

Maintaining Consistency between UML Models with Description Logic Tools

Tom Mens¹, Ragnhild Van Der Straeten², and Jocelyn Simmonds¹

¹ Programming Technology Lab

Department of Computer Science, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

`tom.mens@vub.ac.be`, `jsimmond@dcc.uhile.cl`

² Systems and Software Engineering Lab

Department of Computer Science, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

`rvdstrae@vub.ac.be`

Abstract. In current-day object-oriented software development, the software design (typically modelled as a collection of UML diagrams) is an essential aspect. As a result, future object-oriented legacy systems will encounter the non-negligible problem of design inconsistencies in its various forms. Unfortunately, in its generality, detecting these design inconsistencies is an undecidable problem. Therefore, we suggest to restrict to description logic, a decidable fragment of first-order predicate logic, to specify and detect inconsistencies between UML models. With concrete experiments in Loom and Racer, two tools that provide support for some variant of description logic, we intend to show the feasibility of this approach, and to identify the potential limitations of description logics for the purpose of maintaining design consistency in evolving legacy code.

1 Introduction

An object-oriented software design is typically specified as a collection of UML models, typically a combination of many different class diagrams, sequence diagrams and state diagrams. In object-oriented legacy systems, this essentially gives rise to two different kinds of inconsistency problems: (1) the design may be internally inconsistent or incomplete; (2) the design may be “out of sync” with the source code (*c.f. design erosion*).

Even if the legacy system by itself is in a consistent state, there is a significant risk that re-engineering activities (such as refactoring and restructuring) give rise to unexpected inconsistencies. Hence, it is essential to provide a means to check and maintain consistency of object-oriented designs. Unfortunately, current-day UML CASE tools provide poor support for maintaining semantic consistency between evolving UML models.

To counter this problem, there is first of all a need to specify the consistency between different UML models. A concrete motivating example of an inconsistency between a UML statechart and a UML sequence diagram is illustrated in

Figure 1. The sequence diagram does not respect the order of operations imposed by the statechart.

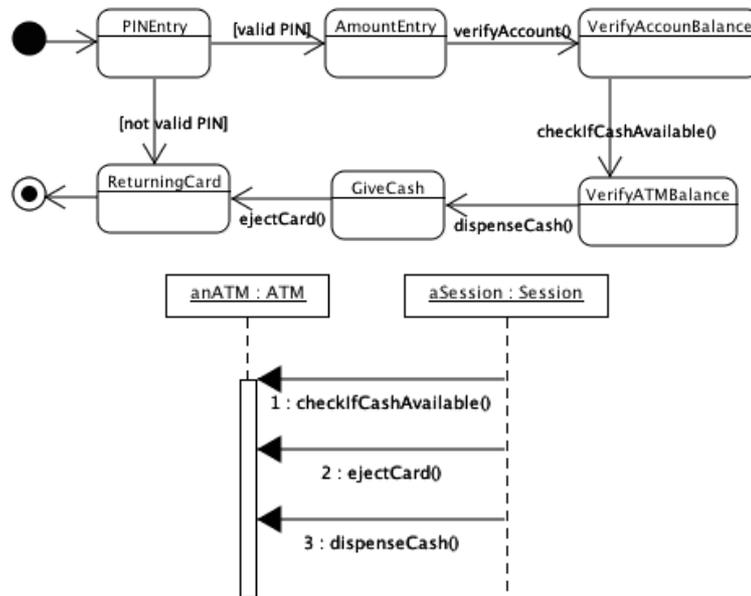


Fig. 1. Example of an inconsistency between the statechart of the class `ATM` and a sequence diagram: the operations `dispenseCash()` and `ejectCard()` appear in a different order in both diagrams.

Description logic (DL) seems to be a suitable formalism for the purpose of maintaining consistency between UML models and diagrams, because it contains 5 reasoning tasks that can directly be used to achieve this: subsumption, instance checking, relation checking, concept consistency and knowledge base consistency. As such, we propose to use DL to partially automate this crucial activity in the software re-engineering process, thus increasing the reliability of the process.

2 Description logic

2.1 Motivation

The main motivation for resorting to description logic is because it is a decidable fragment of first-order logic that possesses sound and complete reasoning mechanisms [1]. In his PhD, Kim Mens [5] tried to maintain consistency between a software architecture and its corresponding source code using a logic programming language. Unfortunately, the algorithm to check consistency turned out to

be very inefficient. Moreover, due to the fact that a full-fledged programming language was used, there was no guarantee whatsoever concerning the decidability or completeness of the consistency algorithm. Description logic solves the latter problem because it is a decidable and complete fragment of first-order predicate logic. We also have good hope that the efficiency will be higher.

Many concepts in object-oriented programming languages originate from frame-based systems. Description logic is based on the same ideas as these systems and provides them with precise semantics. DL has already proven its worth in knowledge representation and reasoning. A DL consists of a description language, a knowledge specification language and, most importantly, automatic reasoning procedures. The latter allows to reason about the consistency of knowledge bases. By analogy, we hope to use them to reason about design model consistency as well. Indeed, due to its semantics, DL is well-suited to express the design of a software system. For example, [2] translated UML class diagrams to the description logic \mathcal{DLR} .

To validate the feasibility of DL for our purposes, we will evaluate two different tools: *Loom* and *Racer*. *Loom* [4] provides a complete environment for the development of knowledge based systems. Although Loom has an incomplete classification algorithm, we can restrict ourselves to a subset that is complete. The *Racer* [3] system is a knowledge representation system that implements a highly optimized tableau calculus for a very expressive description logic. It has a complete classification algorithm but its query language is less expressive than the one of *Loom*. Based on a number of experiments we will choose the tool that is most suited for our needs.

2.2 Representing UML models and their evolution

In order to be able to reason about UML in DL, we need to have an explicit logic representation of the UML metamodel. Secondly, each specific UML diagram for which we want to check consistency needs to be represented in DL as an instance of this metamodel.

Rather than doing everything manually, we envision an automated translation from UML models into DL. As a CASE tool we will use Poseidon to express UML models. These can be exported into XMI format. Using an XSLT processor tool, these XMI files can be translated into the appropriate DL format.

Note that we will also need to make some small changes to the UML metamodel itself, because in its current version it is inadequate to check model consistency and to support model evolution. For details about this, we refer to [6].

Finally, to be able to express model evolution and model consistency, we need to distinguish two types of consistency between UML models. *Horizontal consistency* indicates consistency between different models within the same version. *Evolution consistency* indicates the consistency between different versions of the same model. In the UML metamodel, both kinds of consistency can be expressed by defining two stereotypes for the *Trace* metaclass: \llcorner HorizontalTrace \lrcorner and \llcorner EvolutionTrace \lrcorner .

3 Example in *Loom*

3.1 Specifying models and their evolution

The UML metamodel can be specified in *Loom* in terms of atomic concepts and roles as well as more complex descriptions that can be built from them with concept constructors. As an example we give the specification of the metaclasses *Association* and *AssociationEnd* below:

```
(LOOM:defrelation Is-participant-of
  :domain Class
  :range AssociationEnd)
(LOOM:defrelation Has-participant
  :is (:inverse Is-participant-of))
;Concept ASSOCIATION
(LOOM:defconcept Association
  :is (:and ModelElement
    (:all Association-associationEnd AssociationEnd)
    (:at-least 2 Association-associationEnd)
    (:all Association-link Link)))
;Concept ASSOCIATIONEND
(LOOM:defconcept AssociationEnd
  :is (:and ModelElement
    (:at-most 1 Has-participant)
    (:at-least 1 Has-participant)
    (:at-most 1 AssociationEnd-association)
    (:at-least 1 AssociationEnd-association)
    (:exactly 1 Has-multiplicity)
    (:all AssociationEnd-linkEnd LinkEnd))
  :roles (aggregation ordering isNavigable visibility))
```

As can be seen from this example, each UML metaclass is translated into a *Loom* concept, while the different meta association ends and attributes of a metaclass are translated into *Loom* roles.

To specify concrete UML diagrams in *Loom*, we can specify individuals and assertions about these individuals. The following *Loom* code specifies how we can create the class *Document*, two objects *anASCII* and *aPrinter* as well as an association (*Document-Printer*) between the *Document* and *Printer* class and one of its association ends (*Document-end2*) on the side of *Document*:

```
(create 'Document 'Class)
(create 'anASCII 'Object)
(create 'aPrinter 'Object)

(tellm (:about Document-Printer
  (name Document-Printer)
  (In-namespace Class-Diagram)
  (Association-associationEnd Document-end2)
  (Association-associationEnd Printer-end1)))
```

```
(tellm (:about Document-end2
  (name Document-end2)
  (Has-multiplicity m3)
  (Has-participant Document)
  (In-namespace Class-Diagram)
  (AssociationEnd-association Document-Printer)))
```

To be able to check horizontal consistency between different diagrams within the same version, and evolution consistency between different versions of the same diagram, we needed to add trace relationships explicitly. Below we show an example of how to specify an evolution trace between two subsequent versions of a sequence diagram in *Loom*:

```
(create 'SD_ASCII_1-SD_ASCII_2 'EvolutionTrace)
(tellm (:about SD_ASCII_1-SD_ASCII_2
  (name SD_ASCII_1-SD_ASCII_2)
  (Supplier SequenceDiagram_ASCII_1)
  (Client SequenceDiagram_ASCII_2)))
```

3.2 Detecting inconsistencies

To detect inconsistencies between evolving UML diagrams we can use *Loom*'s query processor. For example, to detect that a new class has been added in the evolved class diagram, we can specify the following evolution consistency rule:

```
(retrieve ?x
  (:and (Class ?x) (the-prev-ver ?x NIL)
  (In-namespace ?x Class-Diagram)))
```

Whenever inconsistencies are detected we can also provide rules to resolve them. For example, the rules below can be used when an evolved sequence diagram uses an operation (in a message call) that is not yet present in the corresponding class defined in a class diagram. If this is the case, the operation needs to be added to this class in the class diagram, and the operation needs to be linked to the specific message call in the sequence diagram to ensure that both remain consistent.

```
(do-retrieve (?x)
  (:and
    (ModelElement ?x)
    (In-namespace ?x NIL))
  (tellm (In-namespace ?x Class-Diagram-2)))

(do-retrieve (?x ?y ?z ?a ?b)
  (:and
    (Stimulus ?x)
    (Received-by ?x ?y)
    (Instance-of-class ?y ?z)
    (Initiates ?x ?a))
```

```
(CallAction-operation ?a ?b)
(Is-owned-by ?b NIL))
(tellm (Is-owned-by ?b ?z)))
```

4 Conclusion

Based on a number of already performed small-scale experiments we are convinced that DL tools can be very helpful in detecting design inconsistencies and resolving these inconsistencies semi-automatically. However, we still need to explore the limitations of the various DL tools that are available, as well as the inherent limitations of the DL formalism itself.

We conclude with a list of open questions that remain to be answered:

- Which kinds of inconsistencies can we detect?
- For which kinds of UML diagrams can we provide consistency support?
- Which kind of automated support for conflict resolution can we provide?
- Can we use DL to provide formal proofs about the consistency?
- Can we apply refactorings directly in the DL representation, and can we use DL to reason about these refactorings, e.g., to provide better conflict resolution?
- Is the approach scalable to large legacy systems?
- Can the approach be used to deal with consistency between UML models and source code as well? If yes, how does it fit in the Model Driven Architecture (MDA) approach?
- How is the proposed approach and formalism related to other approaches for maintaining consistency?

References

1. F. Baader, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
2. A. Calí, D. Calvanese, G. De Giacomo, and M. 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.
3. V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In B. Nebel, editor, *Proceedings of Seventeenth International Joint Conference on Artificial Intelligence, IJCAI-01, 4-10 August, 2001, Seattle, Washington, USA*, pages 161–166. Morgan Kaufmann, 2001.
4. R. MacGregor. Inside the loom description classifier. *SIGART Bull.*, 2(3):88–92, 1991.
5. K. Mens. *Automating Architectural Conformance Checking by Means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 2000.
6. J. Simmonds. Consistency maintenance of uml models with description logic. Master's thesis, Vrije Universiteit Brussel, September 2003. In progress.