	3a	The user does not remember his password
	3a1	The user requests to recover the password by email
	3a2	The system sends a temporary key to the email account that the user provided when registering
	3a3	Continue to step 3
	3b	The user is not registered in the system
	3b1	The new user requests to create an account
	3b2	The system requests the data to create the account
	3b3	The user enters the requested and optional data requested and confirms them.
	3b4	The system creates the user's account
	3b5	Continue to step 2
	4a	Incorrect password. Try <3
	4a1	The system gives a message to the user informing that the password is incorrect.
	4a2	Continue to step 2
Exception Scenarios	4b	Incorrect password. Try =3
	4b1	The system gives a message and blocks the account for security.

Figure 3. Template-based specification of CU-01 Use Case

Rule 7. A *DFA* associated with a use case has an alphabet of two symbols *N* and *A*, which indicates the event that causes each transition from one state to another:

- *N*: The transition that occurs continues with the course of action determined by the current scenario.
- *A*: The transition that takes place bifurcates the current course of action to an alternative or exception scenario. This implies that a condition must be evaluated, which is the first step of the alternative or exception scenario. If the given condition is met, the course of action continues as indicated by the scenario (transition given by a symbol *N*). If the condition is not verified and there is another alternative scenario for that bifurcation, it is represented as a transition labeled by *A* that reaches the first step of this new scenario. This is repeated successively until there are no other alternatives for the same step. In this way, the *DFA* alphabet is determined by the $\Sigma = \{N, A\}$.

Therefore, any course of action valid in a use case (also called "flow") is given by a sequence of input symbols, according to which it is possible to start from the initial state (first step in the main scenario) and go through all the transitions until reaching a final state (*DFA* acceptation state) belonging to F, this being the final step of the main scenario or a final step of an exception scenario.

To illustrate how a DFA is generated from a use case specification by means of the applying of the previously defined transformation rules, the use case shown in Fig. 3 was considered. The employed template is based on the proposed ones by several authors [1, 20, 21], and it was implemented in the SIAR system to normalize the specification of use cases. This use case describes the interactions that occur when a user accesses a web system.

In Fig. 4 (a), the DFA that models the behavior of the use case CU-01 is shown. As can be seen, the different possible flows on the use case are formed by integrating all the scenarios defined in it. For obtaining the *DFA*, the rules previously defined were applied. The formal definition of the use case is expressed as:

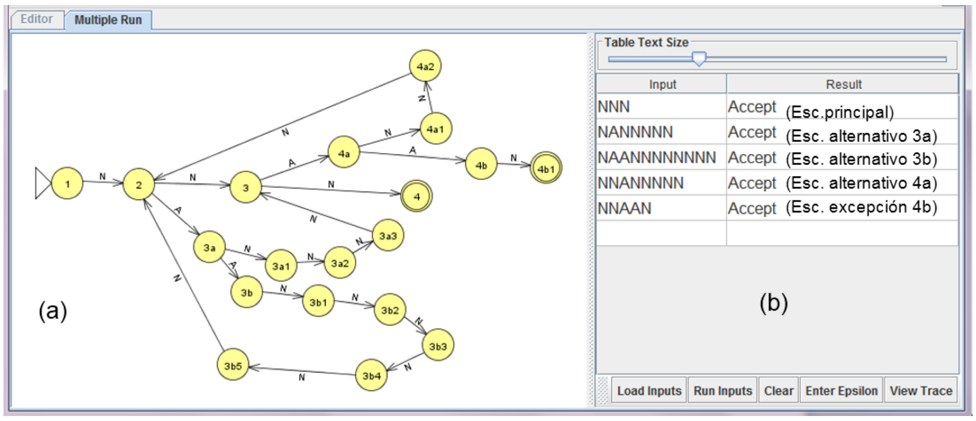
CU-01 = ($Q, \Sigma, \delta, 1, F$), where

$\Sigma = \{N, A\}$,

Q = $\{1, 2, 3, 4, 3a, 3a1, 3a2, 3a3, 3b, 3b1, 3b2, 3b3, 3b4, 3b5, 4a, 4a1, 4a2, 4b, 4b1, TrapState\}$,

F = $\{4, 4b1\}$

δ the transition function that is given by the graph shown in Fig. 4 (a).



Input	Result
NNN	Accept (Esc.principal)
NANNNNN	Accept (Esc. alternativo 3a)
NAANNNNNNNN	Accept (Esc. alternativo 3b)
NNANNNNN	Accept (Esc. alternativo 4a)
NNAAN	Accept (Esc. excepción 4b)

Figure 4. (a) DFA corresponding to CU-01. (b) CU-01 Scenarios simulation

It should be noted that to simplify the visualization, the DFA is not shown completely, obviating the visualization of the trap state and the transitions towards it.

Once the obtained DFA has been loaded in a simulator, it is possible to enter different input strings, which represent the possible flows in the use case in order in order to check whether they are accepted or rejected (thus if they are valid flows for the use case). An input string to be verified will be a string formed by the symbols "N" and "A". If the string representing a flow within the use case is accepted, it means that the corresponding scenario is correctly defined in that use case. On the contrary, if the string is rejected, it means that in some of the flows or scenarios it goes through there is a wrong or inconsistent sequence of steps. This may be because:

- there are unconnected states, paths or steps that were not considered when defining the use case,
- there are scenarios that do not end properly (or remain unfinished), or
- there are infinite loops in the defined scenarios.

Then, the detected inconsistency can be corrected in a new version of the use case, and its corresponding DFA is automatically generated. In this way, use cases can be validated, and traceability links are registered. The side (b) of Fig. 4, shows the result of the simulation that performs the verification of the five possible scenarios defined for the use case CU-01. For the verification of scenarios, the Jflap[5] tool was employed.

4.3 Test cases definition from requirements artifacts using test models

In the process of developing a software or system, use cases represent a means to specify the functional requirements that have to be implemented, in a more detailed and concrete way. Use cases can be also used to drive the generation of the *test cases* [1], [10] that will be executed to verify that the software meets the requirements.

A test case (*TestCase*) specifies the information required for the execution of a test. It comprises the preconditions that should be held before initiating the execution of the corresponding test (*TestPreconditions*), the set of expected inputs and outputs of the test (*TestInput* and *TestExpectedOutput*), the instructions to perform the test (how the inputs are passed to the test object and how the outputs are read from the test object), and the expected post-conditions (*TestExpectedPostconditions*) [1]. The conceptual model presented in Fig. 5 represent theses definitions.

A test case is executed by passing the specified input in the test case to the test object (*TestObject*). Therefore, the execution of a test case comprises a sequence of interactions between the elements in the context of the test (for example, the testers or users) and the object of the test (a system, a component, a function, etc.). When defining a test case, the corresponding interactions can be defined by means of scenarios at different levels of abstraction. Pohl defines in [10] the concept of test case scenario (*TestCaseScenario*) to refer to "types of test cases", which describe sets of test cases that have certain types of inputs and certain types of expected outputs (*TestInputType* and *TestExpectedOutputType*, respectively).

In this paper, the main concepts of the model-based test-case derivation approach proposed by Pohl [1] were included in the conceptual model of the proposed ontology. to allow traceability between the requirements artifacts and the test artifacts. That approach employs use cases and intermediate artifacts (called "test models") to

[5] www.jflap.org

systematically generate the test cases. In the ontology, the concept of "test model" is extended with the DFA artifact type that the authors have proposed on previous research to formalize a use case.

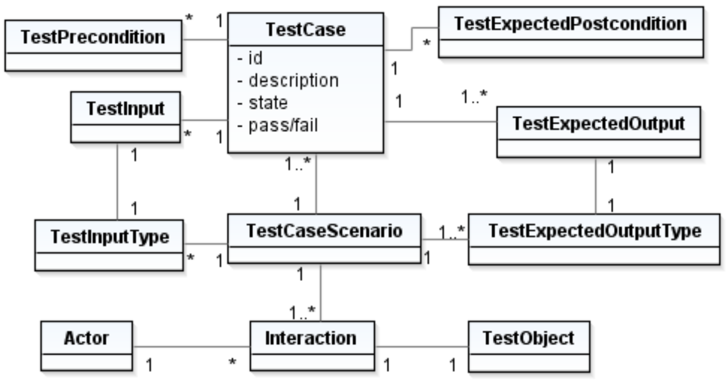


Figure 5: Test Cases Conceptual Model

The proposed ontology includes the necessary elements to represent the test cases derived from the use cases artifacts and maintain the traceability links between the involved artifacts. The traceability links make possible to know, for example, which test cases cover certain flows of a use case, and what is the coverage percentage applied. The conceptual model in Fig. 6 depicts that a *test model* is *developed from* a set of grouped *use case scenarios* (which are part of a use case). Then, on the basis of the test model, which is an intermediate artifact, a set of individual test-case scenarios are identified (*identified-on-the-basis-of* relationship). While the test cases scenarios are defined for certain types of inputs and certain types of expected outputs, the *test cases* have to specify the information required for the test execution. Therefore, the test cases are derived from test-case scenarios, by giving specific values to the inputs and outputs.

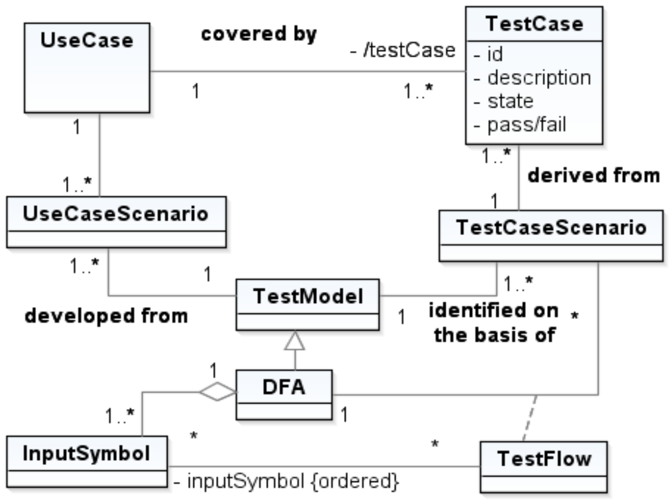
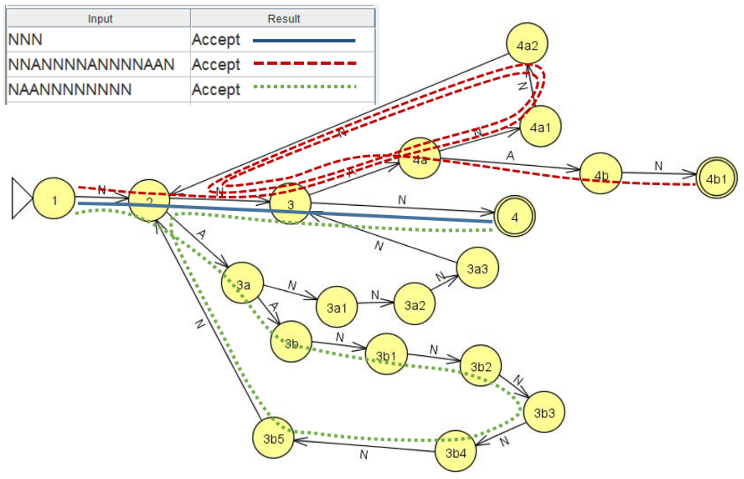


Figure 6: Using DFAs as intermediate test models

In this work, deterministic finite automata are employed as test models to generate the scenarios to test. A DFA models the behavior described by a use case; each path through the DFA is a "test flow" (*TestFlow*) that starts in the automata's initial state and ends in one of its final states. By choosing a coverage criterion, the paths to be tested are selected, and a subset of test-case scenarios is determined. Examples of possible criteria are "Number of tested states / total number of states", "Number of tested transitions / total number of transitions", or "Number of tested exception paths / total number of possible paths". Additionally, the quality of the test can be determined not only by the selected test coverage criterion but also by the percentage to which the criterion should be fulfilled.

An example to illustrate the use of a test-model is given for the use case CU-01. The full definition of CU-01 is provided in the form of a use case template in Fig. 3. By following the transformation rules indicated previously, the corresponding test model is obtained (the DFA in Fig. 4.a). This DFA is employed for guiding the test cases derivation.

Therefore to select the test flows that are going to be covered by the test cases, the criterion "Number of tested states / total number of states" is considered and the coverage percentage set in 84% (16 tested states / 19 total number of states). Fig. 7 shows the test flows that were selected, which are the paths represented by the strings *NNN*, *NNANNNNANNNNAAN*, and *NAANNNNNNNN*. These strings are valid sequences of input symbols that are accepted by the DFA. In Fig. 6, the *TestFlow* concept is equivalent to a sequence of input symbols, which is associated with a test case scenario.



Input	Result
NNN	Accept ▬▬
NNANNNNANNNNAAN	Accept - - -
NAANNNNNNNN	Accept ·····

Figure 7. A DFA that is used as a test-model for deriving a set of test case scenarios. The selected paths depend on the chosen coverage criterion and percentage.

4.4 Traceability Model

Requirements traceability is defined as the ability to describe and follow the life of a requirement in both forward and backward direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration at any of these phases) [22], [23]. The motivations for requirements traceability are diverse. Requirements traceability helps to identify the source of a requirement whether issued by a person, or document, or group of persons. Additionally, requirements traceability allows performing impact analysis, since it is possible to trace what other artifacts are affected due to changes in related requirements. Regarding verification activities, requirements traceability helps in test case verification, thus enabling to keep traceability links between test cases and functional requirements.

Traceability information is often recorded in the form of traceability relationships or links. Between two development artifacts, different types of traceability relationships can exist [23]. Pohl recommends using an information model to define and structure the traceability information and relationship types for a specific project [1]. Within the scope of this research work, a traceability model is proposed (Fig. 8), which include the concepts that the ontology requires for representing traceability knowledge. Fig. 8 introduces, on the one hand, the type of artifacts whose traceability is important to maintain. They are defined as specializations of the concept *TraceableArtifact*. On the other hand, the possible types of traceability relationships that can exist among the artifacts are defined as *TraceabilityRelationship* specializations. The *TraceabilityRelationship* concept is related to the *TraceabilityArtifact* concept by two relationships denoting the source and the destination of a traceability relationship. The cardinality of the relationships can be restricted for specific pair or artifact types.

Pohl proposed five categories of types of traceability relationships: *Condition*, *Content*, *Abstraction*, *Evolution* and *Miscellaneous* [1]. The proposed ontology contemplates these five categories. However, this work is focused mainly on traceability relationships types to representing the "evolution" of artifacts during the development process, from preliminary requirements specifications to test cases artifacts, as well as the intermediate artifacts for test cases derivation. For this reason, just *Evolution* and *Miscellaneous* categories are considered in the conceptual model in Fig. 8.

The types of traceability relationships corresponding to the *Evolution* category enable documenting temporary relationships that can be established between diverse requirements artifacts, and between requirement artifacts and other artifacts of the development process. Within this category, *Formalizes* and *Refines* relationships types are defined. *Formalizes* establishes a traceability relationship between two artifacts *A* and *B*, if *A* constitutes a formal specification of the artifact *B*. For example, this kind of relationship can be maintained between a use case and a Petri network, which formalizes the behavior of the use case [24], or between a use case and a UML activity diagram. It should be noted that both Petri nets and UML activity diagram are artifact types that can be specialized from *TestModel* (although these artifacts are not included in the conceptual models of Fig. 6 and Fig. 8). In addition, the *Replaces* relationship type enables to express that an artifact *A* has been replaced by another artifact *B* during the life cycle of the development process. This situation can occur when an artifact that specifies a requirement (UseCase, Goal, FunctionalRequirement, UseCaseScenario, etc.) is corrected or improved as a consequence of verification & validation requirement activities.

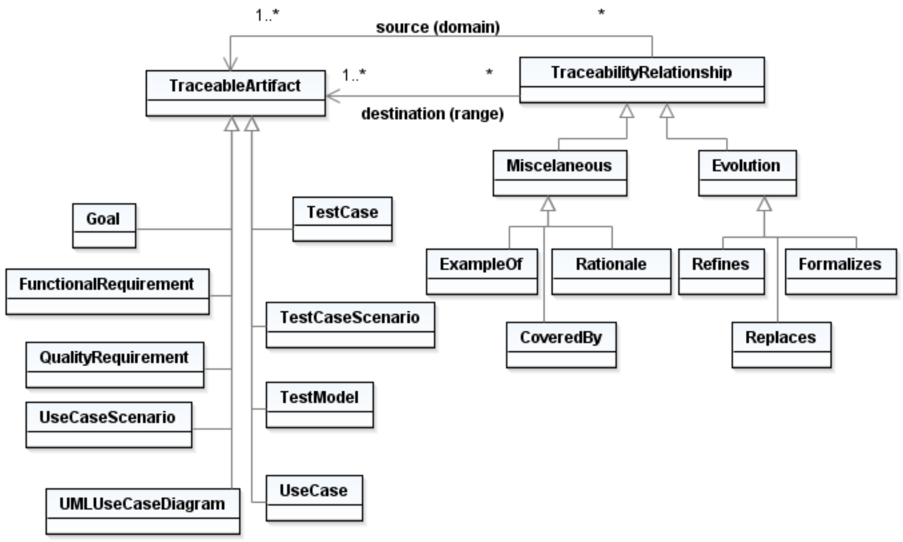


Figure 8: Traceability conceptual model

Moreover, the *Refines* relationship type is established between two artifacts *A* and *B* if *A* describes the artifact *B* in greater detail. An example of this relationship is a use case scenario (*UseCaseScenario* in Fig. 8) that refines certain goal (*Goal* in Fig. 8) since the scenario depicts and illustrates an example of how to achieve the goal. Another example is a template-based specification of a use case (*UseCase* in Fig. 8) that refines certain UML use case diagram (*UMLUseCaseDiagram* in Fig. 8), in the sense that the specification shows further information about use case than the diagram (like the interactions or steps comprised by a use case, which are not present in the diagram).

Miscellaneous traceability relationship class defined by Pohl [1] comprises additional traceability relationship types that can be defined between artifacts. It is specialized in *ExampleOf*, *Rationale*, and *CoveredBy* relationship types. The *ExampleOf* relationship type makes possible to represent that an artifact *A* is an example of another artifact *B*. The *Rationale* relationship type enables representing that one artifact *A* documents the justification of another artifact *B*. For instance, a “rationale” relationship can be used to relate a text fragment to a use case scenario, thereby documenting that the text fragment contains a justification for the existence of the use case scenario.

Another type of *Miscellaneous* traceability relationship is the *CoveredBy* relationship type. It enables linking a test artifact *A* with a requirement artifact *B*, thus stating that *A* is used to verify *B*. For example, when a test case is used to verify the correct implementation of a use case, by covering a certain flow or sequence of steps.

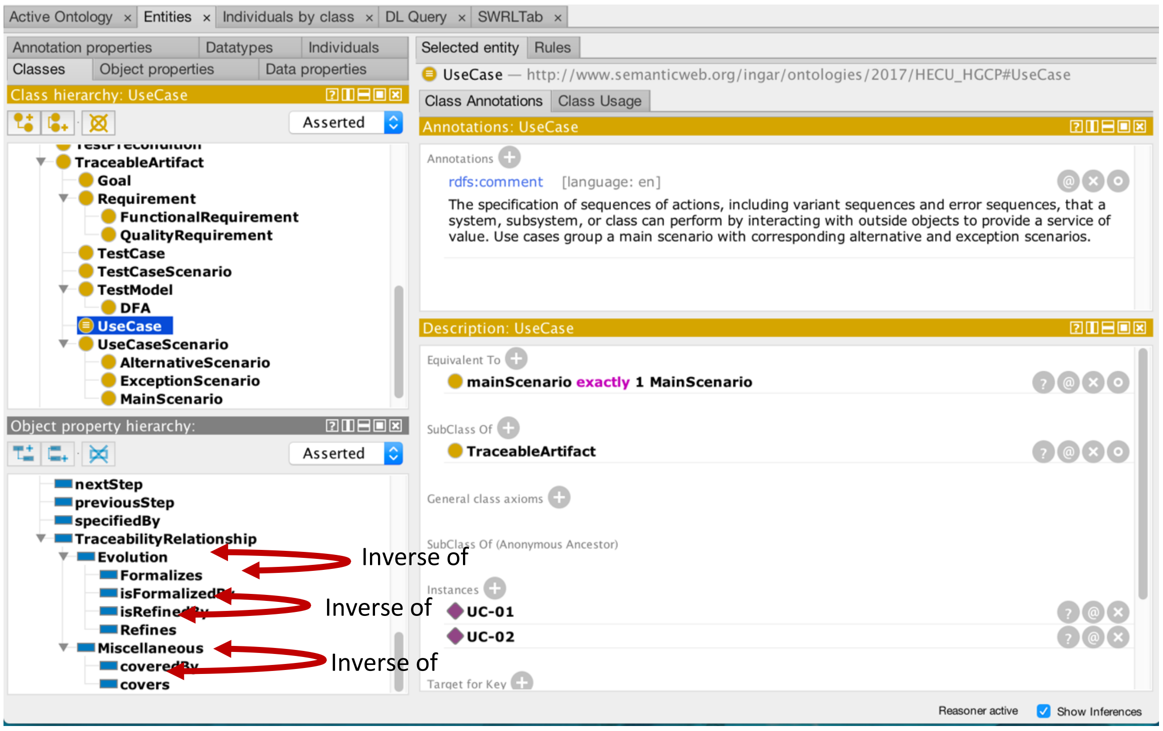
In the proposed ontology, some of the traceability relationships are explicitly defined, whereas others are computed through the definition of inference rules, which are based on the explicit relationships between artifacts types included in the conceptual models of Fig. 1, Fig. 2, Fig. 5 and Fig. 6.

5 Ontology Formalization and Evaluation

The ontology was formalized in OWL 2. The main component of an OWL 2 ontology is a set of axioms, which are statements that say what is true in the domain. OWL 2 axioms can be declarations, axioms about classes, axioms about object or data properties, datatype definitions, keys, assertions (sometimes also called facts), and axioms about annotations.

In particular, by means of axioms OWL 2 allows to express the following different types of properties: (i) object properties, which allow defining relationships between individuals; (ii) datatype properties, which define relationships between individuals and literals (for example, strings, integers, etc.); and (iii) annotation properties, which can be used to describe individuals, classes and properties metadata, such as the language in which a definition is found,

descriptions, and comments on these concepts. In addition, for the object properties that were defined, their *domain* and *range* were specified. For the data type properties, their type, admitted values and cardinality were indicated.
Fig. 9 partially presents the taxonomy of Requirements Engineering concepts that were introduced, which was implemented in the Protégé tool.


Active Ontology × | Entities × | Individuals by class × | DL Query × | SWRLTab ×

Annotation properties | Datatypes | Individuals
Classes | Object properties | Data properties

Class hierarchy: UseCase — Asserted

- TestPrecondition
- TraceableArtifact
  - Goal
  - Requirement
    - FunctionalRequirement
    - QualityRequirement
  - TestCase
  - TestCaseScenario
  - TestModel
    - DFA
  - UseCase
  - UseCaseScenario
    - AlternativeScenario
    - ExceptionScenario
    - MainScenario

Object property hierarchy: — Asserted

- nextStep
- previousStep
- specifiedBy
- TraceabilityRelationship
  - Evolution
    - Formalizes
    - isFormalizedBy
    - isRefinedBy
    - Refines
  - Miscellaneous
    - coveredBy
    - covers

Selected entity | Rules

UseCase — http://www.semanticweb.org/ingar/ontologies/2017/HECU_HGCP#UseCase

Class Annotations | Class Usage

Annotations: UseCase

Annotations

rdfs:comment [language: en]

The specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside objects to provide a service of value. Use cases group a main scenario with corresponding alternative and exception scenarios.

Description: UseCase

Equivalent To
mainScenario exactly 1 MainScenario

SubClass Of
TraceableArtifact

General class axioms

SubClass Of (Anonymous Ancestor)

Instances
UC-01
UC-02

Target for Key

Reasoner active ✓ Show Inferences

Figure 9: Ontology specification in Protégé

Next, certain details of the implementation of ontology are addressed, focusing on the concepts of traceability that are of interest in this work. The *TraceableArtifact* concept (Fig. 8) was implemented as a class (constructor *owl:Class*). As subclasses of *TraceableArtifact*, all the concepts about which it is wanted to keep trace links were defined (*UseCase*, *UseCaseScenario*, *TestCase*, *TestModel*, etc.). On the other hand, the concepts representing the different types of traceability relationships (*covers*, *refines*, *formalizes,* etc.) were implemented as object properties, by using the OWL constructor *owl:ObjectProperty*. It is important to note that, to achieve navigability through the traceability relationships both forward and backward, the corresponding inverse relationships were also defined, as it is indicated in the Object property hierarchy shown in Fig. 9 (*coveredBy*, *isRefinedBy*, *isFormalizedBy*, etc). Moreover, the *domain* and *range* properties were defined to indicate the type of *source* artifact and the type of *destination* artifact expected for the relation.

To complete the implementation of the ontology, and given Protégé's open-world assumption, axioms of disjunction, closure, and coverage were added, so that it is possible to execute reasoners in the ontology. Since OWL assumes that the classes overlap each other unless explicitly stated otherwise, disjunction axioms were incorporated indicating which classes are disjoint.

The ontology was initially populated with knowledge taken from two software development projects, as it is said later. In practice, populating an ontology implies the creation of the individuals (*owl:NamedIndividual*) or instances for the proper classes and setting the adequate object properties between these instances, in order to add the necessary semantics to express the relations among them. The axioms about individuals and their properties are called "facts", "assertions" or "asserted axioms".

Besides, OWL 2 does not make the unique name assumption. That is by having different names two resources are not necessarily different individuals, they could be the same resource with different names. Therefore, in the closed domain in which the ontology is being proposed, when creating the instances, it was necessary to explicitly specify that they were different individuals.

To illustrate how instantiation of concepts was carried out to represent the artifacts that are generated during requirements engineering and testing, some examples are given below.

Fig. 10 presents an objects model that partially represents the different instances that make up the *CU-01* use case artifact, whose template-based specification was presented in Fig. 3. In addition, the corresponding test-model, test-case-scenario and test-case artifacts (named *DFA-01*, *TSc-01* and *TC-01* respectively) are included. To simplify the objects model for the *CU-01* artifact, the instances of *ScenarioStep* are not included, and the instances of *ForkStep* are considered as plain associations between scenarios.

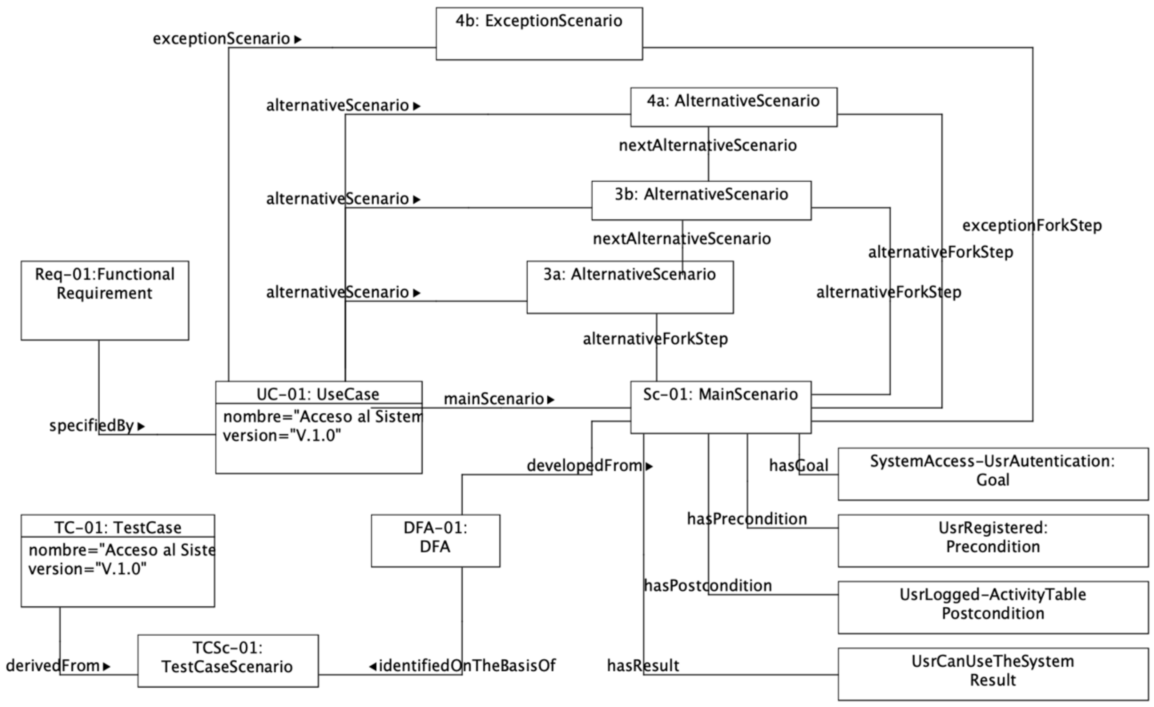


Figure 10: Instances related to *CU-01* artifact (asserted object properties)

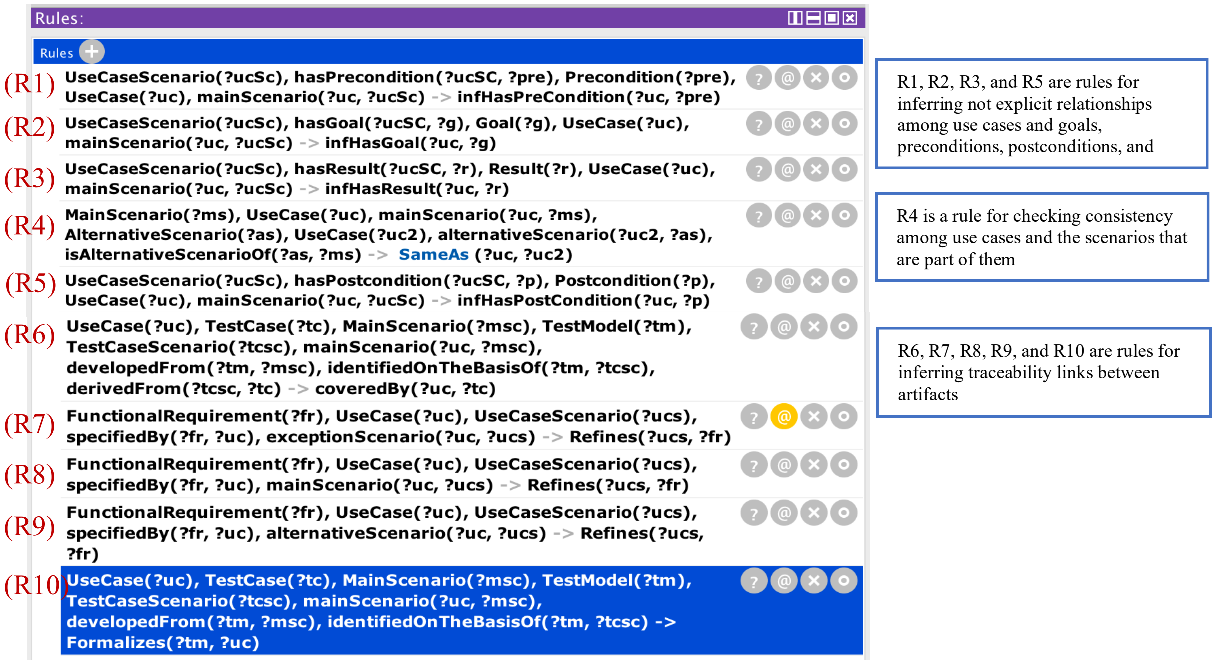
On the other hand, besides asserted axioms that are explicitly present in the ontology, other axioms can be “inferred”. Inferred axioms are those which are obtained by means of a reasoner executed on the ontology axioms. Automated reasoners [25] such as Pellet, FaCT++, HerMiT, ELK and so on take a collection of axioms written in OWL 2 and offer a set of operations on the ontology’s axioms. There exist two main approaches to reasoning: consequence-driven reasoning and tableau-based approaches. Consequence-driven approaches have been shown to be very efficient for so-called Horn fragments of OWL 2 given an ontology O, they apply deduction rules to infer entailments from O and other axioms that have already been inferred. Tableau-based reasoners try to build a model of the ontology by applying completion rules to the individuals in that model – possibly generating new ones – to ensure that all of them (and their relations) satisfy all axioms in the ontology.

To add inference capabilities to trace non-explicit relationships between artifacts, a set of rules was defined in the ontology. They allow the inference of knowledge that is not explicit in the ontology and add checking restrictions to the ontology.

In Fig. 11, some of the rules that were defined in the ontology are shown. Rules *R1*, *R2*, *R3*, and *R5* were specified to infer the relationships that link the *UseCase* instances with the *Goal*, *Precondition*, *Postcondition* and *Result* instances (*infHasGoal*, *infHasPreCondition*, *infHasPostCondition*, and *infHasResult*, respectively). In all cases, the inferred relationships are obtained by navigating through the explicit associations that link a use case with its scenarios, and the scenarios with their objectives, preconditions, postconditions, and results.

Examples of this kind of rules are *R6*, *R7*, *R8*, *R9*, and *R10*. Fig. 12 shows the traceability links that can be inferred applying the specified traceability rules, on the instances of the example in Fig. 10. Fig. 12 also shows two partial views of the Protegé tool, which list the results of the computed inferences (using the Pellet reasoner). Therefore, it should be noted that the traceability relationships instances and their inverses are not explicitly defined in the ontology but are deduced through rules and the execution of a reasoner.

In particular, the section (a) of Fig. 12 shows that the instance representing the requirement *Req1* is refined (*isRefinedBy*, the inverse of the relation *Refines*) by the scenarios *4b*, *SC-01*, *3b*, *4a*, *3a*. Therefore, the ontology permits to answer the question: “What scenarios (main/alternative/exception) were defined to refine a certain preliminary functional requirement?”. This is also possible because rules *R7*, *R8*, and *R9* were specified in the ontology.



Rules:

(R1) UseCaseScenario(?ucSc), hasPrecondition(?ucSC, ?pre), Precondition(?pre), UseCase(?uc), mainScenario(?uc, ?ucSc) -> infHasPreCondition(?uc, ?pre)

(R2) UseCaseScenario(?ucSc), hasGoal(?ucSC, ?g), Goal(?g), UseCase(?uc), mainScenario(?uc, ?ucSc) -> infHasGoal(?uc, ?g)

(R3) UseCaseScenario(?ucSc), hasResult(?ucSC, ?r), Result(?r), UseCase(?uc), mainScenario(?uc, ?ucSc) -> infHasResult(?uc, ?r)

(R4) MainScenario(?ms), UseCase(?uc), mainScenario(?uc, ?ms), AlternativeScenario(?as), UseCase(?uc2), alternativeScenario(?uc2, ?as), isAlternativeScenarioOf(?as, ?ms) -> SameAs (?uc, ?uc2)

(R5) UseCaseScenario(?ucSc), hasPostcondition(?ucSC, ?p), Postcondition(?p), UseCase(?uc), mainScenario(?uc, ?ucSc) -> infHasPostCondition(?uc, ?p)

(R6) UseCase(?uc), TestCase(?tc), MainScenario(?msc), TestModel(?tm), TestCaseScenario(?tcsc), mainScenario(?uc, ?msc), developedFrom(?tm, ?msc), identifiedOnTheBasisOf(?tm, ?tcsc), derivedFrom(?tcsc, ?tc) -> coveredBy(?uc, ?tc)

(R7) FunctionalRequirement(?fr), UseCase(?uc), UseCaseScenario(?ucs), specifiedBy(?fr, ?uc), exceptionScenario(?uc, ?ucs) -> Refines(?ucs, ?fr)

(R8) FunctionalRequirement(?fr), UseCase(?uc), UseCaseScenario(?ucs), specifiedBy(?fr, ?uc), mainScenario(?uc, ?ucs) -> Refines(?ucs, ?fr)

(R9) FunctionalRequirement(?fr), UseCase(?uc), UseCaseScenario(?ucs), specifiedBy(?fr, ?uc), alternativeScenario(?uc, ?ucs) -> Refines(?ucs, ?fr)

(R10) UseCase(?uc), TestCase(?tc), MainScenario(?msc), TestModel(?tm), TestCaseScenario(?tcsc), mainScenario(?uc, ?msc), developedFrom(?tm, ?msc), identifiedOnTheBasisOf(?tm, ?tcsc) -> Formalizes(?tm, ?uc)

R1, R2, R3, and R5 are rules for inferring not explicit relationships among use cases and goals, preconditions, postconditions, and

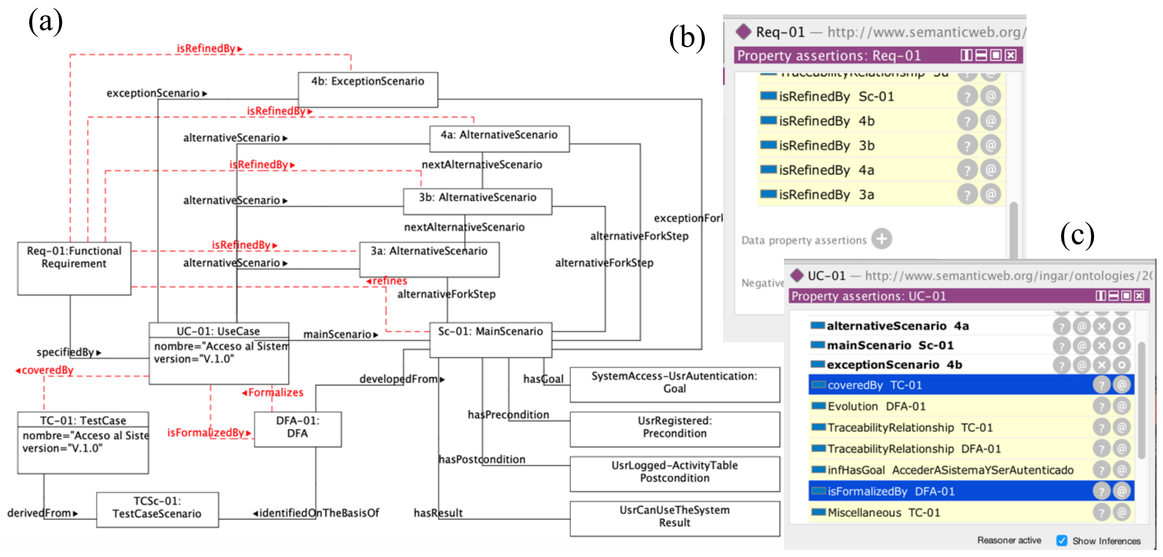
R4 is a rule for checking consistency among use cases and the scenarios that are part of them

R6, R7, R8, R9, and R10 are rules for inferring traceability links between artifacts

Figure 11: Examples of rules defined in the ontology

Also, it is inferred that the use case *CU-01* is related to *TC-01* (an instance of *TestCase*) by means of the traceability relation *coveredBy.* Additionally, it is inferred that the same instance called *CU-01* is related to the instance *DFA-01* (an instance of *DFA* concept) by means of the traceability relation *isFormalizedBy.* The inferred properties axioms are shown in section (c) of Fig. 12, which can be obtained running a reasoner. In this way, the defined rules allow determining: (i) which test cases were defined to cover the possible flows of a use case, and, (ii) by means of which artifacts were formalized the covered flows of the use case.

Going back to the competency questions that were established when the ontology development process was initiated (Table 1), it is possible to validate the ontology analyzing how the represented knowledge enables to answering them. For example, the above example permits to find out "What test cases cover the different paths to be tested on a use case?". On the other hand, it is possible to answer to the question "What test model was generated as an intermediate artifact for the analysis of a use case and the subsequent derivation of test cases?". Rules *6* and *10* plays an important role in obtaining this knowledge.



Req-01 — http://www.semanticweb.org/

Property assertions: Req-01

- isRefinedBy Sc-01
- isRefinedBy 4b
- isRefinedBy 3b
- isRefinedBy 4a
- isRefinedBy 3a

UC-01 — http://www.semanticweb.org/ingar/ontologies/2

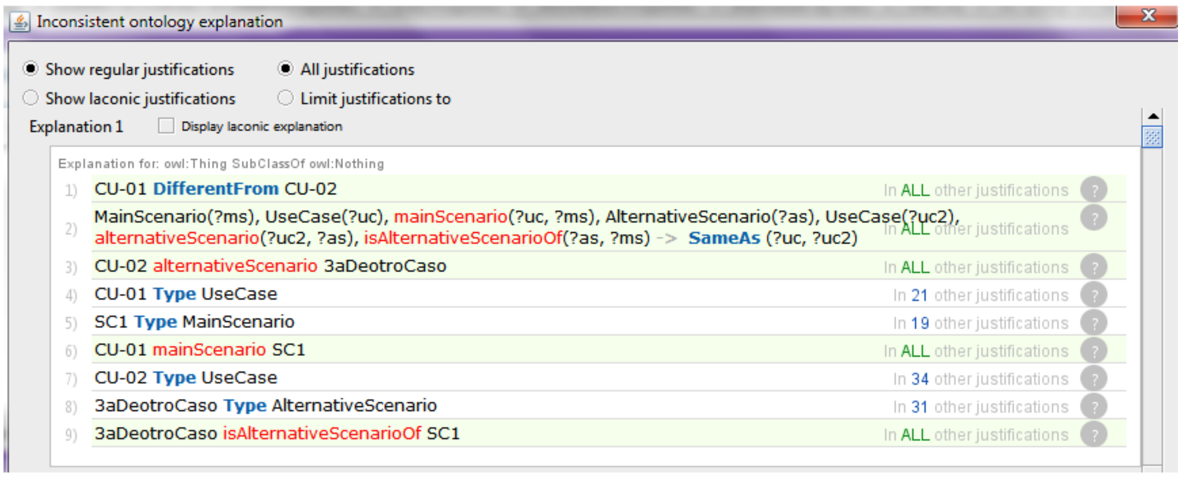
Property assertions: UC-01

- **alternativeScenario 4a**
- **mainScenario Sc-01**
- **exceptionScenario 4b**
- coveredBy TC-01
- Evolution DFA-01
- TraceabilityRelationship TC-01
- TraceabilityRelationship DFA-01
- infHasGoal AccederASistemaYSerAutenticado
- isFormalizedBy DFA-01
- Miscellaneous TC-01

Figure 12: Inferred Object Properties (The Pellet reasoner is active)

In addition to the rules that allow inferring new knowledge, another set of rules incorporates model consistency restrictions. For example, it is possible to define rules to indicate that if a main scenario is associated with certain alternative scenarios, then, all of them are scenarios of the same use case. Similarly, if one scenario step is the next step of another scenario, then, both steps belong to scenarios of the same use case.
The definition of the former consistency rule can be seen in Fig. 11 (rule *R4*). In particular, this rule specifies that if a main scenario *?ms*, associated with a use case *?uc*, has an alternative scenario *?as* and, at the same time, this alternative scenario is associated with another use case *?uc2*, thus *?uc* and *?uc2* are the same individual. However, during the creation and population of the ontology, it was specified that all individuals are different from each other. Therefore SameAs(?uc, ?uc2) consequent makes the ontology inconsistent.

In order to show how this rule allows detecting violations of the constraints represented, an instance of *AlternativeScenario* (named *3adeOtroCaso*) is added to the ontology, which is linked to the main scenario already defined (*SC-01* in Fig. 10) and with a new use case *CU-02*. In addition, a statement that specifies that this new CU-02 individual is different from the individual named CU-01 is added. Due to fact that OWL 2 does not assume the unique name principle, two individuals that have different names not necessarily are different individuals and a reasoner may infer that they are the same entity. So, if two instances are different entities it is necessary to specify explicitly such situation in order to avoid the reasoner infers them as the same individual. This new state of the ontology violates the restriction indicated in the previous paragraph (that a main scenario must be associated with the same use case to which its alternative scenarios are associated). Therefore, new individuals are incorporated into the ontology to represent this situation, then the reasoner is synchronized, and as a result, it detects that the ontology becomes inconsistent after these changes and presents an explanation (Fig. 13).


Inconsistent ontology explanation

[✓ Show regular justifications] [✓ All justifications]
Show laconic justifications   Limit justifications to
Explanation 1   Display laconic explanation

Explanation for: owl:Thing SubClassOf owl:Nothing

1)	CU-01 **DifferentFrom** CU-02	In ALL other justifications
2)	MainScenario(?ms), UseCase(?uc), mainScenario(?uc, ?ms), AlternativeScenario(?as), UseCase(?uc2), alternativeScenario(?uc2, ?as), isAlternativeScenarioOf(?as, ?ms) -> **SameAs** (?uc, ?uc2)	In ALL other justifications
3)	CU-02 alternativeScenario 3aDeotroCaso	In ALL other justifications
4)	CU-01 **Type** UseCase	In 21 other justifications
5)	SC1 **Type** MainScenario	In 19 other justifications
6)	CU-01 mainScenario SC1	In ALL other justifications
7)	CU-02 **Type** UseCase	In 34 other justifications
8)	3aDeotroCaso **Type** AlternativeScenario	In 31 other justifications
9)	3aDeotroCaso isAlternativeScenarioOf SC1	In ALL other justifications

Figure 13: Explanation about a consistency violation of the ontology

Fig. 13 shows nine statements that are found by the reasoner when checking the ontology consistency, which explain the detected inconsistency. These statements are explicitly defined or inferred by the reasoner in the checking process. The statements that are shown in Fig 13 indicates that:

- The individuals labeled as *CU-01* and *CU-02* are both instances of the *UseCase* class (4 and 7).
- *CU-01* and *CU-02* are different individuals (1).
- The individual *SC1* is an instance of *MainScenario* (5) and it is related to *CU-01* through the *mainScenario* relationship (6)
- *3aDeotroCaso* is an instance *AlternativeScenario* class (8) and is linked to *SC1* through the *isAlternativeScneario* association (9) and to *CU-02* through the association labeled as *alternativeScenario* (3)
- *SC1* is the main scenario of *UC-01* use case and it has an alternative scenario labeled as *3aDeotroCaso*, which in turn is an alternative scenario of the *UC-02* use case. So, when applying Rule 4 (R4 in Fig. 11 and statement 2 in Fig. 13), *UC-01* and *UC-02* are inferred as being the same individual, which triggers an inconsistency with the assertion 1.

The proposed approach for artifacts traceability was validated with a proof of concept. Some of the authors have worked on the project for the development of the SIAR tool, aimed to support the specification and consistency checking of requirement artifacts [18]. The authors also have experience in the use of an open source test management software. To prove the feasibility of the proposed ontology for tracing artifacts and making possible to different tools interoperate, they carried out a series of tests. Firstly, a minimal API was developed to access the metamodel of the SIAR tool for requirements specifications. The test management tool provides its own API, which was used to access its repository. On the other hand, a Java component based on Apache Jena[6] was developed to manage the ontology model and manipulate the instances and RDF triples. By using the services of this component, it was possible to create and keep synchronized the instances in the ontology with the artifacts from the different tools, as well as the links among the instances corresponding to the requirements artifacts and test artifacts. The component works together with a reasoner to infer the non-explicit traces and, in this way, to perform queries about related artifacts. Previously, it was necessary to extend some concepts of the ontology to adjust the model to the metamodels of both management tools.

Then, the ontology was initially populated with knowledge about artifacts generated during two projects, which were extracted from the repository of the tool for specifying use cases [19]. This activity was quite straightforward and simple, and a Jena based import software was used. Then, another portion of knowledge was imported into the ontology from the repository of the test management software, which has also been employed for the same two projects [5]. Since the artifacts created came from different databases, it was necessary to link the use case scenarios individuals and the corresponding test models individuals (by means of defining corresponding object properties axioms). This was possible because the related artifacts have the same "id" data type property value that has been derived from an "id" field in the original data sources. This activity was carried out by executing SPARQL[7] insert queries.
The tests were carried out on requirements and test artifacts of small software projects for the developing of two research prototypes, which allowed the authors to obtain preliminary results to conclude that the proposal is valid and of potential application in the software industry.

6 Related Works

Several authors have defined the concepts that intervene in requirements artifacts specifications and their relationships. In [26] a generic meta-model for textual representations of use cases was proposed with the goal of creating an essential and easily extendable meta-model that can be used directly in Model-Driven Engineering activities. It is based on an analysis of 20 studies, that propose templates or meta-models. The more common elements of these studies were represented in a meta-model, together with their more frequent attributes. In [21] the authors proposed requirements templates that can improve requirements elicitation and expression, and two kinds of patterns: linguistic patterns, which are very used sentences in natural language requirements descriptions that can be parameterized and integrated into templates, and requirements patterns, which are generic requirements templates that are found very often during the requirements elicitation process and that can be reused with some adaptation. The conceptual models in these two proposals define specific requirement artifacts and the involved concepts and relationships, which helped us in the definition of the conceptual model for the ontology developed in this work.

A recent work [10] reports the state of the art about strategies for the generation of test cases from requirements artifacts. That paper analyzes the literature on the automatic derivation of test cases, classifying it according to various criteria, like what type of input device is used, whether they use intermediate test models or not, the transformation techniques applied, and the degree of coverage considered. Among the results, there are mentioned some proposals that focus on the traceability between the artifacts and how to implement it. However, their approaches differ from the adopted in this work, which achieves interoperability between use cases specification supporting tools and test case management tools. Neither they are intended to infer traceability relationships between different artifacts of requirements, and between these and artifacts of use cases, nor to detect problems of inconsistency in the specification of artifacts. Having inferred traceability relationships facilitates the maintenance of the traceability links when the involved artifacts change, without the need to eliminate and generate new links. This represents an advantage over approaches that employ traceability matrices and static links between artifacts.

There are several ontology-based approaches for the improvement of Requirements Engineering activities, but few of them have employed ontologies to address interoperability between use case specification tools and test case management tools. A systematic review of the literature on the use of ontologies in requirements engineering, which covers 66 publications between 2007 and 2013 [6], reports that there are few proposals with an ontology-approach that comprise more than one phase of the requirements engineering process. Most are limited to the specification phase, and just a few others include analysis, management, and elicitation phases. Moreover, only 7% of the literature included in the study considers the stages of validation of requirements.

In relation to the types of requirements artifacts that these proposals have addressed, the same study [6] reports that 43% is based on textual requirements (using natural language), being around of 24% the approaches that include

[6] https://jena.apache.org/documentation/ontology/
[7] https://www.w3.org/TR/rdf-sparql-query/

scenarios and use cases within the artifacts of requirements. However, such a review of the literature does not mention works that maintain traceability relationships between requirements artifacts and testing artifacts.

7 Conclusions

In this paper, an ontology-based approach for the specification of functional requirements based on structured use cases and test cases was presented, which allows the representation of the traceability links between these artifacts and the intermediate artifacts that support automatic test cases generation. The proposal has the following advantages:
i) it enables the documentation and specification of several functional requirements types, such as template-based use cases artifacts;
ii) introduces the concept of "test model" as an intermediate artifact based on formal models, which allows the derivation of a set of test cases scenarios, specifying a certain coverage criterion. In particular, a method of transforming use cases into deterministic finite automata was proposed;
iii) achieves interoperability between use cases specification supporting tools and test case management tools, since it is based on a conceptual model that supports the metamodels of these tools;
iv) it allows to infer traceability relationships between different requirements artifacts, and between these and test artifacts, as well as to detect problems of inconsistency in their specification;
v) the possibility of having inferred traceability relationships favors the maintenance of the traceability links when the involved artifacts change, without the need to eliminate and generate new links. This represents an advantage over approaches that employ traceability matrices and static links between artifacts.

Despite numerous advances, the software industry still faces challenges and needs [12] regarding the specification of well-structured and unambiguous requirements, the generation of tests based on requirements artifacts, and the management of traceability links between requirements and test cases.

Future works include extending the ontology with other requirements artifacts, such as UML models (class diagrams, state machine diagrams, and activity diagrams), and other types of traceability links that involve other artifacts of the design process.

Acknowledgments

This work was supported by Universidad Tecnológica Nacional (Proyecto IPN4409 "Herramientas y Métodos de soporte a la Ingeniería de Software: requerimientos, estrategias ágiles, y calidad de procesos y productos") and CONICET.

References

[1] K. Pohl, *Requirements Engineering: Fundamentals, Principles, and Techniques*, Springer, 2010.

[2] C. Allmann, C. Denger, and T. Olsson, "Analysis of Requirements-based Test Case Creation Techniques," IESE-Report No. 046.05/E, 2005.

[3] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergad, *Object Oriented Software Engineering. A Use Case Driven Approach*. Addison Wesley, 1992.

[4] M. Uschold and M. Gruninger, "Ontologies: Principles, Methods and Applications," *Knowledge Engineering Review,* 11(2), 93-136, 1996. Available: https://doi.org/10.1017/S0269888900007797

[5] M. L. Roldán, M. Vegetti, S. Gonnet, M. Marciszak, and H. Leone, "Un Modelo Conceptual para la Especificación y Trazabilidad de Requerimientos Funcionales basados en Casos de Uso y Casos de Prueba," *Congreso Nacional de Ingeniería en Informática/Sistemas de Información (CONAISII 2017)*, Santa Fe, 2017.

[6] D. Dermeval, J. Vilela, I. I. Bittencourt, J. Castro, S. Isotani, P. Brito, and A. Silva, "Applications of ontologies in requirements engineering: a systematic review of the literature," *Requirements Engineering*, Volume 21, Issue 4, pp 405–437, 2016. Available: https://doi.org/10.1007/s00766-015-0222-6

[7] T. Gruber, "A translation approach to portable ontology specifications," Knowledge Acquisition, 5, pp. 199-220, 1993. Available: https://doi.org/10.1006/knac.1993.1008

[8] Institute of Electrical and Electronics Engineers: IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610124990). IEEE, New York, 1990.

[9] O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Proceedings of the First IEEE International Conference on Requirements Engineering (ICRE' 94)*, IEEE Computer Society Press, Los Alamitos, 1994, pp. 94-101.

[10] A. Mustafa, W. M. N. Wan-Kadir, and I. Ibrahim, “Comparative Evaluation of the State-of-art Requirements-based Test Case Generation Approaches,” *International Journal on Advanced Science, Engineering and Information Technology*, vol. 7, no. 4-2, pp. 1567-1573, 2017. Available: http://dx.doi.org/10.18517/ijaseit.7.4-2.3409

[11] S. Hesari, R. Behjati, R., and T. Yue, “Towards a systematic requirement-based test generation framework: Industrial challenges and needs,” *21st IEEE International Requirements Engineering Conference (RE)*, Rio de Janeiro, 2013, pp. 261-266. Available: https://doi.org/10.1109/RE.2013.6636727

[12] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, “One evaluation of model-based testing and its automation,” *Proceedings of the 27th international conference on Software engineering (ICSE '05)*, ACM, NY, USA, 2005, pp. 392-401. Available: https://doi.org/10.1145/1062455.1062529

[13] A. Reuys, S. Reis, E. Kamsties, K. Pohl, “The ScenTED Method for Testing Software Product Lines,” *Software Product Lines*, Springer Berlin Heidelberg, pp. 479-520, 2006.

[14] M. Gruninger and M. S. Fox, “Methodology for the Design and Evaluation of Ontologies,” *IJCAI Workshop on Basic Ontological in Knowledge Sharing, Montreal*, Canada, 1995.

[15] M. Fernández-López, A. Gómez-Pérez, J. P. Sierra, and A.P, Sierra, “Building a chemical ontology using methontology and the ontology design environment.” *IEEE Intelligent Systems and their Applications*, 14, 37–46, 1999. Available: https://doi.org/10.1109/5254.747904

[16] M. C. Suárez-Figueroa, A. Gómez-Pérez, E. Motta, E., and A. Gangeni, *Ontology Engineering in a Networked World*, Springer, Berlin, 2012. Available: https://doi.org/10.1007/978-3-642-24794-1

[17] K. E. Wiegers, *Software Requirements*, 3rd Edition, Microsoft Press, 2013.

[18] M. M. Marciszack, M. Perez Cota and M. A. Groppo, “Metodología y herramienta de soporte para validar modelos conceptuales a través de máquinas abstractas,” *Ciencia y Tecnología*, Universidad de Palermo, 15, pp. 165-180, 2015.

[19] J. Hopcroft, R. Motwani, J. Ullman, *Introducción a la teoría de autómatas, lenguajes y computación*, 3 ed., Pearson, 2007.

[20] A. Cockburn, *Writing Effective Use Cases (1st ed.)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[21] A. Durán Toro, B. Bernárdez Jiménez, A. Ruiz Cortés and M.A. Toro Bonilla, “Requirements Elicitation Approach Based in Templates and Patterns,” *Workshop em Engenharia de Requisitos* (WER 1999), Buenos Aires, 1999.

[22] O. C. Z. Gotel and A. C. W. Finkelstein, “An Analysis of the Requirements Traceability Problem,” Technical Report TR-93- 41, Department of Computing, Imperial College, 1993.

[23] B. Ramesh and M. Jarke, “Toward Reference Models for Requirements Traceability,” *IEEE Transactions On Software Engineering*, Vol. 27, No. 1, 2001. Available: https://doi.org/10.1109/32.895989

[24] J. Zhao and Z Duan, Verification of Use Case with Petri Nets in Requirement Analysis.” *Lecture Notes in Computer Science*, vol 5593. Springer, Berlin, Heidelberg, 2009. Available: https://doi.org/10.1007/978-3-642-02457-3_3

[25] B. Parsia, N. Matentzoglu, R.S. Gonçalves, B. Glimm, and A. Steigmille, “The OWL Reasoner Evaluation (ORE) 2015 Competition Report,” *Journal of Automated Reasoning*, Vol. 59 (4), pp 455–482, https://doi.org/10.1007/s10817-017-9406

[26] F. L. Siqueira and P. S. Muniz Silva, “An Essential Textual Use Case Meta-model Based on an Analysis of Existing Proposals,” *Workshop em Engenharia de Requisitos* (WER 2011), pp. 419-430, 2011.