Using ALF within the CoSTest process for Validation of UML-based Conceptual Schema

Maria Fernanda Granda[1, 4], Nelly Condori-Fernández[2,3], Tanja E. J. Vos[4]

[1] University of Cuenca, Computer Science Department, Cuenca, Ecuador
fernanda.granda@ucuenca.edu.ec
[2] Vrije Universiteit van Amsterdam, Amsterdam, The Netherlands
n.condori-fernandez@vu.nl
[3] University of A Coruña, Coruña, Spain
n.condori.fernandez@udc.es
[4] Universitat Politècnica de València, PROS Research Centre, Valencia, Spain
{fgranda, tvos}@pros.upv.es

Abstract. The Unified Modelling Language (UML) is widely used for modelling software systems and its integration with executable languages, such as the Action Language for Foundational UML (ALF), provides a bridge between the graphical specification techniques used by mainstream software engineers and the precise analysis and validation techniques essential for the model-driven development of information systems. As far as we know, the idea of transforming Conceptual Schemas (CS) based on UML Class Diagrams into ALF to execute systematic ALF-based test cases against these CSs and to report defects by checking logs has not been explored to date. In this paper, we use ALF to create a testing environment to validate requirements and verify some system properties at the CS level. We also report on some of the implementation details and design decisions of our proof-of-concept tool, as well as its limitations and possible use scenarios.

Keywords: UML to ALF · conceptual schema validation · model validation · UML class diagram · CoSTest tool

1 Introduction

In previous work we proposed an approach for testing-based validation of Conceptual Schemas (CS) in a Model-driven environment [1], in which a group of engineers (e.g. requirements engineers) specifies requirements models from which the test scenarios are automatically generated with abstract test cases (i.e. a concrete story of a user-system interaction and the expected result). These test cases are then used to validate the requirements in an early phase of software analysis and design (e.g. CS). However, in order to execute the test cases systematically and automatically against conceptual schemas, they must be translated into an executable language. In this context, the Unified Modelling Language (UML) has been widely used to draw models for analysing, designing and documenting software that can then be written/transformed

In: C. Cabanillas, S. España, S. Farshidi (eds.):
Proceedings of the ER Forum 2017 and the ER 2017 Demo track,
Valencia, Spain, November 6th-9th, 2017,
published at http://ceur-ws.org

into any programming language. One of the crucial issues when creating precise, standard UML models was the imprecision of semantics specified in the UML standard. This issue was finally addressed with the adoption by the Object Management Group (OMG) in 2008 of the Foundational UML (fUML) specification (an executable subset of UML) and the Action Language for fUML (ALF) adopted in 2010. These standards allow the UML model to be represented both graphically and textually (while preserving its semantic level) [2], e.g. with the Eclipse-based open-source UML modelling tool Papyrus[1]. Papyrus provides the ability to execute fUML models, thanks to its model execution platform Moka[2], which makes it possible to interact with an execution and analyse the manipulated values. However, this tool is limited by not having access to ALF source-level debugging, which would make it a lot easier to test complex behaviour [2] as well as execute several test cases and analyse the logs in a systematic testing process. Since the open-source ALF Reference Implementation[3] is distributed without a graphical tool, it allows executable models to be written completely textually in ALF. This opens up the possibility of using ALF to automatically and systematically execute a set of test cases against CS and analyse their execution trace in order to detect defects at the conceptual schema level. For this solution to be viable a transformation from UML to ALF is also required.

The paper describes the automatic transformation of a UML class diagram (CD) into ALF language in the context of the CoSTest tool[4] for the systematic testing of conceptual schemas. The resulting translated model is semantically equivalent to the original, meaning that the contract semantics (i.e. pre and post conditions), derived features, operation bodies, and association class are implemented as elements of the ALF units. The paper's contributions are: (1) Translating UML into ALF; (2) Using ALF as a language for writing/executing test cases. We also evaluated these transformations by using our freely available CoSTest validation tool with eight CS.

The paper is structured as follows: Section 2 introduces a simple but representative example of the UML CD-based CS used. Section 3 summarizes the background. Section 4 reviews related work. Section 5 describes the mapping rules between UML and ALF notation by describing its application to the example. Section 6 gives an overview of the last phase of CoSTest to generate ALF based-test cases, as well as execution details of our validation approach. Section 7 demonstrates the application of the transformations to ALF in eight CS. Section 8 discusses the design decisions, limitations and alternative applications of the approach. The conclusions and future work are outlined in Section 9.

2 Motivating Example

To show how ALF supports the validation in the CoSTest tool, we will use a simple model of an order from the domain of e-commerce. The first thing to decide is the

[1] https://eclipse.org/papyrus/
[2] http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
[3] http://modeldriven.github.io/Alf-Reference-Implementation/
[4] https://staq.dsic.upv.es/webstaq/costest.html

information that needs to be kept on an order and how this is related to information on the customer placing the order. This can be clearly represented by using a UML class diagram, such as the one shown in Figure 1 with part of an Order CS using an UML class diagram. This diagram was entered graphically using the Papyrus tool and it models an order as recording the date it was received, prepayment and dispatch status, and has a set of order lines, each of which specifies the quantity of a certain product included in the order. It also shows that an order is placed by a single customer (i.e. corporate or personal customer), who may make many orders over time. Each of these orders has several order lines, each of which refers to a single product, with net price and quantity available. Each customer has a name and address. A personal customer also has a credit card number and a corporate customer has a contact name, credit rating and credit limit. The products are identified by a name in a language, a description and a url.

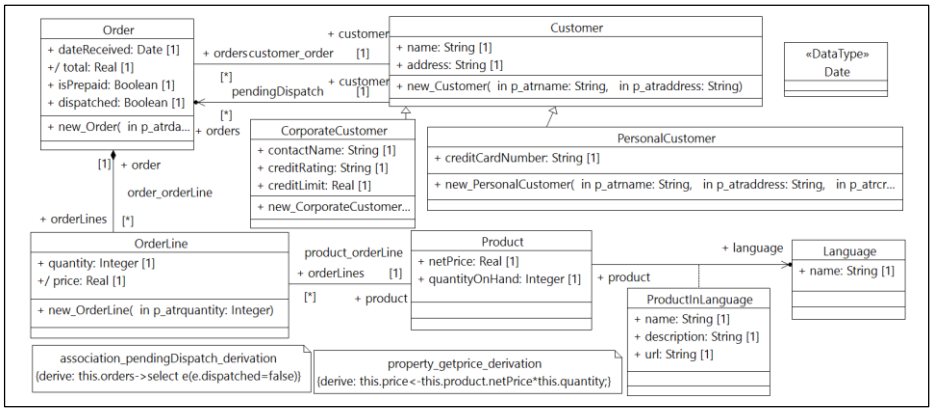


Fig. 1. A partial view of order example CS

In addition, there is also behaviour associated with the classes shown in the diagram. Suppose, for instance, that you would like to compute the total of the Order, along with a new_OrderLine operation that adds a new line item for a given Product and updates the total appropriately. To do this, the newOrderLine will use a new property_getPrice_derivation operation on the OrderLine, which will be derived from the related constraint. Finally, a derived association pendingDispatch calculates the customer orders that are pending dispatch.

3 Background

3.1 Executable Conceptual Schema using ALF

Since the executable CS of a system should describe its structure and behaviour. A class diagram is the UML's main building block and shows structural elements of the system at an abstract level (e.g. class, association class), their properties (owned attribute), relationships (e.g. association and generalization) and operations [3]. On the other hand, the behavioural part, is specified by characterizing how event execution can come

about (unfold). In UML, this is done by collaboration, sequence, activity and state chart models, as well as the textual specification of methods.

In this context, an executable model is at the next higher layer of abstraction, abstracting away both specific programming languages and decisions about the organization of the software (e.g. data structure and partitioning) so that a specification built in Executable UML can be deployed in various software environments without change [4]. A key ingredient of any Executable UML variant is the use of an Action language (type of pseudocode) that allows designers to completely specify fine-grained behavioural aspects of the model (e.g. to define the behaviour of a method of a class). ALF is a platform-independent language that works at the same semantic level as the rest of the UML-based CS. This means that actions allow directly a manipulation of the elements of the conceptual schema (no assumptions are made about middleware, implementation language or software design policy) and they are capable of being translated into different implementations for different platforms and languages.

In this paper we propose to use a UML-based Class diagram and derive part of the CS behaviour from the constructor operations, constraints, invariants, pre and post conditions represented in the class diagram by translating them into ALF code as part of the specification of a method (i.e. constraints and invariants) or a conditional inside of the method specification (i.e. pre and post conditions).

3.2 CoSTest process for validating Conceptual Schemas

We developed an early testing technique supported by the CoSTest tool to validate requirements at model level [5]. Figure 2 provides an overview of how CoSTest operates by covering three main phases: (i) test suite generation, (ii) CS under test generation, (iii) test execution and report generation with the faults detected and the coverage analysis. The red frame corresponds to the work presented in this paper.

i) Test Suite Generation. This phase supports the semi-automatic generation of test cases using a model-driven process. The first two steps of this process are explained in detail in a previous work [1]: (1) transform a Requirements Model (i.e. system requirements at business level) into Test Model (i.e. contains information about the test items and their order of precedence); (2) transform the Test Model into Test Scenario Model identifying the different sequences of events (i.e. test paths) from test model; (3) generate the test values for test cases; (4) transform each test scenario into Test case scripts (ALF script), which contains the abstract test cases; (5) select the type of test cases (e.g. positive test cases only or including negative test cases); (6) generate concrete and executable test cases into ALF textual specifications; and (7) prioritize and select the test cases for execution based on mutation testing [6]. Steps 5-6 are explained in greater detail in Section 6. Figure 2 shows some artefacts used in our Order example.

ii) Generation of the Executable Conceptual Schema under test. In this phase the executable CS using ALF is generated (see Step 8 in Figure 2) using the UML-to-ALF transformation described in Section 5. Then (Step 9), we can parse the CS before starting the execution of CS in the testing process (see Section 5).

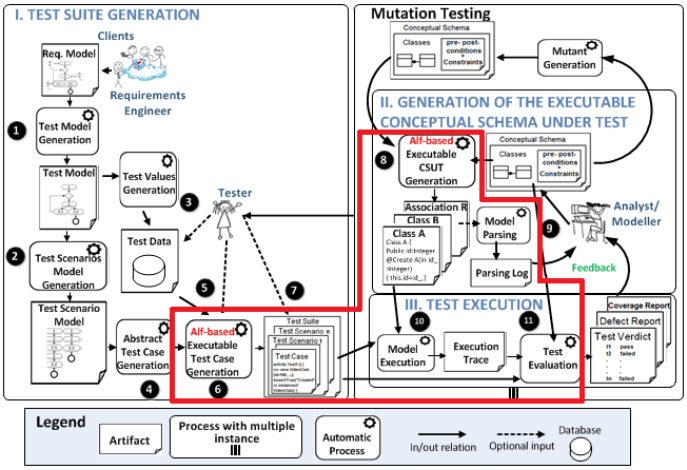
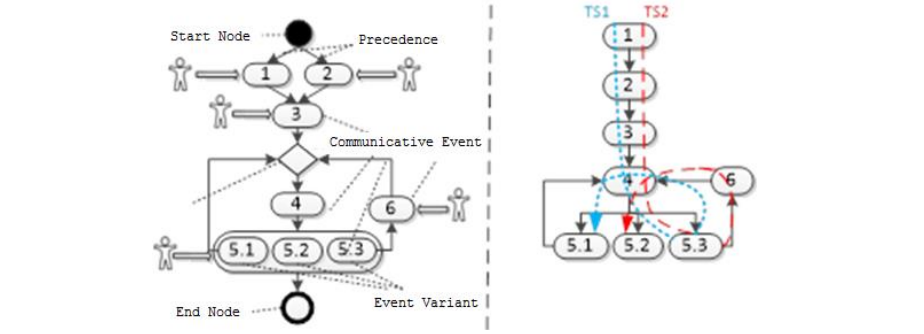


Fig. 2. Overview of the CoSTest process for CS validation




```
private import Order::*;
public import Alf::Library::BasicTypes::*;
public import Alf::Library::Asserts::*;
activity AbsTScenario_1_Order () {
language_= new Language(p_atrname="English");
customer_=new Customer(p_atrname="John Papiro", p_atraddress="Gran Via 120")
AsserTrue("Object created", customer_ instanceof Customer);
}
```

c) test case to validate the object creation

Fig. 3. Example of artefacts used in the test case generation for Order example

iii) Test Execution. In this phase the test cases are executed against the CS and a list of defects and a coverage analysis are reported (see Steps 10 and 11 in Figure 2). Further information is given in Section 6.

4 Related Work

Although there are a number of studies addressing the verification of UML models that include actions [7] [8] [9] [10], only some of them [11] [12] [13] are aligned with the ALF action language standard.

The Papyrus tool [14][2], an open-source UML tool under the Eclipse Modelling Project uses ALF to validate UML models. This tool has executable modelling capabilities including: (1) creating a complete program as a graphical UML class model, with detailed behavioural code written textually using ALF; (2) synchronizing the graphical representation of a UML class with its textual representation in ALF; (3) concurrent execution of an activity and (4) debugging an executing activity. This means a user (modeller/analyst/tester) can manually enter the tests as an activity diagram to perform the testing and debugging process. There is also a work [14] that provides feedback and lessons learned by the Papyrus team regarding the implementation and use of the fUML with ALF from the perspective of domain-specific users. Research has also been carried out [15][16] on using fUML and ALF as the basis for specifying the semantics of domain-specific modelling languages. However, to authors' knowledge, there is no possibility of automatically obtaining a full version of the UML model in ALF code from these tools.

This paper describes the use of ALF for generating/writing executable test cases as well as for transforming a UML CD-based CS in an executable model. These ALF-based artefacts are then used within the CoSTest process for validation of UML-based Conceptual Schemas by executing the test cases against the executable CS in an ALF-based testing environment. We also report on the usefulness of our UML-to-ALF transformation and its parsing to validate a set of mutation operators [17] and a mutation tool [18] implemented with the intention of evaluating the effectiveness of the test cases generated by CoSTest [6] and to prioritize them.

5 Generation of Executable Conceptual Schema using ALF

We use the ALF language as a notation for representing UML CD-based CS and for reasoning about this model. To obtain the result outlined in the previous section we defined a model-to-text transformation of UML to ALF, which we describe in this section. The mapping is specified as an Acceleo[5] transformation included in CoSTest and we outline here its points of interest. Table 1 provides information about the main transformations according to the ALF standard.

For derived associations, we add an attribute to the class (e.g. sequence) and create a getter operation (e.g., association_<DerivedAssociationName>_derivation). We then attach the operation generated from the constraint expression to the getter. Figure 4 shows highlighted with red rectangles the ALF code for the pendingDispatch derived association of our example. All the examples given in this section have been translated into ALF using our ALF translator implemented in CoSTest, and executed using the fUML execution engine.

We decided to use the Reference Implementation[6] as an fUML engine because (1) it is based on the reference implementation and (2) it provides an execution log. Thanks to (1) we have confidence in its conformity to the fUML specification. And (2) means

[5] https://www.eclipse.org/acceleo/

[6] http://modeldriven.github.io/fUML-Reference-Implementation/

that systematic testing (i.e. reviewing hundreds of logs) is simpler than with the Moka[7] implementation, which is more suitable for an interactive testing.

Table 1. An overview of the mappings between UML element and ALF code

UML element	Example of concrete syntax in ALF
Package is translated into an ALF package with the classes and associations that conform it.	**package** <CS name> { public class <class name> [specializes <class name>]; … public assoc <association name>; … }
Class is translated into ALF unit with attributes, operations, parameters and the constructor method for creating new object instances.	namespace <CS name> { **class** <class name> [**specializes** <class parent name>]; public <attribute name>:data type; …
Inheritance poses a particular problem in translating UML to ALF, since a subclass is dependent on its superclass, and this is an operation dependence since creation of a subclass instance requires invocation of its superclass constructor (i.e. super). The inheritance relations are translated into ALF by using the *specializes* clause.	@Create <class name> (in <attribute name: data type>, …) { **super**.<class parent name> (parameter1, …); <method statements> } … // specification of methods public <**operation name**> (in <parameter name: data type>, …) { <method statements> } }

Association is translated into ALF unit, which creates a new link (i.e. an instance of an association) in the association with end values object1 (with role1) and object2 (with role2).	namespace <CS name> {
	assoc <association name> {
	public 'role1': Object1[1];
	public 'role2': Object2 [*]; }

Aggregation: In order to guarantee that the particular semantics of composition is preserved (transitive propagation of properties, lifetime dependence between the related entities, etc), the aggregation is translated using the *compose* clause	namespace <CS name> {
	assoc <association name> {
	public 'role1': Object1[1];
	public 'role2': **compose** Object2;
	}

Constraints are included in the UML models using mechanisms such as body, pre and post conditions. These mechanisms need to be translated into ALF elements (i.e. operation method, method conditional) to be executable.	public <**operation name**> (in <parameter name: data type>, …) { **if** <condition> { <statements>} **else** { <statements>} … //method statements }

The transformation of UML CD-based CS into ALF is performed in two steps:

1. **Model-to-text transformation** translates the UML CD-based CS into ALF units. This transformation is written in Acceleo code. It takes as inputs an UML CD-based CS, and gives as output an ALF -based CS. The resulting ALF-based CS contains the elements generated from the transformation of all CS elements given as input.

[7] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

```
class Customer{
 public address:String;
 public name:String;
 public pendingDispach: Order [] sequence;
 @Create Customer ( in p_atrname:String, in p_atraddress:String)
 // Corresponding to new_customer
 {    this.name=p_atrname;
      this.address=p_atraddress;   }
 public association_pendingDispach_derivation(){
     this.pendingDispach= this.orders->select e(e.dispached=false);
 }
```

Fig. 4. Example of a derived association translated to ALF of the Order example

2. **ALF unit parsing**. Semantically, ALF maps the CS to the Foundational UML (fUML [19]) subset. The resulting ALF-base CS is semantically equivalent to the original one. Then fUML provides the virtual machine for the execution of the ALF units. Further details can be found in the ALF Reference Implementation.

The current version of our ALF transformation supports most UML CD constructs with the following notable exceptions: (1) features required to specify abstractions could be added with relatively little work; (2) transformation of OCL constraints. Currently, the UML CD-based CSs used in our approach use directly the ALF language to specify the constraints. But, there is an approach enabling OCL and fUML Integration by transformation that could be used to address this issue [20].

We applied our ALF -based tool to check the generation of CoSTest test cases and measure their effectiveness in several subject CS [6]. In addition, we conducted research [18][17] into using UML-to-ALF transformation as the basis for specifying valid mutation operators and parsing mutants in the context of our validation tool for UML CD-based CS.

6 Generation and Execution of Test Cases using ALF

The last step executed in CoSTest to generate test scenarios (see Section 3.2) is the transformation into ALF code of the abstract test cases (e.g. services, triggers, assertions and links exemplifying the interaction of actors with the system) using Acceleo, which are concretized with values entered by the tester or generated from the data model. The result of this transformation is a test suite with concrete and executable ALF-based test cases.

In ALF, an executable test case is an activity that provides the specification of parameterized behaviour as the coordinated sequencing of subordinate ALF units including assertions which describe how the model should behave. A CS can demonstrate semantic correctness and completeness with its requirements specification if no assertion is violated when executing the test suite.

Correctness covers both syntactic correctness (right or well-formedness syntax) and semantic correctness (right meaning and relations relative to the knowledge about the domain) [21]. Completeness is to have all the necessary information according to the purpose of modelling [21]. Incompleteness of the UML CD-based CS can also be detected via testing of the translated ALF, enabling particular test scenarios to be executed. For example, in the Order CS specification the lack of an invariant to ensure

that creditCardNumber of a PersonalCustomer class is unique can be identified by instantiating two PersonalCustomer with the same credit card number, if the scenario is executed without reporting an error then the invariant is required, otherwise the CS contains the invariant (the required invariant is context PersonalCustomer inv UniqueCreditCard: body: PersonalCustomer->isUnique e(e.creditCardNumer)).

The semantic correctness of UML-CD-based CS with respect to a domain can also be checked against the evaluation of constraints. Then the elements exercised in the test cases can be used to identify other defects, such as redundant (e.g. duplicated element) and extraneous (e.g. CS elements not used by test cases) elements.

A test suite generated by CoSTest contains hundreds of test cases, so that this automatic generation could not have been produced without ALF.

CoSTest generates three kinds of test cases and focuses on instantiation semantics of UML Structures to test:

1. **The occurrence of events**. An event is the execution of some operation (method) of the CS, which may have several kinds of defects, of which the following can be highlighted: (a) the pre-conditions of the event may not allow the occurrence of valid events. (b) the post-conditions may not precisely define the intended effect of events. (c) the method of an operation may produce a state that does not satisfy the CS invariants.

```
AssertTrue (“Object created”, order_ instanceof Order);
AssertTrue("Association Created", order_.orderlines->size()>0);
```

2. **The non-occurrence of events**. For the set of constraints defined in the CS to be correct and complete, not only must the constraints be satisfied by valid CS states, but those constraints must also rule out invalid states. Testing the CS may be a practical mean of detecting missing constraints. This is done by setting up one or more test cases, with a state established in the requirements as invalid, such as values of the range, minimum cardinality violation, unique value violation for class attributes, followed by an assertion that the state does not satisfy at least one CS constraint. For example, when an attribute of an object should be unique (e.g. unique name for Customer) CoSTest generates a test case including two instances of this object (customer in our example) with the same value in the unique attribute (i.e. name) and test the required constraint with an assertion as follows:

```
customer_=new Customer (p_atrname=”name”, p_atraddress=”Address1”);
customer2_=new Customer (p_atrname=”name”, p_atraddress=”Address2”);
AssertFalse (“Not Exist”, customer_ instanceof Customer);
```

3. **The contents of CS objects.** It is often useful to include an assertion on the current state of an object instance in the CS in a test case. The purpose may be to check that one or more derivation rules derive the expected results, or that a navigational expression yields the expected results or that the effect of one or more events implies an expected result in the CS object instance. For example, CoSTest generates the following two assertions to test the derivation rules corresponding to the total derived attribute (in Order class) and the pendingDispatch derived association of our example (see Figure 1).

```
AssertEqual("property_total_derivation", order_.total, order_.
orderLines->collect e(e.price)->reduce Sum);
AssertEqual ("association_pendingDispatch_derivation", customer_.
pendingDispatch, customer_.orders->select e(e.dispatched=false));
```

The oracle and test goal of each test case is derived from the type of test cases selected (i.e. positive or negative). The expected value (oracle) for the positive test cases (assertionEqual or assertionTrue) is "true" and with negative conditions the False assertion (assertionFalse) must be true, otherwise the test case fails. The test cases are then evaluated by using these oracles and goals included in the test cases. A test case returns the verdict Pass (if the assertion is satisfied), Fail (if the assertion is not satisfied) or Inconclusive (if it was not possible to execute all the statements previous to the assertion). When the verdict of the assertion is Fail or Inconclusive, the execution trace (i.e. ALF execution log) is analysed to report the defects by using the information shown in Table 2.

Table 2. Relationship between fault and defect reported by CoSTest

Fault reported by execution of the ALF code	Defect Reported by CoSTest
propertyAccessExpressionFeatureResolution	Missing or private Association
instanceCreationExpressionConstructor	Missing Class (or private)
behaviorInvocationExpressionReferentConstraint	Missing Operation (or private)
propertyAccessExpressionFeatureResolution	Incorrect Association
linkOperationExpressionArgumentCompatibility	Incorrect Association Ends
instanceCreationExpressionConstructorlessLegality	Incorrect Constructor
assignmentExpressionSimpleAssignmentTypeConformance	Incorrect Parameter Data Type
tupleNullInput in a createlink statement	Incorrect null Value in Association
tupleNullInput in an operation statement	Incorrect null Value in Parameter
instanceCreationExpressionDataTypeCompatibility	Incorrect Operation Signature
behaviorInvocationExpressionArgumentCompatibility	Incorrect Parameter Data Type
superInvocationExpressionOperation	Incorrect Super Class

The information included in Table 2 was obtained by means of analysis of the faults reported in the logs and the defects injected in the CS.

7 Application of ALF within the CoSTest process

In order to evaluate the syntactic correctness and completeness of the transformation rules to ALF used in CoSTest (i.e. for CSUT and executable test cases), we applied our UML-to-ALF transformation (see steps 8 and 9 in Figure 2) and our generator of ALF-based test cases (see step 6 in Figure 2) to eight CSs. In particular, this experiment took as input CSs containing a variety of characteristics that can be present in UML CD-based CS, including classes, relations (i.e. association, composite aggregation, and generalization) and different types of constraints (i.e. pre-condition, post-condition and body condition). These CSs were of different sizes and domains (e.g. information

systems, games). On case is taken from industrial (i.e. IM), others CSs were found in the literature (i.e. [22], [23], [24], [25] and [26]). The different CSs were specified using UML2 and Papyrus[8] tools. Table 3 shows the number of the ALF-based test cases and test scenarios generated for different CSs by CoSTest. Test suites used in this study include tests checking all the CS class operations and constraints. Table 4 summarizes the characteristics (i.e. UML class diagram elements) of these CSs.

This experiment let us to evaluate the transformation rules of the CoSTest by verifying the syntactic correctness and evaluating the completeness of the transformed CSUT. Then, the results obtained by parsing for these transformations to ALF (i.e. CSUT and test cases) were 100% well-formed and complete. These CSs are publicly available in the project website https://staq.dsic.upv.es/webstaq/costest.html, so that, the test cases can be again generated and the experiment can be replicated with the CoSTest tool.

Table 3. Details of the ALF-test suites generated by CoSTEst for Subject CS

Conceptual Schema	# Test Scenarios	# Test Cases
Video Club system (VC)	1	36
Medical Treatment system (MT)	1	28
Sudoku Game (SG)	2	90
Expense Report system (ER)	3	88
Online Conference Review system (OCR)	3	51
Super Stationary system (SS)	2	62
Photography Agency system (PA)	3	162
Incident Management system (IM)	50	115

Table 4. Elements of the Subject Conceptual Schemas

Element	VC	MT	SG	ER	OCR	SS	PA	IM
Classes	5	6	11	7	10	9	15	6
Attributes	19	26	26	36	61	44	43	29
Derived Attributes	2	0	6	6	1	1	33	0
Operations	8	13	19	24	16	32	30	13
Parameters	27	43	48	75	77	91	82	51
Associations	4	5	6	8	10	9	19	4
Derived Associations	0	0	2	0	0	0	0	0
Composite Aggregations	0	0	3	0	0	0	0	0
Constraints	16	9	19	21	14	12	45	8
Generalizations	0	0	4	0	3	0	0	0
ALF units	**10**	**12**	**27**	**16**	**21**	**19**	**35**	**11**

In order to validate the subject CSUT by executing test cases against them, we injected faults into subject CSs and executed the CoSTest testing process (see steps 8-

[8] https://eclipse.org/papyrus/

11 in Figure 2). Then, defects such as missing (e.g. class, attribute, constraint, operation, association), incorrect (e.g. operation, parameter) and extraneous elements (e.g. derived attribute, attribute) were reported by CoSTest. These data are outside the scope of this paper, so that, more detailed information on the testing process (e.g. injected defects, founded defects) can be found in [6].

8 Discussion

Conceptual schemas are particularly useful in discussions with problem domain stakeholders. They are straightforward to understand and a lot of detail can be presented in a well-laid-out, compact diagram. For most people, this is far easier to understand than large blocks of text or written descriptions. However, in order to execute a large number of test cases (i.e. hundreds of them) in a systematic way by exercising different test scenarios and elements of the UML CD-based CS and then to report the defects found, we require a tool that allows us to execute a set of test cases against a CS, so we used ALF in the context of CoSTest tool as the execution environment that provides the ability to execute UML models and report their defects systematically. A tool such as CoSTest, which is based on ALF for representing UML CD-based CD, has two main usage scenarios:

1. **Verifying well-formed UML models by parsing**. The syntax of the language provides the rules for how to construct well-formed statements. The semantics of the language provides the specification of the meaning of well-formed statements. Users (analysts/modellers/testers) could therefore use our ALF–based tool for verifying that a model is actually well-formed. This model can be a conceptual schema under test, as in our validation approach, or a mutant model used to evaluate mutation operators [17], to implement a mutation tool [18] and to evaluate the effectiveness of the test cases of a tool such as CoSTest [6].
2. **Validating UML models by execution**. CoSTest performs verification and validation of UML CD-based CS. This means the tool performs correctness and completeness checks on sets of elements of a CS as well as the elements covered in the test cases are used to identify other defects such as redundant (e.g. duplicated element) and extraneous (e.g. CS elements not used by test cases) elements.

Finally, there is still a long way to go to bring ALF tooling to a level comparable with other existing professional environments that execute and test CS, such as placing breakpoints into ALF specifications, which would make it a lot easier to test and debug complex behaviour.

9 Conclusions and Future Work

ALF and fUML are relatively new standards for building further executable UML specifications and their implementations have only appeared recently. So far, however, most fUML-based execution tooling has been intended to primarily address needs for

system simulation and analyse the behaviour of the model with values manipulated by a tester.

In this paper we have shown that ALF could be used to validate requirements at model level by systematically executing a set of the ALF-based tests. To do so, in Section 2 we described a simple UML CD-based CS built in Papyrus, which was then translated into ALF using a transformation we had developed. The result was a model of a system ready to be tested (i.e. executable) using the CoSTest tool and a set of the ALF-based test cases generated by it. The tool is able to report defects (i.e. syntactically incorrect elements) by parsing the CS, missing and incorrect elements by executing the model, as well as redundant and extraneous elements by coverage analysis of the testing process. We have also used our ALF-based approach in other scenarios to evaluate mutation operators [17], to implement a mutation tool [18] and to evaluate the effectiveness of the test cases of a tool such as CoSTest [6].

We have also reported some limitations of the current ALF-based tool from user feedback. We hope the issues that are related to the standards will be addressed by the Executable UML Working Group in the near future, so that technological improvements resulting from these refinements of the standards can be integrated in the future development of CoSTest. In addition, we will do a comparative analysis of the use of other languages to write the test scripts.

Acknowledgments

This work is supported by SENESCYT of the Republic of Ecuador, Spanish Ministry of Economy, Industry, Competitiveness and the Generalitat Valenciana under the projects the PGE (TIN2016-78011-C4-1-R), FEDER (TIN2013-46238-C4-3-R), TIN2016-80811-P and PROMETEO II/2014/039, and cofinanced with ERDF.

References

1. Granda, M.F., Condori-Fernandez, N., Vos, T.E.J., Pastor, O.: Towards the automated generation of abstract test cases from requirements models. In: 1st International Workshop on Requirements Engineering and Testing. pp. 39–46. IEEE, Karlskrona, Sweden (2014).
2. Seidewitz, E., Tatibouet, J.: Tool Paper : Combining Alf and UML in Modeling Tools – An Example with Papyrus –. In: OCL@MoDELS. pp. 105–119 (2015).
3. Object Management Group: Unified Modeling Language (UML). (2015).
4. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison Wesley (2002).
5. Granda, M.F., Condori-fernández, N., Vos, T.E.J., Pastor, Ó.: CoSTest : A tool for Validation of Requirements at Model Level. In: 25th International Requirements Engineering Conference - Tool Demo (2017).
6. Granda, M.F., Condori-Fernández, N., Vos, T.E.J., Pastor, Ó.: Effectiveness Assessment of an Early Testing Technique using Model-Level Mutants. In: 21st International Conference on Evaluation and Assessment in Software Engineering. , Karlskrona, Sweden (2017).
7. Graw, G., Herrmann, P.: Transformation and Verification of Executable UML Models. Electron. Notes Theor. Comput. Sci. 101, 3–24 (2004).
8. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M., Van de Pol, J., Marchi dos Santos, O.: Automated Verification of Executable UML Models. In: International Symposia on Formal Methods for Components and Objects. pp. 225–250 (2010).

9. Laurent, Y., Bendraou, R., Baarir, S., Gervais, M.-P.: Formalization of fUML : An Application to Process Verification. In: International Conference on Advanced Information Systems Engineering. pp. 347–363 (2014).
10. Xie, F., Levin, V., Browne, J.C.: Model Checking for an Executable Subset of UML. In: 16th IEEE International Conference on Automated Software Engineering (2001).
11. Lai, Q., Carpenter, A.: Defining and Verifying Behaviour of Domain Specific Language with fUML Categories and Subject Descriptors. In: Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications (2012).
12. Micskei, Z., Konnerth, R., Benedek, H., Semeráth, O., Vörös, A., Varró, D.: On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf. In: 1st Workshop on Open Source Software for Model Driven Engineering. pp. 31–41 (2014).
13. Planas, E., Cabot, J., Gómez, C.: Lightweight and static verification of UML executable models. Comput. Lang. Syst. Struct. 46, 66–90 (2016).
14. Guermazi, S., Tatibouet, J., Cuccuru, A., Dhouib, S., Gérard, S., Seidewitz, E.: Executable Modeling with fUML and Alf in Papyrus : Tooling and Experiments. In: 1st International Workshop on Executable Modeling. pp. 3–8 (2015).
15. Tatibouët, J., Cuccuru, A., Sébastien Gérard, Terrier, F.: Formalizing Execution Semantics of UML Profiles with fUML Models. In: International Conference on Model Driven Engineering Languages and Systems. pp. 133–148 (2014).
16. Mayerhofer, T., Langer, P., Wimmer, M.: xMOF : A Semantics Specification Language for Metamodeling. In: Satellite Events of MODELS (2013).
17. Granda, M.F., Condori-Fernandez, N., Vos, T.E.J., Pastor, Ó.: Mutation Operators for UML Class Diagrams. In: CAiSE (2016).
18. Granda, M.F., Condori-fernández, N.: A Model-level Mutation Tool to Support the Assessment of the Test Case Quality. In: ISD (2016).
19. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML). (2012).
20. Massimo, T., Jouault, F., Saidi, Z., Delatour, J.: Enabling OCL and fUML Integration by Transformation. In: European Conference on Modelling Foundations and Applications. pp. 156–172 (2016).
21. Mohagheghi, P., Dehlen, V., Neple, T.: Definitions and approaches to model quality in model-based software development - A review of literature. Inf. Softw. Technol. 51, 1646–1669 (2009).
22. España, S., González, A., Pastor, Ó., Ruiz, M.: Technical Report Communication Analysis and the OO-Method : Manual Derivation of the Conceptual Model the SuperStationery Co. Lab Demo. , Valencia (2011).
23. España, S., González, A., Pastor, Ó., Ruiz, M.: Integration of Communication Analysis and the OO-Method: Rules for the manual derivation of the Conceptual Model. , Valencia (2011).
24. Tort, A., Olivé, A.: Case Study: Conceptual Modeling of Basic Sudoku, http://guifre.lsi.upc.edu/Sudoku.pdf.
25. Tort, A.: A Basic Set of Test Cases for a Fragment of the osCommerce Conceptual Schema, http://hdl.handle.net/2117/6130.
26. Planas, E., Olivé, A.: The DBLP Case Study, http://guifre.lsi.upc.edu/DBLP.pdf.