Inconsistency Detection between UML Models Using RACER and *nRQL*

Ragnhild Van Der Straeten
SSEL, Vrije Universiteit Brussel,
Pleinlaan 2, Brussels, Belgium
rvdstrae@vub.ac.be

Abstract

An object-oriented software design consists of models that embody a consistent view on the software system under study. We focus on design models expressed in the Unified Modeling Language (UML) and more specifically on class, state machine and sequence diagrams. In this paper, we report on our experiences in using RACER and its New Racer Query Language (*nRQL*) for detecting inconsistencies between models. By means of a simple, yet representative, example we show how different representations in RACER are used for the detection of inconsistencies. As such, the full power of both T-box and A-box reasoning is used in the context of inconsistency detection.

1 Motivation

During software development, models are built representing different views on a software system. Models can also get refined or evolve. In all cases, there is an inherent risk that the overall specification becomes inconsistent. We will focus on design models expressed in the Unified Modeling Language (UML) [13], currently the state-of-the-art modeling language. More specifically, we focus on class diagrams, expressing the static structure of the system and sequence and state machine diagrams specifying (parts of) the behaviour of the system.

A formal foundation for consistency management enables us to specify precise and unambiguous definitions of concepts involved in software modeling. Also CASE tools benefit from a formal approach, preventing an ad-hoc approach to consistency management. To detect inconsistencies between different UML models, we choose for a logic-based approach for the following reasons: (1) The declarative nature of logic is well suited to express the design models which are

also of declarative nature. (2) Logic reasoning engines can deduce implicitly represented knowledge from the explicit knowledge. (3) First-order logic and theorem proving based on the standard inference rules of classical logic have been proposed by several authors for expressing software models and the derivation of inconsistencies from these models resp. ([8], [12]).

However, theorem proving has two major disadvantages. First of all, first-order logic is semi-decidable and secondly, theorem proving is computationally inefficient. This is why we reside to the use of Description Logics (DLs) [1] and more specifically to decidable DLs.

Another important feature of DL systems is that they have an open world semantics, which allows the specification of incomplete knowledge. This is e.g. useful for modeling sequence diagrams which typically specify incomplete information about the dynamic behaviour of the system.

Due to their semantics, DLs are suited to express the static structure of the software application. For example, Calí *et al.* [4] translate UML class diagrams to a DL.

Due to the fact that there is a one-to-one mapping between a certain DL (called $\mathcal{ALCI}_{reg}$) and *converse*-PDL, DLs are also suited to express certain behaviour of a software application.

Several implemented DL systems exist from which we have selected the state-of-the-art *RACER* [11] system. This paper reports on our experiences in using RACER and its query language for inconsistency detection between UML models. We show how different approaches are suited for the detection of different inconsistencies. As well A-box as T-box reasoning is used.

The next section introduces an illustrative example containing some inconsistencies between different UML diagrams. Section 3 explains a representation approach using the T-box reasoning facilities of RACER to check for certain inconsistencies. Section 3 explains how *nRQL* is used for inconsistency checking. Tool support automating the detection of inconsistencies is presented in Section 5. In Section 6, related work is discussed and Section 7 concludes this paper.

2 Examples of Inconsistencies

The example used throughout this paper, is based on the design of an automatic teller machine (ATM), originally developed by Russell Bjork for a computer science course at Gordon University. We express our diagrams in UML version 1.5.

In [15], we made a two-dimensional classification of inconsistencies occurring between UML diagrams. This classification is based on the diagrams affected and on the fact if structural or behavioural aspects are affected. In this section, the *multiplicity conflict*, *navigation conflict* and *invocation consistency* are

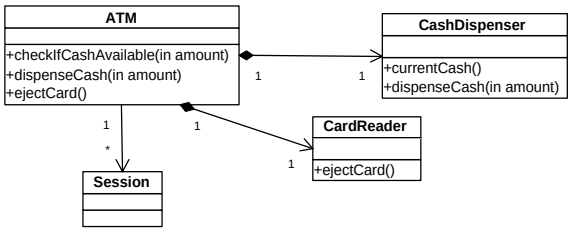

Figure 1: Class diagram for *ATM* example

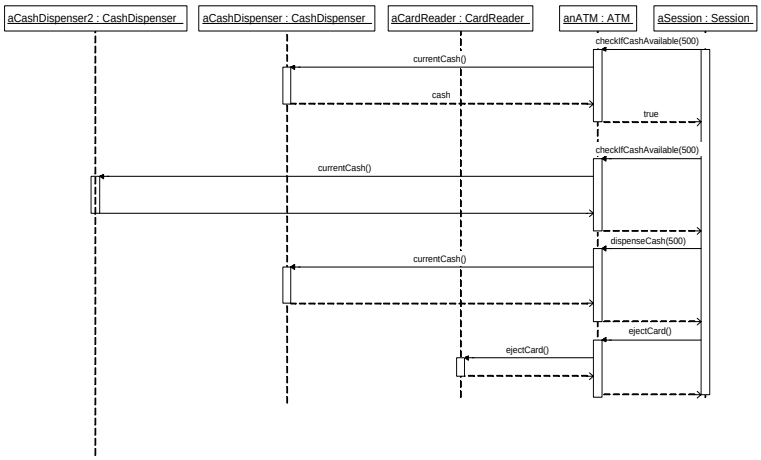

Figure 2: Sequence diagram

discussed.

Consider a class diagram as specified in Figure 1 and a sequence diagram in Figure 2 specifying a particular scenario between objects, i.e. instances of the classes represented in Figure 1.

A first inconsistency arises when an object in a sequence diagram does not respect the *multiplicity restrictions* as imposed by the corresponding class diagrams. In UML a multiplicity consists of a range which has a lower and upper bound. This lower and upper bound indicate the minimum, resp. maximum number of instances to which a certain instance can be connected and as such can interact with. The model consisting of the UML diagrams of Figure 1 and 2 suffers from this inconsistency. In the sequence diagram an instance of *ATM* is associated to two instances of *CashDispenser* and this is in contrast to the one-one association specified between the *ATM* class and the *CashDispenser* class in Figure 1.

Navigation conflict is another inconsistency occurring between this class diagram and sequence diagram. The arrow on the association between the *ATM* and *Session* class indicates that objects of type *ATM* can communicate with objects of type *Session* but not vice versa. In the sequence diagram, messages are

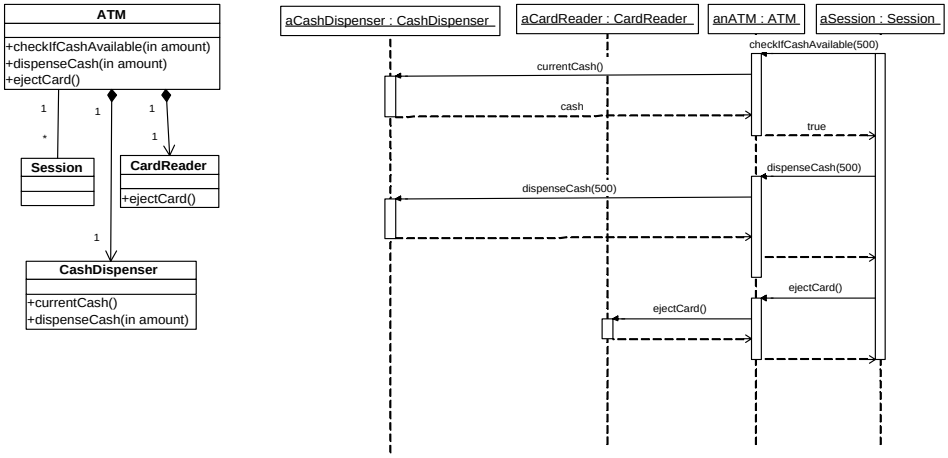

Figure 3: Resolved class and sequence diagram

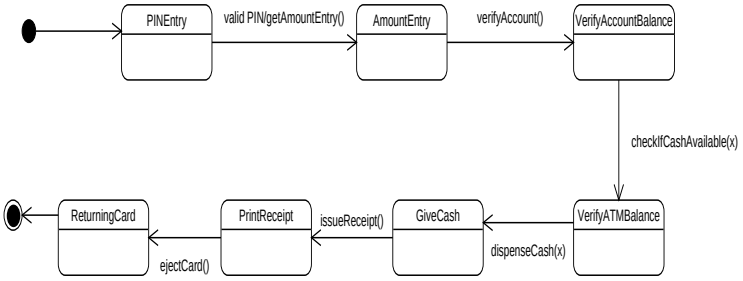

Figure 4: Protocol state machine diagram for the refined *ATM* class

sent from *aSession* object to *anATM* object violating the navigation restriction.

In Figure 3 the resolved class and sequence diagrams are shown. User intervention is necessary to resolve the above discussed inconsistencies. We do not go into detail on how to correct these inconsistencies because this is out of the scope of this paper.

Consider a refinement of the specified behaviour of the *ATM* class by adding behaviour for printing a receipt to this class. Figure 4 shows the protocol state machine for the refined *ATM* class. This *protocol* state machine specifies usage protocols, i.e. the legal transitions an *ATM* object can trigger.

Invocation consistency specifies that the behaviour of the original *ATM* class must be a subset of the behaviour specified for the refined *ATM* class. This consistency is based on Liskov's well-known substitution principle.

The UML model consisting of the corrected sequence diagram in Figure 3 and the state machine diagram in Figure 4 is *invocation inconsistent*. The call sequence *checkIfCashAvailable(), dispenseCash(), ejectCard()* expressed in the sequence diagram of Figure 3 is not valid on the state diagram in Figure 4.

In the next two sections, we will show how different approaches using DL

can help in identifying the presented inconsistencies.

3 Direct DL Detection Approach

The direct approach translates UML user models in terms of atomic concepts and roles of a certain DL. This is also the approach taken by Calí *et al.* [4]. They translate UML class diagrams into the DL $\mathcal{DLR}$ which allows them to reason about class(es) consistency, class equivalence and on class hierarchies.

Class diagrams can be translated in an analogue way into the DL $\mathcal{ALCQHI}_{R^+}(D^-)$ supported by RACER. We use RACER version 1.8.

UML sequence diagrams encompass two different views. First of all, they specify which sequences of messages are legal and secondly, they also represent how objects are connected to each other. We will call this the *interaction*, resp. the *communication view*. The latter view represents an instantiation of the corresponding concepts defined by class diagram(s).

Some inconsistencies can be detected using this approach. Suppose the class diagram of Figure 1 is directly translated into a RACER T-box using the same representation as in [2] and the object view of the sequence diagram of Figure 2 is represented as instances of the concepts defined in the T-box. This will result immediately in a logical inconsistency, because the upper bound of the multiplicity of the role representing the association between the class *ATM* and the class *CashDispenser* is violated by the instances defined in the corresponding A-box. Remark that, because of the open world assumption, the lower bound of the multiplicity of this association cannot be checked in this way. To detect such inconsistencies another approach, explained in Section 4, is necessary.

Also for inconsistencies about sequences of messages as specified by state machine and sequence diagrams, a direct representation of these diagrams into a RACER T-box, is necessary. For the different inconsistencies, only the relevant parts of a model are translated.

Recall the invocation consistency to be checked between the sequence diagram specified in Figure 3 and the state machine in Figure 4. The sequence of operation calls expressed by both diagrams is translated into RACER. The different messages defined in the sequence diagram and the transitions on the state machine are translated to atomic, disjoint concepts which are connected through a binary relation `r`, a kind of accessibility relation as defined in modal logics.

The sequence of messages as expressed by the state machine in Figure 4 is expressed in RACER as follows:

```
(equivalent (and validpin getamountentry) (some r verifyaccount))
(equivalent verifyaccount (some r checkifcashavailable))
(equivalent checkifcashavailable (some r dispensecash))
```

```
(equivalent dispensecash (some r issuereceipt))
(equivalent issuereceipt (some r ejectcard))
(disjoint getamountentry validpin verifyaccount checkifcashavailable
          dispensecash issuereceipt ejectcard)
```

Remark that in this case, a protocol state machine is considered to be complete as opposed to the sequence diagram. This is why the state diagram is translated using "equivalent" statements. The sequence of messages specified by the sequence diagram in Figure 2 is expressed as follows:

```
(implies checkifcashavailable (some r dispensecash))
(implies dispensecash (some r ejectcard))
```

The set of unsatisfiability concepts is `verifyaccount`, `checkifcashavailable` and `dispensecash`. This corresponds to the fact that the behaviour specifications are invocation inconsistent.

The current disadvantage of this representation is that it is not possible with the current DL tools to give proper feedback to the user. The reasoning engine only indicates that a set of concepts is inconsistent. We are not able to deduce from this information which sequences do occur in the state diagram and do not in the sequence diagram.

4 Indirect DL Detection Approach

The UML specification is defined using a metamodeling approach. A language definition normally consists of an abstract syntax, a concrete syntax and semantics. The UML metamodel includes all the concrete graphical notation, abstract syntax and semantics for UML. The UML abstract syntax consists of UML class diagrams. The concrete syntax is informally specified UML notation. Well-formedness rules are constraints on the abstract syntax and as such specify when an instance of a particular language construct is meaningful.

In this approach, the different concepts of the UML metamodel are translated into RACER. The UML metamodel is a collection of class diagrams which are translated into T-box declarations. The user-defined UML diagrams, such as the ones presented in Section 2, are instances of the metamodel and translated into corresponding Abox assertions. This ensures that the user-defined models are consistent with the UML metamodel. If metamodel information is necessary, this approach is taken to identify inconsistenties.

The class diagram in Figure 1 and sequence diagram in Figure 2 are translated to instances of metamodel concepts. For the detection of the navigation inconsistency as specified in Section 2, one *nRQL* query is necessary. This query traverses the UML metamodel searching for an association and corresponding association end(s) by which objects are linked to each other and over which an operation is called and which is not navigable.

```
(retrieve (?object1 ?class2 ?association ?associationend)
 (and (?object1 ?class1 instance-of)
      (?object2 ?class2 instance-of)
      (?object1 ?stimulus sender)
      (?object2 ?stimulus receiver)
      (?stimulus ?operation sends)
      (?class2 ?operation has-feature)
      (?stimulus ?association related-to)
      (?associationend ?association associationends)
      (?class2 ?associationend participant)
      (not (?associationend Navigable))))
```

This approach is also needed to check if the lower multiplicity bound of an association is not violated in a sequence diagram. Two *nRQL* queries are necessary. The first one traverses the UML metamodel searching for objects that are linked to each other by the same association.

```
(retrieve (?object1 ?class2 ?association)
 (and (?object1 ?class1 instance-of)
      (?object2 ?class2 instance-of)
      (?object1 ?stimulus sender)
      (?object2 ?stimulus receiver)
      (?stimulus ?link stimulus-link)
      (?link ?association stimulus-link)))
```

This query is very similar to the query in our previous example. However, the queries are not reusable in the sense that if (part of) a certain query is needed in another more complex query, the body should be copied and pasted into the more complex query. Queries are anonymous in *nRQL* and as such not reusable which is, in our context, recognized as a disadvantage of the query language.

For every list of bindings returned by the aforementioned query, the lower bound of the multiplicity specified for the corresponding association must be checked by a second query. If there are less objects than specified by the lower bound, an inconsistency arises.

```
(retrieve (told-value (lower ?mult-range))
 (and (?association ?assocend associationends)
      (?assocend ?class2 participant)
      (?assocend ?multiplicity has-multiplicity)
      (?multiplicity ?mult-range has-range)))
```

In this query `?association`, `?object1` and `?class2` are constants which are results given by the first query. Those constants must be hardcoded into the query which makes the query not reusable and it must be rewritten for every single case. The solution here would be to allow also constants in the set of variables.

5 Tool Support

The tool chain we set up is called RACOoN (*Ra*cer for *Con*sistency). UML design models are expressed in the UML CASE tool Poseidon [9]. These design models are exported in XMI format, and translated into description logics format using Saxon, an XML translator. The logic code is then asserted into a knowledge base maintained by the RACER logic reasoning engine. This tool chain allows us to specify UML models in a straightforward way and to automate the crucial activity of detecting inconsistencies. We deliberately chose for a tool chain, as opposed to a single integrated tool, to accommodate for the rapid evolution of standards (such as UML, XML and XMI), and to facilitate the replacement of existing tools (e.g., Saxon or Poseidon) by other ones that are more appropriate in a certain context.

6 Related Work

Finkelstein *et al.* [8] explain that consistency between partial models is neither always possible nor is it always desirable. They suggest to use temporal logic to identify and handle such inconsistencies. Grundy *et al.* [10] claim that a key requirement for supporting inconsistency management is the facilities for developers to configure when and how inconsistencies are detected, monitored, stored, presented and possibly automatically resolved. They describe their experience with building complex multiple-view software development tools supporting inconsistency management facilities. Our DL approach is also easily configurable, by adding, removing or modifying logic declarations in the knowledge base.

A wide range of approaches for checking consistency has been proposed in the literature. Engels *et al.* [6] propose a general methodology to deal with consistency problems based on the problem of protocol statechart inheritance. This idea is elaborated in [7] with dynamic meta modeling rules for specifying the consistency conditions in a graphical, UML-like notation. Model transformation rules are used to represent evolution steps, and their effect on the overall model consistency is explored. Ehrig and Tsiolakis [5] investigated the consistency between UML class and sequence diagrams. Class diagrams were represented by attributed type graphs with graphical constraints, and sequence diagrams by attributed graph grammars. As consistency checks between class and sequence diagrams only existence, visibility and multiplicity checking were considered. In [14] the information specified in class and statechart diagrams is integrated into sequence diagrams. The information is represented as constraints attached to certain locations of the object lifelines in the sequence diagram. The supported constraints are data invariants and multiplicities on class diagrams and state and guard constraints on state diagrams. The above mentioned approaches only deal with very specific and a limited number of inconsistencies.

In Calí *et al.* [4] user-defined class diagrams are translated in a description logic that can express n-ary relations. We are inspired by this translation to translate our UML profile. The inconsistencies treated in this work are different from the types of inconsistency we treat. To be able to check our inconsistency categories meta-level knowledge is needed which is not included in their translation. In the same context, [3] proves that reasoning on UML class diagrams is EXPTime hard.

7 Conclusion

In this paper, we report on our experiences with the state-of-the-art DL tool RACER and its query language in the context of inconsistency detection between UML models. We argue that different DL representations of UML diagrams can be used for identification of different inconsistencies. This approach makes use of the reasoning tasks inherent to DLs. However, *nRQL* queries are not reusable because they are anonymous and do not allow constants as variables. This minor disadvantages of *nRQL* is recognized in this paper.

In [15], we presented a conceptual classification of different inconsistencies between UML models. As future work, we want to classify those different inconsistencies based on the different DL detection approaches.

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] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams is EXPTIME-hard. In *Proc. of Int'l Workshop on Description Logics, Rome, Italy*, 2003.

[4] Andrea Calí, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Reasoning on UML class diagrams in description logics. In *Proc. of IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD 2001)*, 2001.

[5] 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.

[6] 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*, number 2185 in Lecture Notes in Computer Science, pages 272–286. Springer-Verlag, October 2001. Toronto, Canada.

[7] 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.

[8] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, LNCS, pages 84–99. SpringerVerlag, 1993.

[9] Gentleware. Poseidon, http://www.gentleware.com/products/poseidonpe.php3, March 18 2004.

[10] John C. Grundy, John G. Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.

[11] Volker Haarslev and Ralf Möller. RACER system description. In *Int'l Joint Conf. Automated Reasoning (IJCAR 2001)*, 2001.

[12] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

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

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

[15] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between UML models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proc. Int'l Conf. UML 2003*, volume 2863 of *LNCS*, pages 326–340. Springer-Verlag, 2003.