Detecting Inconsistencies between UML Models Using Description Logic

Ragnhild Van Der Straeten, Jocelyn Simmonds
Vrije Universiteit Brussel, System and Software Engineering Lab
Brussels, Belgium
rvdstrae@vub.ac.be, jsimmond@dcc.uchile.cl

Tom Mens
Service de Génie Logiciel, Université de Mons-Hainaut
Mons, Belgium
tom.mens@umh.ac.be

Abstract

An object-oriented software design is often modelled as a collection of UML diagrams. There is an inherent need to preserve the consistency between these diagrams. Moreover, through evolution these diagrams get modified and can become inconsistent. To be able to preserve their consistency the formalism of description logics is used. *Loom*, a second generation reasoning tool, and RACER, a state-of-the-art reasoning tool, are used as particular description logic reasoning systems. Based on our experience with these tools, we argue that state-of-the-art description logic tools must offer a more extensive query language.

1 Introduction

An object-oriented software design is often specified as a collection of UML diagrams [10]. The different diagrams represent different aspects of the software application. Therefore, there is an inherent risk that the overall specification of the system becomes inconsistent and as such it is necessary to check the consistency between related UML diagrams.

Especially in the context of evolution, it is necessary to ensure that the overall consistency remains preserved. Unfortunately, current-day UML CASE tools provide poor support for maintaining consistency between UML models. This results in less maintainable and comprehensible models.

To solve this problem, we need a formal specification of consistency, and a formal reasoning engine that relies on this specification to detect and resolve inconsistencies between models. To achieve this task we use the formalism of description logic (DL) [1]. As description logic tools we chose *Loom* [9], a second generation tool having an incomplete reasoning algorithm and RACER [8], a state-of-the-art description logic tool having a complete reasoning algorithm. We argue that for our purposes it is necessary that the DL reasoning system consists of an extensive query language. This allows us to specify UML models and consistency rules in a straightforward and generic way.

The structure of the paper is as follows. Section 2 introduces the semantics of UML and presents the translation of the UML metamodel into the DL $\mathcal{ALCQHI}$. In section 3 we discuss some consistency conflicts and show how they can be detected using *Loom* and RACER. Section 4 concludes this paper.

2 Representing UML Metamodel in $\mathcal{ALCQHI}$

The *de facto* modelling language for the analysis and design of object-oriented software applications is UML. The visual representation of this language consists of several diagram types. The UML semantics is described using a metamodel that consists of three views: (1) The abstract syntax is expressed as a class diagram consisting of classes, associations, generalizations and attributes. (2) Well-formedness rules, expressed in the Object Constraint Language (OCL) [10], specify when an instance of a particular language construct is meaningful. (3) The semantics, described in natural language, defines the meaning of a well-formed language construct.

We deliberately confine ourselves to three kinds of UML diagrams: class diagrams representing the static structure of the software application, sequence diagrams representing the behaviour of the software application in terms of the collaboration between different objects, and state diagrams modelling the behaviour of one single object.

We treat two types of consistencies. (1) *Horizontal consistency* indicates consistency between different diagrams within a given model. (2) *Evolution consistency* indicates the consistency between different versions of the same (sub)model [6].

A wide range of approaches for checking consistency has been proposed in the literature ([4], [5], [3], [12], [7]). However, most of these approaches only discuss one type of UML diagram or do not take into account the UML metamodel.

We chose to translate the different concepts of the UML metamodel into the logic $\mathcal{ALCQHI}$ which is supported by state-of-the-art DL reasoning systems such as RACER. *Loom*, on the other hand, supports the description logic $\mathcal{ALCQRIFO}$. For our experiments, we used RACER, version 1.6.7 and *Loom*, version 4.0.

Abstract Syntax Representation For the encoding of this class diagram in $\mathcal{ALCQHI}$, we used the same approach as in [2]. Because in the UML metamodel all associations are binary, they can be translated in the same way as an aggregation in [2], with the extra assertions for an association `ASSOC` between the classes `C1` and `C2`: $\exists$`ASSOC` $\sqsubseteq$ `C1` and $\exists$`ASSOC`$^-$ $\sqsubseteq$ `C2`.

Well-formedness Rules Some of the well-formedness rules of the UML metamodel, specified in OCL, can be expressed on the TBox containing the abstract syntax of the UML metamodel.

As an example, consider the constraint on state diagrams that "A `FinalState` cannot have any outgoing transitions." The `FinalState` is a special kind of state signifying that the entire statemachine has completed. In our TBox, a concept `FinalState` is defined as a subconcept of `State` and of `StateElement`. The specified constraint is added as a number restriction on the role `outgoing`, which represents all outgoing transitions of the particular state and which is defined as a role with domain `State` and range `Transition`. This results in: `FinalState` $\equiv$ `State` $\sqcap$`StateElement` $\sqcap(\leq$ `0 outgoing`).

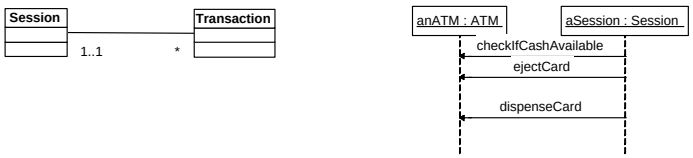

Figure 1: Classless instances conflict

Other well-formedness rules have to be checked using queries on the ABox, containing the user-defined class, sequence and state diagrams. These queries are similar to the queries used for detecting inconsistencies as discussed in the next section.

3 Detecting Inconsistencies

Based on a detailed analysis of all the UML concepts appearing in the three different diagrams, several consistency conflicts are identified and classified. Due to space limitations, we will only discuss the *classless instances* and *reachability* conflicts. The full classification and all the *Loom* queries can be found in [11].

Classless instance arises when an object in a sequence diagram is the instance of a class that does not exist in any class diagram. An example of this conflict is shown in Figure 1, where the object *anATM* is an instance of *ATM* in the sequence diagram on the right side of Figure 1 but this class does not appear in the class diagram on the left side of the same figure. Classes in a class diagram are represented by the concept `class` in our TBox and a class diagram by the concept `classmodel`. `has-classmodel` is a role that contains the associated class diagram of a class.

In this example, the advantage of specifying incomplete knowledge in DL comes into play. From the user-defined diagrams as shown in Figure 1 it is not explicitly known that *ATM* is a class. However, because of the specification that *anATM* is an object and it is an *instance of ATM*, the classification mechanism concludes that *ATM* is a class.

In *Loom* we can use the query language to find all the classes that have no related class diagram:

```
(do-retrieve (?object ?class)
 (:and
  (Object ?object)
  (Instance-of-class ?object ?class)
  (has-classmodel ?class NIL))
 (format t "Classless instance conflict: ~S~%" (get-value ?object 'name)))
```

In the Lisp version of RACER a similar query to the `do-retrieve` statement of *Loom* can be formulated using the loop facility of Common Lisp. However, this implies that no optimization of the query can be performed.

```
(loop for ?object in (concept-instances object) do
   (loop with ?name = (first (rerieve-individual-fillers ?object 'name)) do
      for ?class in (retrieve-individual-fillers ?object 'instance-of-class)
        when (null (retrieve-individual-fillers ?class 'has-classmodel)) do
        (format t "Classless instance conflict: ~S~%" ?name)))
```

However, it would be a big advantage if RACER offers such a query language as part of the RACER server and not of the language Common Lisp as shown in this example.

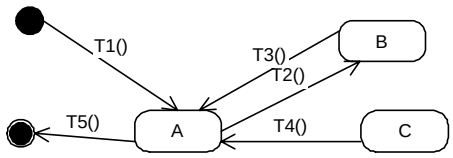

Figure 2: Reachability Problem.

Reachability conflict appears when in a state diagram a particular state is not reachable, which implies that there exist no path at all from the start state to that particular state. For example in Figure 2, the state `C` is not reachable because there is no incoming transition.

In RACER this can be checked by exploiting role hierarchies and role transitivity. In our TBox, we define the transitive role `successor` between two `State`s and the role `direct-successor` which is a subrole of the previous role. The `direct-successor` role specifies which states are directly linked to each other. Fillers for the role `direct-successor` however, must be created every time a transition between two states is created. This happens in the ABox and as such implies that there must exist a rule that introduces the right fillers for the role `direct-successor` every time a `transition` concept is instantiated. Again, this can be specified in the Lisp version of RACER making use of the Lisp programming language.

In *Loom*, we used the query language and Lisp to determine if all the states are reachable.

```
(defun find-path (?top-state ?state1 ?state2)
 (let* ((?cond (ask (:same-as ?state1 ?state2))))
  (if (equal ?cond T)
   (format t "State ~S is reachable~%" ?state2)
   (do-retrieve ()
    (:and

     (Is-container-of ?top-state ?state1)
     (for-some (?transition ?state)
      (:and
       (source ?state1 ?transition)
       (target ?state ?transition)
       (Is-container-of ?top-state ?state)
       (:predcall #'find-path ?top-state ?state ?state2))))
   ()))))
```

4 Conclusion

Evolution and the fact that each UML diagram represents another view on the application are different causes of inconsistencies. To be able to preserve consistency between different UML diagrams, we use the formalism of description logic. As particular description logic reasoning systems, we use *Loom*, a second generation description logic tool and RACER a state-of-the-art reasoning tool.

Several consistency conflicts are identified and classified. Based on two illustrative conflicts, we plead for state-of-the-art DL tools having an extensive query language. This would allow us to specify UML models and consistency rules in a straightforward and generic way.

References

[1] F. Baader, D. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[2] Daniela Berardi. Using DLs to reason on UML class diagrams. In *Proc. Workshop on Applications of Description Logics, Aachen, Germany*, pages 1–11, 2002.

[3] H. Ehrig and A. Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 workshop on graph transformation systems*, pages 77–86, March 2000.

[4] Gregor Engels, Reiko Heckel, and Jochen Malte Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In Martin Gogolla and Cris Kobryn, editors, *Proc. Int'l Conf. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, number 2185 in Lecture Notes in Computer Science, pages 272–286. Springer-Verlag, October 2001. Toronto, Canada.

[5] Gregor Engels, Reiko Heckel, Jochen Malte Küster, and Luuk Groenewegen. Consistency-preserving model evolution through transformations. In *Proc. Int'l Conf. UML 2002*, number 2460 in Lecture Notes in Computer Science, pages 212–227. Springer-Verlag, October 2002.

[6] Gregor Engels, Jochen Malte Küster, Luc Groenewegen, and Reiko Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *Proc. ESEC/FSE 2001.* ACM Press, 2001.

[7] Pascal Fradet, Daniel Le Métayer, and Michaël Périn. Consistency checking for multiple view software architectures. In *Proc. Int'l Conf. ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 410–428. Springer-Verlag, 1999.

[8] Volker Haarslev and Ralf Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Proc. of Seventeenth International Joint Conference on Artificial Intelligence*, pages 161–166. Morgan Kaufmann, 2001.

[9] Robert MacGregor. Inside the LOOM description classifier. *SIGART Bull.*, 2(3):88–92, 1991.

[10] Object Management Group. Unified Modeling Language specification version 1.5. formal/2003-03-01, March 2003.

[11] Jocelyn Simmonds. Consistency maintenance of UML models with description logics. Master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, September 2003.

[12] Aliki Tsiolakis. Semantic analysis and consistency checking of UML sequence diagrams. Master's thesis, Technische Universität Berlin, April 2001. Technical Report No. 2001-06.