Using Description Logic to Maintain Consistency between UML Models

Ragnhild Van Der Straeten[1], Tom Mens[2], Jocelyn Simmonds[1], and Viviane Jonckers[1]

[1] Systems and Software Engineering Lab
Department of Computer Science, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
rvdstrae@vub.ac.be, jsimmond@dcc.uchile.cl, viviane@info.vub.ac.be
[2] Service de Génie Logiciel, Université de Mons-Hainaut
6, Av. du Champs de Mars, 7000 Mons, Belgium
tom.mens@umh.ac.be

Abstract. A software design is often modelled as a collection of UML diagrams. There is an inherent need to preserve consistency between these diagrams. Moreover, through evolution those diagrams get modified leading to possible inconsistency between different versions of the diagrams. State-of-the-art UML CASE tools provide poor support for consistency maintenance. To solve this problem, an extension of the UML metamodel enabling support for consistency maintenance and a classification of inconsistency problems is proposed. To achieve the detection and resolution of consistency conflicts, the use of description logic (DL) is presented. DL has the important property of being a decidable fragment of first-order predicate logic. By means of a number of concrete experiments in *Loom*, we show the feasibility of using this formalism for the purpose of maintaining consistency between (evolving) UML models.

1 Introduction

A software design is typically specified as a collection of UML diagrams [17]. Because different aspects of the software system are covered by many different UML diagrams, 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 design evolution, it is necessary to ensure that the overall consistency is preserved. Hence, it is important to provide a means to detect and resolve the inconsistencies between related UML diagrams and models.

A first type of consistency, indicating consistency between different models within the same version, is called *horizontal consistency*. *Evolution consistency* indicates consistency between different versions of the same model.

Unfortunately, current-day UML CASE tools provide poor support for maintaining consistency between (evolving) UML models. This results in less maintainable and comprehensible models.

To counter this problem, there is first of all a need to specify the consistency between (evolving) models in a formal and precise way. The current UML metamodel [17] provides poor support for consistency preservation and software evolution, e.g. versions are not supported. Therefore, the first contribution of this paper is to show how such support can be integrated in the UML metamodel with only some minor additions.

Based on the different kinds of inconsistencies observed between UML models, a *classification of inconsistencies* is proposed. To be able to detect and resolve inconsistencies, both a formal specification of model consistency and a formal reasoning engine relying on this specification is needed . Therefore, in this paper we propose to use the formalism of description logic (DL) [2].

DL is a two-variable fragment of first-order predicate logic that offers a classification task based on the subconcept-superconcept relationship. In most description logics, this classification task is decidable and complete. While the satisfiability problem is undecidable in first-order logic, most DLs have decidable inference mechanisms. These inference mechanisms allow to reason about the consistencies of knowledge bases specified by DLs, as such these mechanisms enable the identification and resolution of consistency problems.

As description logic tool we chose *Loom* [16] because of its extensive query language and associated production rule system. This allows us to specify UML models, their evolution, consistency rules and also design improvements in a straightforward way. As such, the crucial activity of detecting and resolving design inconsistencies can be partially automated, thus increasing the maintainability of the software.

In the next section the developed UML profile for model consistency is explained. Before introducing the running example used in this paper in section 4, a possible classification of inconsistencies is given in section 3. A motivation for the use of description logic is given in section 5. Section 6 discusses some experiments and section 7 gives a summary of related work. We conclude in section 8.

2 UML Profile for Model Consistency

With the current version of UML [17] we are not able to check model consistency and to support model evolution. It must be possible to express the existence of different versions of a model. Therefore, a UML profile for model consistency is developed.

We deliberately confine ourselves to three kinds of UML diagrams: class diagrams, sequence diagrams and state diagrams. Consequently, our UML profile consists of subsets of the *Core*, *Model_Management*, *Common_Behaviour*, *Collaborations* and *State_Machines* packages of the UML metamodel. An overview of the used subset of the *Core* package is shown in Figure 1. Remark that the metaclass *Modelelement* is not displayed for reasons of clarity.

In our UML profile, horizontal as well as evolution consistency can be expressed by defining two stereotypes for the *Trace* metaclass: *HorizontalTrace* and

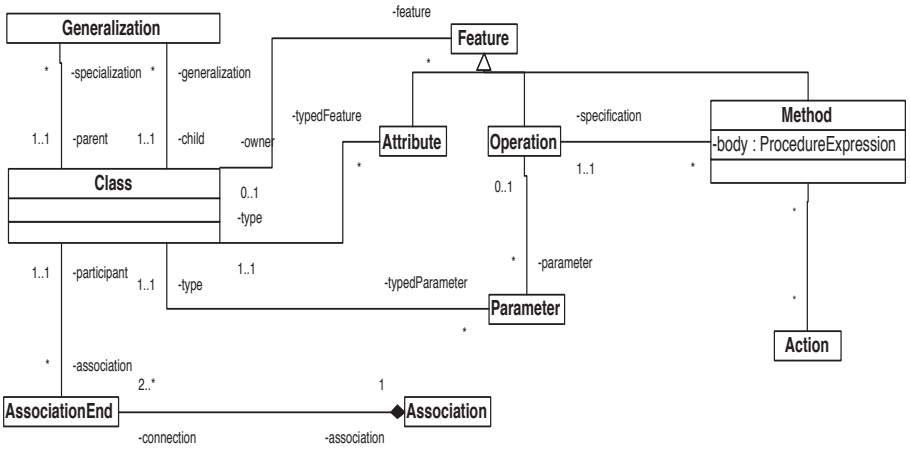

Fig. 1. Subset of UML Core Package

EvolutionTrace (see Figure 2). To this extent, we also need a notion of *Versioned-Model*, which is a stereotype for the *Model* metaclass. It adds a tag-value pair (`version`,*Integer*) to denote the model version. The term *vertical consistency* also exists, but is not treated here. It is used to specify the relationship between a model and a refinement of this model that includes more specific details. In the UML metamodel, the *Refinement* relationship, which is a stereotype of the *Abstraction* metaclass, can be used for this purpose.

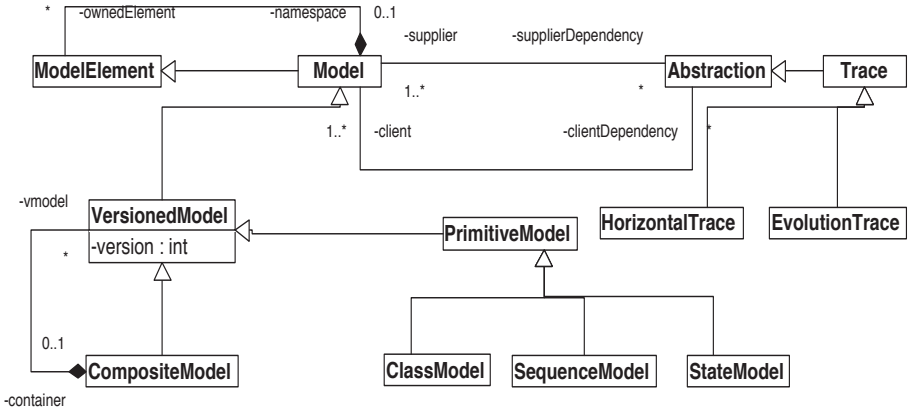

Fig. 2. Subset of UML Model Management Package

To specify the kind of models that can be related by horizontal or evolution consistency, the *Model* metaclass in the package *Model_Management* is stereo-

typed to distinguish between primitive models and composite models. *PrimitiveModel* is a stereotyped *Model* that can be specialised (stereotyped) further into *ClassModel*, *SequenceModel* and *StateModel*. *CompositeModel* is a stereotyped *Model* that is a container of *VersionedModels* all belonging to the same version. In order to keep track of the models belonging to a *CompositeModel*, a tag-value pair (`vmodel`, *Set(VersionedModel)*) is introduced. For *VersionedModel*, a tag-value pair (`container`, *CompositeModel*) is needed. Both tag-value pairs are represented together as a bidirectional association in Figure 2.

Further well-formedness rules must be specified for the newly introduced *Model* and *Trace* stereotypes and tag-value pairs:

- A *CompositeModel* contains at least one *PrimitiveModel* (either directly or indirectly).
- All *VersionedModel*s contained in the same *CompositeModel* should have the same *version number*.
- A *HorizontalTrace* can only be specified between *VersionedModels* belonging to the same *CompositeModel*.
- An *EvolutionTrace* can only be specified between different versions of the same *VersionedModel*.

We also need a subset of the packages *Common_Behaviour* and *Collaborations* as depicted in Figure 3. To be able to indicate if an *AttributeLink* is accessed or updated, the *UninterpretedAction* metaclass is stereotyped into *UpdateAction* and *AccessAction*. A slightly different but equally valid alternative is proposed in [14]. To keep track of the *AttributeLink* that is accessed or updated by an instance of *AccessAction* or *UpdateAction*, different kinds of tag-value pairs are introduced. The pair (`updateAttr`, *AttributeLink*) for stereotype ≪Update≫ keeps track of the updated attribute link. These pairs are again represented as associations in Figure 3.

Finally, a subset of the *State_Machines* packages will be needed. As a first approximation, we ignore the metaclasses *SynchState*, *SubMachineState*, *StubState*, *SignalEvent* and *TimeEvent* and the meta attribute *isConcurrent* of metaclass *CompositeState*.

This extended subset of the UML metamodel covers the basic notions of class, sequence and state diagrams and permits the detection and resolution of inconsistencies.

3 Classification of Inconsistency Conflicts

In this section, we propose a classification of inconsistency conflicts that can be observed between (evolving) UML class, sequence and state diagrams. The proposed classification is based on two dimensions.

The first dimension indicates whether structural or behavioural aspects of the models are affected. Structural inconsistencies arise when the structure of the system is incomplete, incompatible or inconsistent with respect to existing behaviour.

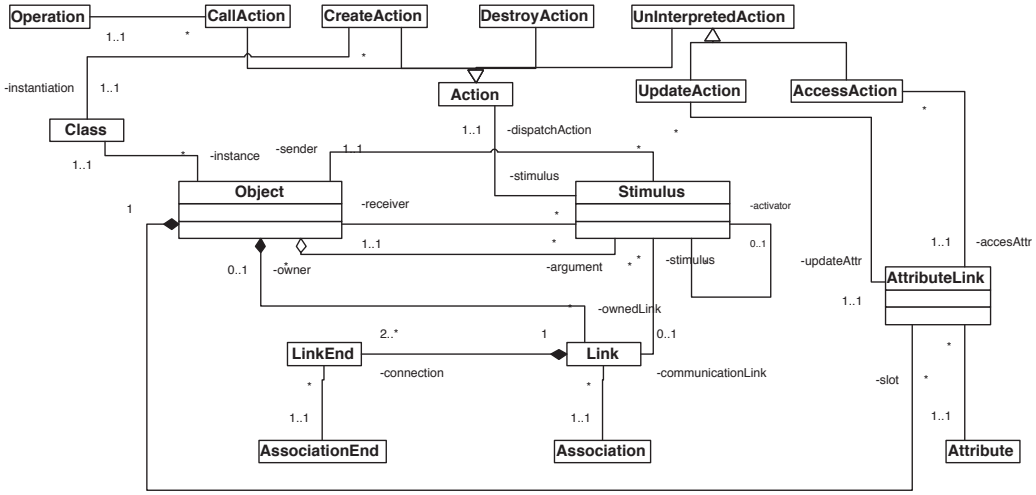

Fig. 3. Subset of UML Common_Behaviour and Collaborations Packages

The second dimension considers the type of affected model. For this purpose, class diagram, sequence diagram and state diagram are classified following the four layers of the meta-tower of the MDA standard[3], i.e. **Instance**, **Model**, **Meta-Model** and **Meta-Meta-Model**. A class diagram belongs to the **Model** level because the model elements it represents (more specifically, classes and associations) serve as definitions for instances (more specifically, objects, links, transitions and events) in sequence and state diagrams which belong to the **Instance** level. Conflicts can occur at the **Model** level, between the **Model**

	Behavioural	**Structural**
Model-Model		dangling (type) reference inherited association conflict
Model-Instance	incompatible definition	**instance definition missing**
Instance-Instance	invocable behaviour conflict observable behaviour conflict **incompatible behaviour conflict**	disconnected model

Table 1. Two-dimensional inconsistency conflict table

and **Instance** level, or at the **Instance** level. The classes of observed conflicts are listed in table 1. Because of space limitations, only the *instance definition missing* and *incompatible behaviour* conflicts are detailed here.

– **Instance definition missing** occurs when an element definition does not exist in the corresponding class diagram(s). This class of conflicts represents structural conflicts between class, sequence and state diagrams because the structure of the software system as specified in the class diagram is incomplete or incompatible with respect to existing instances. These conflicts can be caused by removing elements from a diagram or having not yet included the necessary element(s). This class of conflicts represents the following conflicts:
  - *Classless instance* arises when an object in a sequence diagram is the instance of a class that does not exist in any class diagram.
  - *Classless statechart* arises when the state diagram is associated to a class that does not exist in any class diagram.
  - *Dangling (inherited) feature reference* arises when a stimulus, event, guard or action references an attribute or operation that does not exist in the corresponding class (or its ancestors).
  - *Dangling (inherited) association reference* arises when a certain link (to which a stimulus (or stimuli) is related) in a sequence diagram is an instance of an association that does not exist between the classes of the linked objects (or between the ancestors of these classes).
– **Incompatible behaviour** conflicts indicate conflicting behaviour definitions between state diagram(s) and sequence diagram(s). More particularly this conflict arises when the ordered collection of stimuli received by an object in a sequence diagram does not exist as a sequence of events in the state diagram of that class.

Concrete examples of the above conflicts are shown in the next section.

4 Running Example

To illustrate how description logic can help us to maintain the consistency between evolving UML models, we introduce a running example. Figure 4 shows a class diagram containing a *Document* class hierarchy together with a *Printer* and *Previewer* class. Three types of documents exist: *ASCIIDoc*, *PSDoc* and *RTFDoc*. *Document* itself is an abstract class. The class diagram also contains a *Printer* and *Previewer* class allowing the user to print or preview a document.

The left-hand side of Figure 5 depicts a sequence diagram specifying the printing behaviour of an ASCII document. An instance of *ASCIIDoc* receives a `print` message, as a consequence the content of the document is converted and sent to the printer. The ASCII content needs to be converted because in this example, the *Printer* is supposed to be a PostScript printer. We have similar sequence diagrams for printing a *RTFDoc* and a *PSDoc*. In the case of a *PSDoc*, the `convert` message is obviously unnecessary. There are also three similar sequence diagrams for previewing documents.

Now, consider an evolution of this sequence diagram, as illustrated in Figure 5. The basic evolution step consists of introducing a Visitor design pattern [13]. The overall idea of the Visitor pattern is to gather related operations and

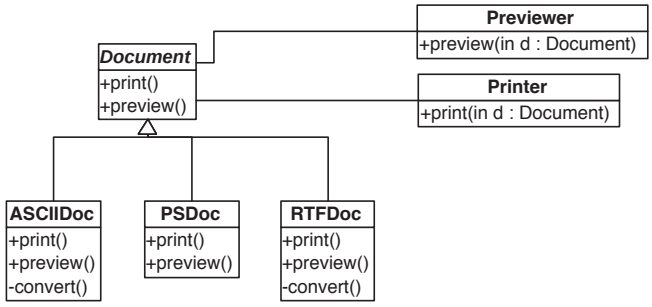

Fig. 4. Class diagram version 1

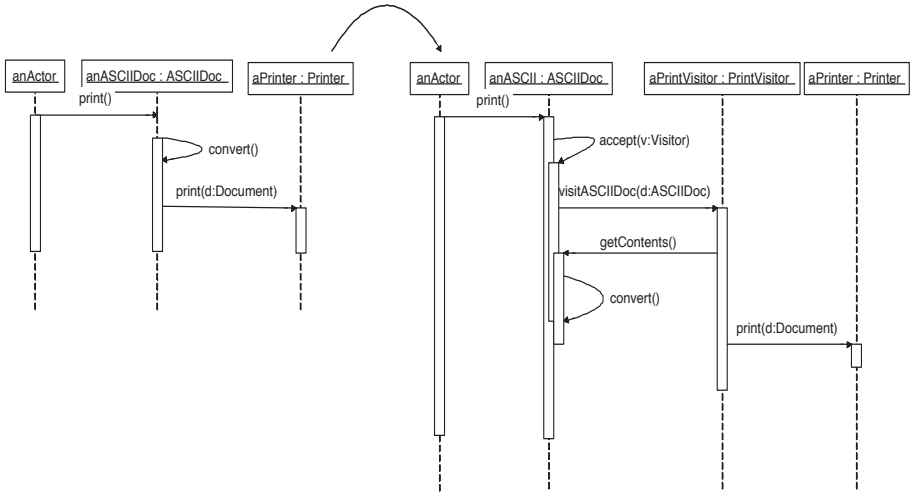

Fig. 5. Sequence diagram version 1 and its evolved version 2

localize them in a visitor. For this purpose, a *PrintVisitor* class is introduced together with an `accept` method in all the classes of the *Document* hierarchy. The `accept` method calls a specific visit method for each different kind of `print` functionality. For example, in the evolved sequence diagram on the right of Figure 5, `accept` invokes the method `visitASCII` on the *PrintVisitor* class to print the contents of the document. A similar evolution is needed for printing PS and RTF documents.

With respect to consistency, we would like to know whether the original class diagram of Figure 4 is still consistent with the evolved sequence diagram on the right of Figure 5. Intuitively, one can see that the answer is negative. The following inconsistencies with the original class diagram can be derived from the evolved sequence diagrams:

1. The sequence diagram refers to an object of a class *PrintVisitor* that does not belong to the class diagram. This is an example of a *classless instance*.

2. As a consequence of the first conflict, the *PrintVisitor* class does not define the methods `visitPSDoc`, `visitASCIIDoc` and `visitRTFDoc`. In addition, none of the subclasses of *Document* define an `accept` method. These are occurrences of *dangling feature reference*.
3. The class *PrintVisitor* does not have an association with the *Printer* class. Such an association is needed, because a `print` message is sent from *PrintVisitor* to *Printer*. This is an example of the *dangling association reference* conflict.
4. Finally, from the original class diagram and the evolved sequence diagram a design improvement can be deduced. Every subclass of *Document* has a `print` method in the original class diagram. From the sequence diagrams we observe that the `print` message has the same implementation for each *Document* subclass: it simply redirects responsibility to the visitor by invoking the `accept` method. This allows us to derive that the `print` method can be **pulled up** (pull up method refactoring [11]) and as such disappears in the subclasses of *Document* but becomes concrete in the *Document* class.

If all the inconsistencies and design improvements are addressed, the class diagram given in Figure 7 can be derived.

After the introduction of the evolved sequence diagrams, a state diagram for *ASCIIDoc* is introduced. The state machine in Figure 6 shows the substates of the `Print` superstate. The `Print` superstate is entered if an `accept` event (raised by the invocation of the `accept()` operation) occurs and it implies that the `Accepting` state is entered. A `convert` event (raised by the invocation of the `convert()` operation) causes a transition to the `Converting` state and finally, the `getContents` event (raised by the invocation of the `getContents()` operation) causes a transition to the `Visited` state.

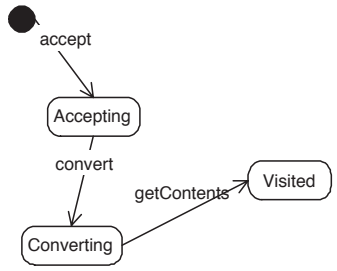

Fig. 6. State diagram inconsistent version

In this case, we would like to know whether the part of the state diagram as shown in Figure 6 is consistent with the sequence diagram on the right of Figure 5. The evolved sequence diagram of Figure 5 and the part of the state diagram in Figure 6 also conflict because the order in which the messages are received by the object `anASCII` is different from the order in which the corresponding events occur in the state diagram.

In the remainder of this paper, we will show how to use the formalism of description logic to automatically detect and resolve these kinds of inconsistencies between different kinds of UML diagrams.

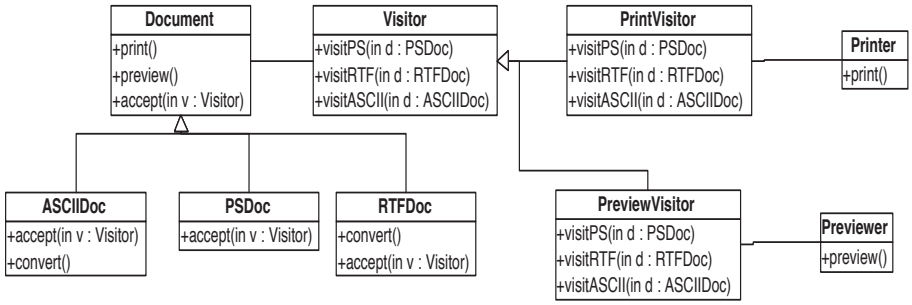

Fig. 7. The evolved class diagram, i.e. class diagram version 2

5 Description Logic

Description Logics (DLs) are a family of knowledge representation formalisms. These formalisms allow us to represent the knowledge of the world by defining the *concepts* of the application domain and then use these concepts to specify properties of individuals occurring in the domain. The basic syntactic building blocks are atomic concepts (unary predicates), atomic roles (binary predicates) and individuals (constants). The expressive power of the language is restricted. It is a two-variable fragment of first-order predicate logic and as such it uses a small set of constructors to construct complex concepts and roles.

The most important feature of these logics is their reasoning ability. This reasoning allows us to infer knowledge that is implicitly present in the knowledge base. Concepts are classified according to subconcept-superconcept relationships, e.g. `ASCIIDoc` is a `Document`. In this case, `ASCIIDoc` is a subconcept of `Document` and `Document` is the superconcept of `ASCIIDoc`. Classification of individuals provides useful information on the properties of individuals e.g., if an individual is classified as an instance of `ASCIIDoc`, we infer that it is also a `Document`. Instance relationships may trigger the application of rules that insert additional facts into the knowledge base e.g., the specification of a rule stating that all `ASCIIDocs` have a `Printer`, has as result that an individual known to be an `ASCIIDoc`, is also known to have a `Printer`. The classification reasoning task is one of the main reasons why we resort to DL.

Another important feature of DL systems is that they have an open world semantics, which allows the specification of incomplete knowledge. Due to their semantics, DLs are suited to express the design structure of a software application. For example, Calí [5] *et al.* translated UML class diagrams to the description logic $\mathcal{DLR}$.

Even with all the expressive power of first-order logic, it is not possible to define the transitive closure of a relation in first-order logic. In [4] this is also recognized as a deficiency of OCL. The well-formedness rules of the UML metamodel which are expressed in OCL make heavy use of additional operations to navigate over the metamodel. These operations are often recursive and this could be avoided if it was possible to express transitive closure in OCL [4]. In most DLs it is however possible to define the transitive closure of a role.

Several implemented DL systems exist (e.g., Loom, Classic, and so on). Below we explain one such system, and argument why we have selected it for carrying out our experiments.

5.1 Translation of UML Profile into Loom

The *Loom* system offers reasoning facilities on concepts and individuals for the DL $\mathcal{ALCQRIFO}$. This logic extends the basic description logic $\mathcal{ALC}$ with qualified number restrictions on roles, inverse roles, role hierarchy and nominals. The reason why we have chosen *Loom*, also its distinguishing feature from other DL systems, is the incorporation of an expressive query language for retrieving individuals and its support for rule-based programming.

Our UML profile will 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 translation into *Loom* of the meta association *vmodel*. Meta associations are translated into *Loom* as roles between concepts. The association *vmodel* between a *CompositeModel* and a *VersionedModel* is translated into the role `vmodel` with as domain the concept `CompositeModel` and as range `VersionedModel`.

```
;Relation compositeModel-versionedModel
(LOOM:defrelation vmodel
 :domain CompositeModel
 :range VersionedModel)
(LOOM:defrelation container   ;container is the inverse of vmodel
 :is (:inverse vmodel))
```

UML metaclasses are translated into *Loom* concepts. As an example, the translation of the metaclass *CompositeModel* which is a stereotyped *Model* is given:

```
;Concept COMPOSITEMODEL
(LOOM:defconcept CompositeModel
 :is (:and VersionedModel
  (:all vmodel VersionedModel))
 :in-partition $VersionedModel$)
```

In the same way all the other classes, associations and attributes in the UML metamodel are translated into *Loom*. Logic rules are also used to specify the OCL well-formedness rules of our UML profile.

The modeling elements of the user-defined class, sequence and state diagrams are specified as instances of the appropriate classes, association and attributes of the UML metamodel. This guarantees the consistency of the user-defined model elements with the UML metamodel. As an example, the *Document* class is represented by the instance `Document-1.0` of the concept `Class`. Furthermore, different properties for `Document-1.0` are specified, e.g. this class has two operations presented by `print-Document-1.0` and `preview-Document-1.0` which are instances of the concept `Operation`. The complete translation of the metamodel into *Loom* code can be found in [18].

```
(create 'Document-1.0 'Class)
(tellm (:about Document-1.0
 (name Document)
 (Has-feature print-Document-1.0)
 (Has-feature preview-Document-1.0)
 (Is-parent-of Document-ASCIIDoc-1.0)
 (Is-parent-of Document-PSDoc-1.0)
 (Is-parent-of Document-RTFDoc-1.0)
 (IsAbstract true)
 (In-namespace Class-Diagram-1.0)))
```

6 Experiments

To carry out our experiments, the diagrams of Figure 4, Figure 5 and Figure 6 were manually translated into *Loom*. To detect and resolve inconsistencies between models *Loom*'s query processor is used. Due to space limitations only important fragments of the developed *Loom* predicates are shown. All predicates for all the inconsistencies as detailed in section 3 can be found in [18].

Classless instance. To detect that there are objects in an evolved sequence diagram which are instances of classes that do not belong to any class diagram, the following rule is specified:

```
(retrieve (?class ?obj ?seq-diagram)
 (:and (Class ?class)
  (the-prev-ver ?class NIL)
  (Instance-of-class ?obj ?class)  ;is ?obj an instance of ?class
  (In-namespace ?obj ?seq-diagram))) ;is ?obj present in ?seq-diagram
```

The `the-prev-ver` role links different versions of the same model elements. The statement `(the-prev-ver ?class NIL)` checks for all classes which do not have a previous version. With this predicate we automatically detect that the evolved sequence diagram used an object (`aPrintVisitor`) that is an instance of a new class (*PrintVisitor*) that was not present in the original class diagram.

Dangling feature and association reference. With a similar predicate, we detect the new operations `visitASCII(ASCIIDoc)`, `visitPS(PSDoc)`, `visitRTF(RTFDoc)`, `accept(Visitor)` and `getContents()` that are introduced in the evolved sequence diagrams and are not present in the original class diagram. Another *Loom* predicate is used to detect a new association between

PrintVisitor and *Printer*, because an instance of *Printvisitor* sends the `print` message to *Printer*.

For most of the detected inconsistencies we also provide rules to automatically resolve them. For example, if the evolved sequence diagram contains some new operations (e.g., `accept` and `visitASCII`), we can add them to the evolved class diagram using the following predicate. This predicate retrieves the owner of the operation by querying the object that receives the stimulus associated to the corresponding action. In this case, the *initiator* meta association and *dispatchaction* metaclass are used.

```
(do-retrieve (?stim ?obj ?class ?action ?op)
 (:and
  (Stimulus ?stim)
  (Received-by ?stim ?obj)       ;object ?obj receives stimulus ?stim
  (Instance-of-class ?obj ?class) ;object ?obj is an instance of ?class
  (Initiates ?stim ?action)  ;stimulus ?stim initiates the action ?action
  (DispatchAct-op ?action ?op) ;operation ?op has dispatchaction ?action
  (Is-owned-by ?op NIL))   ;operation ?op does not belong to any class
 (tellm (Is-owned-by ?op ?class)))
```

The statement `(tellm (Is-owned-by ?op ?class))` resolves the inconsistency by making the operation `?op` a member of the class `?class`.

Finally, it is possible to identify candidates for the method pull-up refactoring. This is done using the information that is available in the sequence diagrams. These predicates can be found in [18].

Incompatible behaviour. To check if there is an incompatible behaviour conflict, the order of the received operations by an object is compared with the order of operations associated to the events of its state machine. For this purpose the names of the operations received by an object and the names of the operations associated to the events are retrieved. Again, the *initiator* meta association and *dispatchaction* metaclass are used, as shown in the following fragment of the logic predicate used to detect this conflict:

```
...(:and
   (Receiver-of ?obj ?stim) ;object ?obj receives stimulus ?stim
   (Initiates ?stim ?action) ; ?stim initiates ?action
   (DispatchAct-op ?action ?op) ;?op has dispatchaction ?action
   (name ?op ?name)...  ;?name is the name of ?op
```

In order to retrieve the operations associated to events in the state machine, the information contained by the *Transition* metaclass is used. The source and target state of the transition are retrieved and the event which triggers the transition and finally the name of the corresponding operation is retrieved for comparison.

```
... (:and
    (Is-source-of ?from-state ?transition)
    (Triggered-by ?transition ?event) ;?transition is triggered by ?event
    (Is-occurrence-of ?event ?op) ;operation ?op is related to ?event
    (name ?op ?name) ;?name is the name of ?op
    (Is-target-of ?state ?transition))...
```

Discussion. We applied all the above mentioned rules to the example of section 4. This enable us to detect and resolve all inconsistencies as specified in that section. It is also possible to generate a new version of the class diagram, that incorporated the new information that the user added to the sequence diagram. The inconsistency between the behaviour specified in the sequence diagram on the right of Figure 5 and the state diagram in Figure 6 is also detected by our rules.

For our current experiments, we manually translate the UML models into DL format. However, we are currently working on an automatic translation of UML models (exported from Poseidon in XMI 1.2 format) using XSLT. To this extent, we intend to use the SAXON XSLT processor tool (`saxon.sourceforge.net`).

7 Related work

Finkelstein *et al.* [10] 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 inconsistencies. Grundy *et al.* [15] 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.

A wide range of different approaches for checking consistency has been proposed in the literature. Engels *et al.* [8] motivate a general methodology to deal with consistency problems based on the problem of protocol statechart inheritance. In that example, statecharts as well as the corresponding class diagram are important. Communicating Sequential Processes (CSP) are used as a mathematical model for describing the consistency requirements. This idea is further enhanced in [7, 9] with dynamic meta modeling rules as a notation for the consistency conditions because of their 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 [6] investigate the consistency between UML class and sequence diagrams. UML class diagrams are represented by attributed type graphs with graphical constraints, and UML sequence diagrams by attributed graph grammars. As consistency checks between class and sequence diagrams only existence, visibility and multiplicity checking are considered. In [20] 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. Fradet *et al.* [12] use systems of linear inequalities to check consistency for multiple view software architectures.

Finally, note that consistency of models should not be confused with consistency of a modeling language. UML has been formalized within rewriting logic

and implemented in the Maude system by Ambrosio Toval and his students [1, 19]. Their objectives are to formalize UML and transformations between different UML models. They focus on using reflection to represent and support the evolution of the metamodel.

8 Conclusion

In this paper we propose and validate an approach to detect and resolve inconsistencies between different versions of a UML model, specified as a collection of class diagrams, sequence diagrams and state diagrams. For research purposes, we restrict ourselves to a significant subset of the UML metamodel.

The formalism used, is description logic, a decidable fragment of first-order predicate logic. More specifically, we use the *Loom* knowledge representation tool to formally specify UML models as a collection of concepts and roles.

Logic rules are used to detect and to suggest ways to resolve inconsistencies. Based on a simple but illustrative example, we illustrate the feasibility of the approach. Until now, we only use small examples. The question remains if our approach remains feasible for larger models.

Obviously, a lot of future work remains to be done. We will investigate how the formal properties of DL can help us to prove interesting properties about consistency between UML models. We need to further automate the consistency maintenance process, by directly invoking the description logic engine from within a UML CASE tool (such as Poseidon), and providing feedback about the detected inconsistencies to this CASE tool. We need to incorporate other kinds of UML diagrams (such as collaboration diagrams and activity diagrams). We need to extend our ideas to deal with co-evolution and consistency maintenance between different levels of abstraction, more specifically, source code and UML models. This idea, which is also explored in [14] will allow us to provide better formal support for the round-trip engineering and model-driven architecture process.

References

1. J. Alemán, A. Toval, and J. Hoyos. Rigorously transforming UML class diagrams. In *Proc. 5th Workshop Models, Environments and Tools for Requirements Engineering (MENHIR)*, 2000. Universidad de Granada, Spain.
2. F. Baader, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.
3. J. Bézivin and N. Ploquin. Tooling the MDA framework: a new software maintenance and evolution scheme proposal. *Journal of Object-Oriented Programming (JOOP)*, 2001.
4. J.-P. Bodeveix, T. Millan, C. Percebois, C. L. Camus, P. Bazes, and L. Ferraud. Extending OCL for verifying UML model consistency. In L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar, editors, *Consistency Problems in UML-based Software Development, Workshop UML 2002, Technical Report*, 2002.

5. 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.
6. 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.
7. G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Testing the consistency of dynamic UML diagrams. In *Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002)*, June 2002. Pasadena, CA, USA.
8. G. Engels, R. Heckel, and J. M. Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. 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.
9. G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *Proc. Int'l Conf. UML 2002 - The Unified Modeling Language. Model Engineering, Concepts, and Tools*, number 2460 in Lecture Notes in Computer Science, pages 212–227. Springer-Verlag, October 2002. Dresden, Germany.
10. A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, LNCS, pages 84–99. SpringerVerlag, 1993.
11. M. Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.
12. P. Fradet, D. Le Métayer, and M. 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.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems.* Addison-Wesley, 1994.
14. P. V. Gorp, H. Stenten, T. Mens, and S. Demeyer. A UML extension for automating source-consistent design improvements based on refactoring contracts. In *Proc. 6th International Conference on the Unified Modeling Language*, 2003.
15. J. C. Grundy, J. G. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.
16. R. MacGregor. Inside the loom description classifier. *SIGART Bull.*, 2(3):88–92, 1991.
17. Object Management Group. Unified Modeling Language specification version 1.5. formal/2003-03-01, March 2003.
18. J. 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.
19. A. Toval and J. Alemán. Formally modeling UML and its evolution: a holistic approach. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems IV*, pages 183–206. Kluwer Academic Publishers, 2000.
20. A. Tsiolakis. Semantic analysis and consistency checking of UML sequence diagrams. Master's thesis, Technische Universität Berlin, April 2001. Technical Report No. 2001-06.