EAV - Early Analysis Verification

Model Checking of UML Class Diagrams using Relational Logic

Patrick VOGT and Lars-Erik KIMMEL

Informatik (M. Sc.)
Hochschule RheinMain
University of Applied Sciences

February 20, 2012

Contents

1 Introduction

The following document describes an internal project which was done from October 2011 to February 2012 at the University of Applied Sciences Wiesbaden, Germany.

Model checking technologies are often used just for safety critical software systems and are mainly omitted in the everyday software engineering process [BA08, Som10].

To avoid errors in the software development process, there are several generic model checking tools such as the Alloy Analyzer [Jac02, Jac11] or the SPIN Model Checker [Hol97].

So the motivation of this project was finding a possible way to provide model checking for UML class diagrams including OCL constraints using those generic model checking tools. This approach could help to apply model checking technologies to the everyday software engineering workflow because the input data for such a model checker tool are the well-used UML class diagrams. Beneath researching the theory of UML model checking a prototypical software system should be developed which realises these theoretical concepts.

It is possible to apply model checking techniques to an UML class diagram by manually transform a diagram into models which can be analysed (such as an Alloy model) or using the tools which are described in section 2. Section 3 gives a short introduction into the Alloy language for understanding the presented transformation into Alloy code in section 4. This transformation provides finding valid instances of the class diagram as object diagrams or finding counterexamples which contradicts a given assumption.

Section 5 presents the current results and the remaining problems of the presented strategy and section 6 concludes the project's work.

2 Related Work

There are several existing UML model checking approaches which are directly related to this project's work: USE [GBR07], UML2Alloy [ABGR] and FMC [HW12].

USE offers an environment to specify a model given as UML class diagram including OCL textually. Snapshots of the given model can be constructed manually, checked against the corresponding model for validity and are viewed as object diagrams. However USE is still based on a manual definition of snapshots. It is not yet possible to automatically generate valid instances of a model that fulfill every specified constraint and to iterate over those instances similar to the Alloy Analyzer or Constraint Programming. Furthermore it is not possible to force USE to show counterexamples of a class diagram which contradicts a given assumption.

UML2Alloy transforms an UML class diagram including a subset of OCL given as XMI into an Alloy model. The tool is still in beta state and also provides an external view for showing valid instances for the UML class diagrams as object diagrams [Anab]. Unfortunately the reference manual of UML2Alloy suggests that the produced class diagrams should be specified with one specific outdated version of ArgoUML [Anaa].

A similar project is done by [HW12]. They transform a class diagram including OCL into Formula using constraint logic for finding valid instances of the class diagram and showing possible counterexamples for a given assumption.

3 USE and the Alloy Language

This section should give a minimal introduction into the USE domain specific language and into the Alloy language. It shouldn't and can't replace the introductions given at [Ham11] and [Jac11].

For specifying an UML class diagram which should be model checked the domain specific language USE is re-used. The parser was extracted from the USE project and was re-used for the transformation of the class diagram into an Alloy model.

The domain specific language is presentend with a concrete class diagram:

```
model Company

-- classes

class Employee
attributes
  name : String
  salary : Integer
operations
  promotion()
end

class Department
attributes
  name : String
  location : String
  budget : Integer
end

class Project
attributes
  name : String
  budget : Integer
end

-- associations

association WorksIn between
  Employee[*]
  Department[1..*]
end
```

```
association WorksOn between
  Employee[*]
  Project[*]
end

association Controls between
  Department[1]
  Project[*]
end
```

Listing 1: A short example of the USE specification language ([Ham11])

Most of the syntax of USE is self-explanatory. To minimize possible misunderstandings, the cardinalities of associations will be highlighted in the next paragraph with an example.

The assocation `Controls` in Listing 2 means that one `Department` can be associated with several `Projects` and that one `Project` belongs to exactly 1 `Department`.

```
association Controls between
  Department[1]
  Project[*]
end
```

Listing 2: An example to highlight the semantic of the association cardinalities ([Ham11])

Furthermore the OCL constraints are specified in a special section for the constraints. The model in Listing 1 can be expanded with OCL constraints for class invariants and pre- and postconditions.

```
...

-- OCL constraints

constraints

context Department
  -- the number of employees working in a department
  -- must be greater or equal to the number of
  -- projects controlled by the department
  inv MoreEmployeesThanProjects:
    self.employee->size >= self.project->size
```

```
context Employee
  -- employees get a higher salary when they work on
  -- more projects
  inv MoreProjectsHigherSalary:
    Employee.allInstances->forAll(e1, e2 |
      e1.project->size > e2.project->size
        implies e1.salary > e2.salary)

context Project
  -- the budget of a project must not exceed the
  -- budget of the controlling department
  inv BudgetWithinDepartmentBudget:
    self.budget <= self.department.budget

  -- employees working on a project must also work
  -- in the controlling department
  inv EmployeesInControllingDepartment:
    self.department.employee->includesAll(self.
       employee)
```

Listing 3: An example for OCL invariants ([Ham11])

The data structure, the constraints and the operations of the class diagram will be translated into Alloy which is a language for specifying relational models combined with predicates in first order logic.

Classes cannot be mapped directly into the Alloy Language. The smallest union in Alloy is called a *Signature* and represents a set of relations. Alloy should be explained with the definition of a graph:

```
sig V
{
  e: set V
}
```

Listing 4: Alloy signature which represents a graph

The signature `V` represents the nodes of the graph and the corresponding relation `e` represents the edges of the graph. Every relation within a signature has got a type which is combined with the signature it belongs to. Therefore the relation `e` is from the the type `V x V`. So the data structure of the class diagram: the classes, their corresponding attributes and the associations between the classes can be mapped to Alloy signatures. Similar to the class structure, signatures can be `abstract` or can be `extended`. The semantic of both keywords should be known from the objectoriented programming context.

Additional to signatures, Alloy supports predicates which can restrict the relational model. Those predicates can be specified in a syntax which resembles the first order logic. Listing 5 shows the predicate for preventing that a node contains a reflexive edge.

```
pred NoReflexiveEdges
{
  -- for all nodes v: the tuple (v,v) is not element
  -- of the relation e (which represents the edges of
  -- the graph)
  all v: V | not ((v->v) in e)
}
```

Listing 5: Alloy predicates

A second predicate (given in Listing 6) restricts the graph such that every node within the graph is linked to at most two other nodes. The operator "#" is used in Alloy for retrieving the number of tuples within the surrounding relation set. The second new operator "<:" is called domain restrictor and filters every tuple in the right hand relation which contains the left hand element in the first column of the tuple.

```
pred MaxTwoOutEdges
{
  -- for all nodes v: v is linked to at most two
  -- other nodes
  all v: V | #(v <: e) <= 2
}
```

Listing 6: Alloy predicates

Some parts of the data structure, the cardinalities for instance and the OCL constraints of the class diagram are translated into corresponding predicates, that must be fullfilled by the model.

As Alloy is used for finding instances of "large" static relational models, there are no syntactical constructs for describing structual changes over a certain time. The instance which is found is static and cannot be changed dynamically e.g. to construct the class diagram incrementally.

A complete introduction into the Alloy language gives the book from Daniel Jackson [Jac12].

4 Transformation Rules

This section describes the transformation of the UML class diagram including OCL into the Alloy language. The description is separated into the different subsections: data structure, constraints and operations.

4.1 Data Structure

The data structure of the class diagram is transformed into first order logic. The transformation rules are mainly based on [BCG05] and slightly adapted for a further transformation into the Alloy language:

A class `A` with the attribute `name` of type `String` and an assocation `aToB` to an empty class `B` is transformed into one single Alloy signature. Every attribute and every assocation of a class will be transformed into an unique relation within this signature. Listing 7 gives an example for the transformation of class `A` and class `B` into corresponding Alloy signatures.

```
sig Class"A
{
  attr"a"name: one TypeString
  asso"aToB"a"b: set B
}

sig Class"B
{

}
```

Listing 7: Transformation of a Class into a Signature

The token "`"`" is used as a delimiter for separating several parts of an identifier. This token was chosen as a delimiter because it can be part of an Alloy identifier but cannot be used in the USE domain specific language.

The name of the signature always contains two parts: the prefix `Class` and the actual name of the class. A relation which represents an attribute contains the prefix `attr`, the name of the attribute and the name of the class it belongs to. A relation which represents a binary association contains four parts within an identifier: the prefix `asso`, the name of the association, the name of the first associated class and the name of the second associated class. Those naming schemes garantuees that all identifiers within the model are unique. The prefix `asso` and `attr` was chosen for distinguishing relations for attributes and associations. This decision

helps to draw the acutual instances of the class diagram as UML object diagrams. Two of the UML primitive types are supported in the Alloy language: `Boolean` and `Integer`. The primitive types `Real` and `String` must be emulated and filled with dummy values. Those two types are therefore represented through the signatures `TypeReal` and `TypeString`. As the instances of those signatures are replaced with dummy values it is not possible using those types within OCL constraints.

The cardinalities are restricted with predicates for the corresponding relations. Every instance of the class `A` must be combined with exaclty 1 String for the attribute `name`. This attribute restrictions will be expressed with the keyword `one` in the definition of the relation type (see Listing 7).

The cardinalities of the associations cannot be restricted in this way. Therefore a predicate in first order logic is needed to specify the lower bound l_A and the upper bound u_A of the assocation $aToB$ (direction from class `A` to class `B`).

```
pred assocardAToB
{
  -- for all instances a of class A:
  all a: Class"A |
    --the number of B's combined with a is at least l_A
    --AND
    --the number of B's combined with a is at most u_A
    (l_A <= #a.aToB) and (#a.aToB <= u_A)
}
```

Listing 8: Cardinality Restriction for an Association

An assocation is not modelled as a symmetric association. So the association $aToB$ between the classes `A` and `B` will only be specified either in the signature `Class"A` or in the signature `Class"B`. Therefore the same assocation $aToB$ has to be restricted from the *view* of class `B` (to class `A`) with the lower bound l_B and the upper bound u_B.

```
pred assocardBToA
{
  -- for all instances b of class B:
  all b: Class"B |
    --the number of A's combined with b is at least l_B
    --AND
    --the number of A's combined with b is at most u_B
    (l_B <= #aToB.b) and (#aToB.b <= u_B)
}
```

Listing 9: *Reverse* Cardinality Restriction for an Association

These transformations present the implemented subset of the UML specification for UML class diagrams. In addition there are still more transformations e.g. for n-ary associations and association classes.

4.2 OCL Constraints

For providing OCL support a subset of OCL is aswell transformed into relational logic and is part of the model. The OCL transformations won't be explained in this document. The transformation was implemented in an OCL visitor which visits every part of the OCL constraint and transforms it into an equivalent Alloy predicate.

The following OCL operations are currently supported:

- AllInstances
- AsType
- AttrOp
- Collect
- ConstBoolean
- ConstEnum
- ConstInteger
- Exists
- ForAll
- If
- IsTypeOf
- Navigation
- Empty Set-Literal
- Variable
- Compare Operations: $>=$, $>$, $=$, $<>$, $<$, $<=$
- Logical Operations: and, or, $implies$
- Set Operations: $including$, $includes$, $includesAll$, $excluding$, $excludes$, $excludesAll$, $intersection$, $union$
- Collection Operations: $size$, $count$, $isEmpty$, $notEmpty$
- Init Operations: $oclIsNew$

4.3 Operations

For supporting method bodies of UML classes the pre- and postconditions of OCL are used. Therefore every relation rel is combined with an unary relation $Time$ which represents the *value* of this relation for a given time. This approach is called *Time Axis* and is described in [Jac12].

An operation `setValue(v: String): void` which is called on the object x_0 in time t_1 and replaces the old value v_0 with the new value v_1 could therefore be represented through the following relations:

$Time = \{(t_0), (t_1), (t_2), \ldots\}$, $value = \{(x_0, v_0, t_1), (x_0, v_1, t_2)\}$

Additionally every object in the UML object diagram contains a relation $isAlive$ which represents the lifetime of an object. If and only if an element of the unary relation $Time$ is related with this object, the object is instanciated for this given time. Therefore it is possible to specify operations that constructs or destructs an object with pre- and postconditions. For every time only one atomic operation is called upon one object in the object diagram.

Additional to the OCL expressions in Subsection 4.2 the $@pre$ expression is supported for specifying changes within the method body.

5 Problems

There are still problems concerning the transformation of UML class diagrams including OCL constraints. These problems should be described shortly in this section. The problems are: aggregate functions for OCL collections, realising non-atomic operations, filtering symmetric instances of the class diagram.

5.1 Aggregate Functions

A challenging transformation is e.g. the operations for OCL collections such as `sum()` which iterates over a data set and calculates temporary values. The elements of a collection $E = \{e_0, e_1, \ldots\}$ must be combined with an ordered index set $I = \{(i_1, i_2), (i_2, i_3), (i_3, i_4) \ldots\}$. The last column of the relation $E \times I \times Integer$ can contain the temporary sum while adding all the elements of E and iterating over the index set I with the defined sequence in I. With this pattern the result of the `sum()` operation is then combined with the last element of the collection E.

This problem can be solved not only for the `sum()` operation. Reducing recursive functions to several relations could be implemented in a more generic way for supporting every *simple* recursive function/operation.

5.2 Non-Atomic Operations

Currently only atomic operations are supported. So an operation of class C which is called in some time tick t_i can only change the values of this class C. It is not yet possible to specify a constructor for class C which creates an object of Class D and combines objects of class C to objects of class D. Therefore non-atomic operations must be build on top of the atomic operations for seperately creating objects of the two classes and creating the link between those objects.

5.3 Filtering Symmetric Instances

Another issue is that several symmetric instances of the class diagram will be found by the Alloy Analyzer. This likely occurs when the UML primitive types `Boolean` and `Integer` are used. For a class diagram which contains one class with one single boolean instance variable there is at least one other probably uninteressting instance where this instance variable is just toggled. This behaviour gets worse when more boolen instance variables are used. So some criteria must be defined which filters the uninteressting instances and skips them while finding the next fulfilling instance of the class diagram.

6 Conclusion

In this project a prototypical software was developed which uses the presented transformation of UML class diagrams including OCL into relational logic.

The data structure of the class diagram and the method bodies can quite easily be transformed into relational logic. The main challenge is the transformation of arithmetic operations and some aggregate functions of OCL, because they can be called on a *Bag*, which can contain one element multiple times unlike a relation. So the transformation of a bag must be seperated into multiple relations.

Another challenge is that there are several instances of the tranformed relational model which represent an equivalent UML object diagram. Those duplicate instances must be ignored, because symmetries are not interesting for the user of the model checking tools.

The presented transformation is implemented as transformation into the Alloy language in a work-in-progress tool called EAV - Early Analysis Verification. This tool shows instances for a given class diagram in a user defined scope as graphical object diagrams and therefore could help to apply model checking in the everyday software development. The EAV tool can be found at [Vog].

The presented strategy and problems will be further examined in the future to increase the possibilities of model checking for software developement aswell as for the educational teaching of UML and OCL.

Papers

[ABGR] ANASTASAKIS, Kyriakos ; BORDBAR, Behzad ; GEORG, Geri ; RAY, Indrakshi: UML2Alloy: A Challenging Model Transformation. http://dx.doi.org/10.1007/978-3-540-75209-7_30. In: ENGELS, Gregor (Hrsg.) ; OPDYKE, Bill (Hrsg.) ; SCHMIDT, Douglas (Hrsg.) ; WEIL, Frank (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 4735. Springer Berlin / Heidelberg. – ISBN 978–3–540–75208–0, 436-450

[BCG05] BERARDI, Daniela ; CALVANESE, Diego ; GIACOMO, Giuseppe D.: Reasoning on UML class diagrams. In: *Artificial Intelligence* 168 (2005), Nr. 1-2, 70 - 118. http://dx.doi.org/10.1016/j.artint.2005.05.003. – DOI 10.1016/j.artint.2005.05.003. – ISSN 0004–3702

[CCR08] CABOT, J. ; CLARISO, R. ; RIERA, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, 2008, S. 73 –80

[GBR07] GOGOLLA, Martin ; BÜTTNER, Fabian ; RICHTERS, Mark: USE: A UML-Based Specification Environment for Validating UML and OCL. In: *Science of Computer Programming* 69 (2007), S. 27–34

[Hol97] HOLZMANN, G.J.: The model checker SPIN. In: *Software Engineering, IEEE Transactions on* 23 (1997), may, Nr. 5, S. 279 –295. http://dx.doi.org/10.1109/32.588521. – DOI 10.1109/32.588521. – ISSN 0098–5589

[Jac02] JACKSON, Daniel: Alloy: a lightweight object modelling notation. In: *ACM Trans. Softw. Eng. Methodol.* 11 (2002), April, 256–290. http://dx.doi.org/http://doi.acm.org/10.1145/505145.505149. – DOI http://doi.acm.org/10.1145/505145.505149. – ISSN 1049–331X

Books

[BA08] BEN-ARI, Mordechai: *Principles of the Spin Model Checker*. Springer, 2008. – ISBN 978–1–84628–769–5

[Jac12] JACKSON, Daniel: *Software Abstractions: Logic, Language, and Analysis*. Revised. The MIT Press, 2012. – ISBN 978–0262017152

[Kle09] KLEUKER, Stephan: *Formale Modelle der Softwareentwicklung - Model Checking, Verifikation, Analyse und Simulation*. Vieweg+Teubner, 2009. – ISBN 978–3–8348–0669–7

[Som10] SOMMERVILLE, Ian: *Software Engineering*. 9th. Addison Wesley, 2010. – ISBN 978–0137035151

Internet

[Anaa] ANASTASAKIS, Kyriakos (Hrsg.): *UML2Alloy - Reference Manual*. `http://www.cs.bham.ac.uk/~bxb/UML2Alloy/files/uml2alloy_manual.pdf`, Abruf: 30. January. 2012

[Anab] ANASTASAKIS, Kyriakos (Hrsg.): *UML2Alloy - webpage*. `http://www.cs.bham.ac.uk/~bxb/UML2Alloy/index.php`, Abruf: 30. January. 2012

[Ham11] HAMANN, Lars: *A UML-based Specification Environment*. `http://www.db.informatik.uni-bremen.de/projects/USE/`. Version: 2011

[HW12] HORN, Julian ; WENZ, Carl-Phillip: *Model Checking of UML Class Diagrams with OCL using CPL*. `http://www.cpwenz.de/FMC`. Version: 2012

[Jac11] JACKSON, Daniel: *Alloy: A language & tool for relational models*. `http://alloy.mit.edu/alloy/`. Version: 2011

[Kim] KIMMEL, Lars-Erik (Hrsg.): `http://www.cs.hs-rm.de/~lkimm002`

[Obj10] OBJECT MANAGEMENT GROUP: *Object Constraint Language - Version 2.3*. `http://www.omg.org/spec/OCL/2.3.1/`. Version: 2010

[Obj11] OBJECT MANAGEMENT GROUP: *Unified Modeling Language*. `http://www.omg.org/spec/UML/`. Version: 2011

[Vog] VOGT, Patrick (Hrsg.): `http://www.cs.hs-rm.de/~pvogt002`