TOWARDS THE SEPARATION OF CONCERNS BETWEEN MODEL DESIGN AND VALIDATION

Azza Shawki1, Moustafa K. M. Elrawy2, Islam A. M. El Maddah3, Hoda K. Mohamed4  
School of Computer Engineering, Ain Shams University, Cairo, Egypt  
Azzashwki1985@yahoo.com, eng_Mostafa_Khalf@yahoo.com, islam_elmadah@eng.asu.edu.eg, hoda.Korashy@eng.asu.edu.eg

Abstract

Recently, software design process has clearly become much more complicated, thus the integrity of software system design has come to be very difficult to be verified. For this reason, the unified modeling language (UML) has been developed to be a standard language for designing software. Verification and validation of UML models are urgently needed since UML models exhibit behavior is neither wanted nor expected by the designer and this for sure causes severe problems during runtime. This step is usually done by a model checker tool. There are many model checker tools. However, this is usually accomplished by experts who have a good awareness of the model checker language and its construction. That’s why it is necessary for a model validator to work along with the model designer. Thereby achieving the concern of separation accomplishes this step. We construct a bridge to tighten the gap between design and analysis processes through the USE2ALLOY tool. This tool can transform the model produced by the designer to a model suitable for conducting automated analysis by the validator. This paper presents a study of USE2ALLOY with OCL constraints. The translation allows analysis of UML models via Alloy to identify consistencies in those UML models and search for their instances with minimum effort from the designer. We compare our work to other related works and draw conclusions to a future work.

Keywords: UML; class diagram; OCL Model; Constraint; Alloy analyzer; Signature; USE tool; USE2ALLOY

1. INTRODUCTION

As the civil engineer constructs buildings according to the architect’s designs and so does the developer who writes program code according to the software engineering design. Recently, software design process has clearly become much more complicated thus the integrity of software system design has come to be very difficult to be verified. For this reason, the unified modeling language (UML) has been developed to be a standard language for designing software. Verification and validation of UML models are needed because UML models exhibit behavior is neither wanted nor expected by the developer and those behaviors could cause serious of bugs, this step is usually done by a model checker [1]. There are many model checking tools. Model checking programs have been extensively used to verify various systems. However, this is usually accomplished by experts who have a good understanding of the model checker language and its construction. The model designer/developer must be familiar with the syntax of both modeling and property specification languages. Unfortunately, this is not an easy task for non-experts to learn descriptive languages for modeling the system and specifying its invariants/properties. In particular, property specification is a very daunting and error-prone for non-experts and that is why a model validator is obliged to work along with the model designer. Within this classification, the developer/designer has to concern about the construction of the model, the classes and developing process while the validator checks the properties against customer requirements. We mainly aim to tighten the gap between design and analysis technical spaces by creating tools that allow the model produced by the designer to be a model suitable for the validator’s job. A large scale of tools, by which drawing and documentation of UML diagrams are easily constructed, already exists. We choose a USE tool to present UML diagram, on other hand, we choose an Alloy model checker to form the analysis of that model. This paper presents a study of USE2ALLOY, a tool for transforming UML models in form of UML Class Diagrams – presented in USE notation – with OCL constraints1, to Alloy. The conversion allows analysis of UML models via Alloy, to identify consistencies in those UML models and looking for their instances. The following Figure 1 demonstrates our tool.

Figure 1: USE2ALLOY

1 OCL “Object Constraint Language”, constraint is a restriction on one or more values of the system.
We show how UML models can be automatically transformed into Alloy which in return allows us to analyze the model by the Alloy Analyzer. The rest content of this paper is structured as following: **In Section 2**, we describe why we have specifically chosen USE tool, Alloy model checker, and then we compare between them. **While in Section 3**, we describe our approach to validate UML and OCL. **In Section 4**, a case study is used to demonstrate the key features of our tool with respect to the validation process. **In Section 5**, we are presenting a report on the results produced by Alloy analyzer to the model. **Sections 6 and 7** are a conclusion with a summary and deductions for future works.

2. **Why WE CHOOSE THE USE TOOL AND THE ALLOY TOOL**

2.1 **WHY USE TOOL?**

USE (UML-based Specification Environment) allows you to textually define UML model; it is based on a UML class diagram and invariants presented in OCL [2].

There are many features that can be used in this tool. It supports syntactic analysis, type checking, dynamic validation of invariants and pre/post conditions specified in OCL. Also it supports object and sequence diagrams. Furthermore, part of the USE system is a snapshot generator which is based on ‘a snapshot sequence language’ (ASSL) is used to validate and verify specifications. We will distinguish the built-in snapshot generator from our new validator tool.

Here we have chosen USE Tool because it is simple, easy to use, and supports designer with a little validation in executing UML model and checking OCL constraints.

2.2 **WHY ALLOY MODEL CHECKER**

There are slight different categories known as a model finder “kind of tools that define a model which satisfies the system properties”. Alloy is an example of that program which implements techniques for finding finite models based on relational first order logic.

Alloy by Daniel Jackson [3] is a high level modeling language that makes an automatic analysis to a system. It is a symbolic model checker based on first-order logic with relations, as a kind while other model checkers are based on temporal logic. Alloy uses SAT-solvers to verify the satisfiability of properties defined in a model, and to find counter examples for properties.

Unlike a programming language, Alloy model is declarative; it defines the effect of behaviour without giving their mechanism. This allows very partial model to be constructed and analyzed. All kinds of systems can be modeled and analyzed by Alloy, but most used in those involving complex structured state.

One area in which Alloy offers particular advantage over model checkers is for analyzing systems with configurations that are undetermined, or that can change dynamically. In regard to others, it is successfully used for modeling and analyzing of distributed Systems protocols, network configuration protocols, access control, etc.

The following Table 1 specifies more difference between Alloy and other model checker.

Table 1: Comparison between Alloy and other model checker

<table>
<thead>
<tr>
<th>Alloy Analyzer</th>
<th>Other model checkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>works as model finder</td>
<td>Works as model checker that only makes analysis for the whole system</td>
</tr>
<tr>
<td>model checker : creates model that satisfies system properties</td>
<td></td>
</tr>
<tr>
<td>analyzes operations within complex states</td>
<td>analyze state composed of several simple state running in parallel</td>
</tr>
<tr>
<td>Allows structural constraints on the states to be directly described (with sets and relations)</td>
<td>Provide only low-level data type” such as array and records in describing constraint.</td>
</tr>
<tr>
<td>Easily handles data structure of (tables, trees, etc.) because it is based on relations</td>
<td>Still described with simple input language” such as arrays and enumerations.</td>
</tr>
<tr>
<td>Has no built-in idiom, it is possible to express and check properties that cannot always be checked with model checkers</td>
<td>Have built-in idioms, so they can’t check all properties</td>
</tr>
</tbody>
</table>

2.3 **USE ANALYZER VS. ALLOY ANALYZER**

2.3.1 **USE ANALYZER**

USE tool supports designer/developer in both design and analysis stages.

The main task of USE analyzer is to check whether the model satisfies system requirement “OCL invariant “or not. In addition, if the “pre/post condition” operations violate operation specifications or not.

The validation of USE tool stands on ASSL “A snapshot sequence Language “procedures which generate system states, USE generates one snapshot after another until it finds the valid one which is specified to system properties. An ASSL is manually

---

3 Pre-conditions are the things that must be true before a method is called. Post-conditions are the things that must be true after the method is complete.

3 The Boolean Satisfiability Problem abbreviated as SATISFIABILITY or (SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.
defined by ‘Try’ statement which defines a complete set of snapshots “snapshot space”. The disadvantages of USE analyzer can be explained as following: First, it has an enumerative nature as it has to create and check each snapshot. This means larger snapshot space for instance comprises more than a few objects and attributes values, so all possible links cannot be handled. In addition, it takes much longer than it should. Second, USE Tool requires some efforts from designer/developer to check the system state. They should construct the object diagram manually by issuing commands to create and destroy objects, inserting and removing links between objects, and setting attribute values of objects. “Suppose big system”.

2.3.2 ALLOY ANALYZER
The Alloy Analyzer is a tool used to edit, build, and test specifications written in Alloy language. It can automatically analyze properties of UML/OCL models by translating them into relational structures. These relations can be handled by the model finder Kodkod which in return employs SAT solvers to find solutions.

The Alloy analyzer has two kinds of analysis for searching in the specification:

1- Checking: the Analyzer searches for a counter-example to the specification, which would refute the specification’s correctness.

2- Running: it works on finding an example that satisfies all of the constraints of the given specification, basically providing a simulation of the specification.

The Analyzer produces graphic representations for the system “Alloy metamodel” which helps users to better understand and visualize the specification model, evolve or expand system in future. Also, the model can be represented in a tree or a text form. On the other hand, Alloy has a visualizer, which receives commands “constraints” and gives us the results.

The common problems within USE analyzer can be easily got rid of in Alloy analyzer. The Alloy analyzer has an automatic generation of object diagrams which is model-based on translation of UML and OCL to a relational logic, that conforms system properties within specified scope “general bounds to the number of objects, links and attribute values”, making the configuration considerably less-time-consuming.

One of the cooler features in Alloy, it can give us all object diagrams instances “scenarios / example” for an automatically certain model product without any effort from the designer. In USE tool, the designer has to create the object manually which exhausts them in big systems. We cannot forget the wonderful “projection” command in Alloy which used to hide one or more objects in the model in order to focus on others”. NB: we can hide the built in classes like Integer, String … etc.

3. SPECIFYING A UML MODEL WITH USE TOOL AND ALLOY MODEL CHECKER

3.1 SPECIFYING A UML MODEL WITH USE TOOL (CASE STUDY)

The Input of USE Tool is a class model, OCL invariants, set of USE commands “for creating object models” and operation invocations.

Figure 2 illustrates a network system. The Network here consists of nodes, switches, packets, messages and one controller for each node.

Nodes can communicate with each other by ports. Each node consists of a set of switches that is over taken by one controller. Each Switch contains tables that store flow entries. We represent our network model in USE tool by defining a class diagram in a text file as follows Figure 3

```
Figure 3: Network system
```

```
model network
  class Network
    attributes content: Node
    end
  class Node
    attributes end
    class Table
      attributes content: Node
      end
    class Controller
      attributes class Table
      end
  class Controller
    attributes class IP
    end
  class Table
    attributes tables: Table
tables:Table
end
```

```
Figure 3: Classes represented in USE notation on a text file
```

The associations between classes represented in USE as follows in Figure 4

```
association containment association control
between between
Node[*] Switch[*]
end
association has between association srcdesIP
Node[1] between
Controller[*] IP[1]
end
Packet[1]
association contains end
between association msgInPort
Node[1] between
Switch[*] Port[1]
end Message[*]
```
Finally, we define the constraints which conform to the structure of our network model and assure that it will have the ability to work probably right as following in Figure 5.

**Constraints**

**context Network**

```plaintext
-- nodes belong to only one network
inv: Node.allInstances->forAll(n:Node |self.content = n)
```

**context Switch**

```plaintext
inv: self.tables.isDefined implies self.controller.isDefined
-- every table belongs to exactly one switch
inv: Table.allInstances->forAll(t:Table | t.switch = 1)
```

```plaintext
-- each switch has only one last table
inv: Switch.allInstances->forAll(s:Switch | s.tables.nxTable->size = s.tables->size - 2)
```

**context Table**

```plaintext
-- no Next table of a table can be the table itself
inv: self.nxTable<> self
-- no same next table for two different tables
inv: Table.allInstances->forAll(t1,t2 | t1<> t2 implies t1.nxTable <> t2.nxTable)
```

```plaintext
-- the next table of every table must be in the same switch
inv: Table.allInstances->forAll(t:Table | t.nxTable.switch = t.switch )
```

**context Port**

```plaintext
-- All the connections are two ways
inv: Port.allInstances->forAll(p1, p2 | p1 = p2.connect implies p2 = p1.connect )
-- every port belongs to exactly one node
inv: Port.allInstances->forAll(p:Port | p.node=1)
```

**context Network**

```plaintext
 inv: Node.allInstances->forAll(n:Node |self.content = n)
```

**context Switch**

```plaintext
 inv: self.tables.isDefined implies self.controller.isDefined
-- every table belongs to exactly one switch
inv: Table.allInstances->forAll(t:Table | t.switch = 1)
```

```plaintext
-- each switch has only one last table
inv: Switch.allInstances->forAll(s:Switch | s.tables.nxTable->size = s.tables->size - 2)
```

**context Table**

```plaintext
-- no Next table of a table can be the table itself
inv: self.nxTable<> self
-- no same next table for two different tables
inv: Table.allInstances->forAll(t1,t2 | t1<> t2 implies t1.nxTable <> t2.nxTable)
```

```plaintext
-- the next table of every table must be in the same switch
inv: Table.allInstances->forAll(t:Table | t.nxTable.switch = t.switch )
```

**context Port**

```plaintext
-- All the connections are two ways
inv: Port.allInstances->forAll(p1, p2 | p1 = p2.connect implies p2 = p1.connect )
-- every port belongs to exactly one node
inv: Port.allInstances->forAll(p:Port | p.node=1)
```

Figure 6: Class diagram in USE tool

Figure 7: Object diagram in USE tool

After making analysis, we find the following Figure 8

**Figure 5: Constraint in USE notation**

**Figure 6: Class diagram in USE tool**

**Figure 7: Object diagram in USE tool**

**Figure 8: USE Analysis Log file**

**Figure 9: USE class invariant**

We can double click on the false field to show the reasons of false as follows in Figure 10

**Figure 10: Evaluation browser shows the false reasons**

### 3.2 SPECIFYING A UML MODEL WITH ALLOY

The basic structure of Alloy is an atom that describes something in real world, a set of atoms is known as a tuple, a group of tuples is known as relations.

The Alloy uses several expressions to describe the usage of atom and relation. Some of these expressions are: signatures, facts, assertions, predicate and functions.
1. A **sig.** (signature) defines sets of atoms, and their relationship.
   Ex: **sig Switch** {controller: one Controller, contains: one Node, tables: Table}

2. A **fact;** is statement that defines constraints on the elements of the model, creates a restriction on relationships and limits the possible values that they may contain.
   Ex: **fact Switch** {all t: Table | #t.switch = 1}
   This means that the table must be related to only one switch.

3. A **fun** (function) is a reusable formula that is applied to a set of typed parameters to define relations between them and produce appropriate results.

4. A **pred.** (predicate) a (true or false) formula that sometimes makes parameters used in getting its result. Predicates are parameterized constraints.

5. An **assert.** (assertion) Assertions are statements that should be true about the system.

There are two forms of commands.

6. The **check** command, used to check an assertion, in order to see whether the Alloy analyzer finds any counter examples.

7. The **run** command, used to **run a predicate/function in order to find an instances or examples (if any) the Alloy analyzer finds for which the predicate is true.

**Scope;** is general bounds to the number of objects, links and attribute values.

We can define the **scope** that the analyzer checks by saying the number of times that our model running for. Also we can define the number of objects on it.

**Ex:** “run for 5” or “run for 5 but 2 Switches”. The analyzer will then check only possible examples that contain no more than that many of atoms from each set. If it finds an example, then the predicate is satisfiable. If it finds no examples, the predicate may be either invalid or it may be satisfiable but not within the scope you have used.

Both commands take the name of the assertion or function, and an indication of the scope in which the analysis is to be performed.

4. **MAPPING CLASS DIAGRAMS AND OCL FROM USE TO ALLOY**

This section presents a brief on rules we have used to make this transformation from USE Class Diagram to Alloy. We create our tool in Java Language and we take a help from NetBeans workspace.

4.1 **MAPPING USE CLASS TO ALLOY**

The transformation commences by defining the basic elements of each language and maps it from USE to Alloy. The basic entities of USE tool are classes, association between classes, operations and the constraints. The basic entities of Alloy are signature, which represents atoms, fields that represent the relation of atoms; facts represent constraint of model and predicates, which are parameterized constraints.

We map each class to a signature; attribute to an atom, association to a field, operation to a predicate and constraint to fact or predicate.

**Ex:** Translations “switch” from a class in USE to a signature in Alloy.

For USE

```
USE
class Switch
  attributes
    tables: Table

end
```

For Alloy

```
Alloy
sig Switch {
  attributes
    tables: Table
}
```

In Association, the class with the higher multiplier will converted to a field placed inside the class with smaller multiplier as following.

For USE

```
USE
Association control between
  Switch [*]
  Controller [1]
End
```

For Alloy

```
Alloy
sig Controller{
  control: set
  Switch,}
```

If association has a role name, the filed inserted in signature must be defined with associated role name.

**Ex:**

For USE

```
USE
Association control between
  Switch [*]
  Controller [1] role ascontrol
End
```

For Alloy

```
Alloy
sig Controller{
  ascontrol: set
  Switch,}
```

Table 2 shows the correspondence between the main elements of USE Classes/OCL and Alloy.

<table>
<thead>
<tr>
<th>USE Class model</th>
<th>Alloy model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Signature</td>
</tr>
<tr>
<td>Attributes</td>
<td>Atoms of signature</td>
</tr>
<tr>
<td>Operation</td>
<td>Predicate, function</td>
</tr>
<tr>
<td>Constraint</td>
<td>Fact</td>
</tr>
<tr>
<td>Pre &amp; post condition</td>
<td>Fact</td>
</tr>
<tr>
<td>parameterized constraints</td>
<td>Predicate</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Extended signature-fact</td>
</tr>
</tbody>
</table>

4.2 **MAPPING USE OCL TO ALLOY**

Most of the OCL collections operations have a corresponding Alloy expression.

**Ex:** we have the following OCL invariant in USE file converted to fact in Alloy file.

In USE file

```
context Switch
inv: Table.allInstances->forAll(t: Table | t. switch = 1)
```

In Alloy file

```
fact Switch {all t: Table | t. switch = 1}
```

Table 3 lists the mapping between some of OCL expression and Alloy.
Table 3: Mapping between USE OCL and Alloy

<table>
<thead>
<tr>
<th>Use Class model</th>
<th>Alloy model</th>
</tr>
</thead>
<tbody>
<tr>
<td>forAll</td>
<td>All</td>
</tr>
<tr>
<td>Exists</td>
<td>some</td>
</tr>
<tr>
<td>And</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Or</td>
<td></td>
</tr>
<tr>
<td>Not</td>
<td>!</td>
</tr>
<tr>
<td>Size()</td>
<td>#</td>
</tr>
<tr>
<td>Includes, includeAll</td>
<td>in</td>
</tr>
<tr>
<td>Excludes, excludeAll</td>
<td>:in</td>
</tr>
<tr>
<td>Including</td>
<td>+</td>
</tr>
<tr>
<td>Excluding</td>
<td>-</td>
</tr>
<tr>
<td>isEmpty</td>
<td>no</td>
</tr>
<tr>
<td>notEmpty</td>
<td>some</td>
</tr>
<tr>
<td>Exp callexp</td>
<td>Exp callexp</td>
</tr>
<tr>
<td>If exp then exp else exp</td>
<td>Exp =&gt; exp else exp</td>
</tr>
</tbody>
</table>

The tool is written in java language using NETBEANS workspace, any system supporting Java is enough to run this tool. Figure 11 shows the interface of our USE2ALLOY tool.

5. USE2ALLOY–CASE STUDY

Here we will discuss how to transform the last example “Network system” from USE to Alloy.

The input of USE2ALLOY tool is our network system which is presented and designed by USE notation in text file. The output of our tool is Also text file containing our system but with Alloy notation. Now we can easily make analysis by Alloy analyzer. The following Figure 12 represents model after transformation.

By executing this code in Alloy, we can make an analysis and find results from analyzer as following Figure13, Figure14.

Above message shows us that system is consistent and there are instances “run examples”. The following window Figure 15 “opened automatically” shows us the instances of model.

We can get all instances (example or scenarios for our system) by pressing on next bottom at tool bar as follows Figure 16.

We can get the Alloy model by opening the metamodel from “Execute” menu as following Figure 17.
6. RELATED WORK

[4] Presents a tool called UML2Alloy, the author discusses the challenge of transforming UML class diagram & OCL invariants to Alloy via MDA. The transformation here is based on translating UML Metadata to Alloy Metadata. The problem of this technique is that it is limited to a program that has Metadata export method.

Other program cannot use this method like our tool USE Tool does. In addition, it has some limits in translating, for example, they cannot translate multiple inclusions directly to Alloy.

[5] Presents another tool called CD2Alloy which also translates Class diagram to Alloy model checker. But unfortunately does not support constraints like our tool. Also, it is represented as a plug in that takes some advantages of Alloy analysis but not all. For example, the powerful projection commands in Alloy that hides class to focus on others. Also, lists all instance of model “all scenarios of object diagrams”.

Paper [6] has a study involving the open flow Network Switch, the author presents the model on Alloy model checker and he tries to separate a network into a data plane and a programmatic control plane. This enables easy network reconfiguration. The problem here is that the model is big and it is too hard to define it directly in Alloy. So we decide to construct the model in USU tool, then apply our transformational USE2ALLOY tool in order analyze it easily in Alloy model checker.

7. CONCLUSION AND FUTURE WORK

We present here a new tool called USE2ALLOY tool. This tool translates model from design stage “USE model “to Analysis stage “Alloy Model Checker”.

USE tool can represent the structure of any model by defining textually the class diagram and OCL invariants. Also pre/post condition easily. It can verify the syntactic & type checking of model, model consistency, OCL constraints and pre/post invariant.

Alloy is mainly used as a model checker and sometimes as a model finder. It has been used to make verification and validation for a model, also it can guide designer through constructing the structure of model, produce a model even with few input lines of code without any illustration.

In addition, Alloy can find counter object diagram examples for the constrains written by the USE tool developer. Alloy also is capable of automatically generates Object models that can highlight and discover problem and critical parts of the class model expressed in the USE tool.

By this transformation, we can use both easily, one for class model designer in “design stage” and the other as a validator in “model analysis stage”.

We differentiate between tool analyzers which already exist in both USE tool and Alloy model checker, points out the advantages and disadvantages of each and then transform models from one to another.

As for future work, an opposite transformation from Alloy to USE is suggested. This could trace back the problems in model and their corrections to the designer space at the USE environment.

Another recommended way is to use Alloy model checker in defining and verifying the dynamic behavior of a model.

REFERENCES


[4] Seyyed M.A. Shah, Kyriakos Anastasakis, and Behzad Bordbar, From UML to Alloy and Back Again, School of Computer Science, The University of Birmingham, UK.

[5] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited, Software Engineering, RWTH Aachen University, Germany.


[9] Yueming He, Comparison of the Modeling Languages Alloy and UML, Department of Computer Science Portland State University, Portland, Oregon, USA.