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. 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 [14]. 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. Vertical consistency indicates consistency between a model and its successor refinements. However, vertical consistency will not be treated in this paper. [12]

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 [14] provides poor support for consistency preservation and software evolution, e.g. versions are not supported. It is shown 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 com-
plete. While the satisfiability problem is undecidable in first-order logic, most DLs have decidable inference mechanisms. These inference mechanisms allow one 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 [13] because of its extensive query language and associated production rule system. This allows one to specify UML models, their evolution, consistency rules and also design improvements in a straightforward way.

In the next section the developed UML profile for model consistency is explained. Before introducing the running example used in this paper in section 3, a possible classification of inconsistencies is proposed in section 2. A motivation for the use of description logic is given in section 4. Section 5 outlines the UML profile we developed for model consistency. Section 6 discusses some experiments and section 7 gives a summary of related work. We conclude in section 8.

2. Classification of Inconsistency Conflicts

For the sake of presentation, we deliberately confine ourselves to three kinds of UML diagrams: class diagrams, sequence diagrams and state diagrams. In this section, we propose a two-dimensional classification of inconsistency conflicts that can be observed between these (evolving) UML diagrams.

The first dimension indicates whether structural or behavioural aspects of the models are affected. Structural inconsistencies arise when the structure of the system is inconsistent with respect to existing behaviour. They typically appear in class diagrams which describe the static structure of the system. Behavioural inconsistencies arise when the behaviour of the system is inconsistent with respect to existing behaviour or definitions. They typically appear in sequence and state diagrams which describe the dynamic behaviour of the system.

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. Remark that only sequence diagrams representing a CollaborationInstanceSet are considered. Conflicts can occur at the Model level, between the Model and Instance level, or at the Instance level. The classes of observed conflicts are listed in Table 1. Because of space limitations, only the dangling (inherited) association reference conflict which belongs to the instance definition missing class of conflicts is detailed in the next section. The explanation of the other instance definition missing conflicts can be found in [18] together with the explanation of the incompatible behaviour conflicts. All conflicts mentioned in Table 1 are detailed in [15].

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** occurs when a link in a sequence diagram is related to a non-existing association in a class diagram. This includes inherited associations that are lost when inheritance links between classes are removed. This conflict can also be caused by the removal of existing associations or by the omission of the necessary associations when creating the class diagram.

Concrete examples of dangling (inherited) association reference are given in the next section.

3. Running Example

In this section, a running example is introduced that will be used throughout the paper. The automatic teller machine (ATM) example is rather small but sufficiently complex to illustrate our ideas. Figure 1 shows part of a class diagram of our ATM example. This class diagram contains an ATM class together with its subclass PrintingATM and all necessary classes, associations and operations to be able to represent communication with an ATM. The sequence diagram

---

1 adopted from http://www.math-cs.gordon.edu/local/courses/cs211
shows the execution of a withdraw transaction performed on an ATM enabling printing receipts. Starting from these diagrams we will explain the experiments we performed.

In the UML model consisting of the diagrams in Figure 1, two examples of the inherited association conflict can be demonstrated. The first example appears when the inheritance relationship marked as A in Figure 1 is removed. As a result the PrintingATM class does not inherit the association between the ATM and CashDispenser classes. As such the links C and D of the sequence diagram in Figure 1 refer to an illegal association. The second example appears when the association between the ATM and CashDispenser classes, marked as B in the class diagram is deleted. The links C and D refer to a non-existing association. All these experiments are detailed in section 6.

### 4. Description Logic

Description Logics (DLs) are a family of knowledge representation formalisms. Those formalisms allow us to represent the knowledge of the world by defining the concepts of the application domain and then using 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. PrintingATM is an ATM. In this case, PrintingATM is a subconcept of ATM and ATM is the superconcept of PrintingATM. Classification of individuals provides useful information on the properties of individuals e.g., if an individual is classified as an instance of PrintingATM, we infer that it is also an ATM. 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 Withdraw transactions debit an Account, has as result that an individual known to be a Withdraw transaction, is also known to debit an Account. 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 the software application. For example, Cali et al. [4] translated UML class diagrams to the description logic DLR.

Several implemented DL systems exist (e.g., Loom, Classic, and so on). We have selected the Loom system for carrying out our experiments because it offers reasoning facilities on concepts and individuals for the DL ALCQRIFO. This logic extends the basic description logic ALC with qualified number restrictions on roles, inverse roles, role hierarchy and nominals. 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. This makes it possible to specify additional necessary conditions for individuals which are explicitly mentioned and are derived to be instances of a certain defined concept. For example, consider the following query:

```lg
:give values to "the-prev-ver" role of class concept instances
(do-retrieve (?c1 ?c2 ?et ?m1 ?m2)
(:and
 (Class ?c1)
 (Class ?c2)
 (EvolutionTrace ?et)
 (Supplier ?et ?m1)
 (In-namespace ?c1 ?m1)
 (Client ?et ?m2)
 (In-namespace ?c2 ?m2)
 (:same-as (name ?c1) (name ?c2))
 (tellm (the-prev-ver ?c2 ?c1)))
```

This query searches all pairs of classes of which the second class (c2) belongs to the next version (m2) of the model (m1) to which the first class (c1) belongs. If those classes have the same name, we can conclude that the second class is the next version of the first class. To make this relation

<table>
<thead>
<tr>
<th>Model-Model</th>
<th>Behavioural</th>
<th>Structural</th>
<th>dangling (type) reference inherited association conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model-Instance</td>
<td>incompatible definition</td>
<td>instance definition missing</td>
<td></td>
</tr>
<tr>
<td>Instance-Instance</td>
<td>invocable behaviour conflict observable behaviour conflict incompatible behaviour conflict disconnected model</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Two-dimensional inconsistency conflict table
explicit, a relation the-prev-ver is established between the first and the second class.

5. UML

To be able to support consistency and evolution of UML models, we need to develop a UML profile for model consistency. This UML profile contains a subset of the UML metamodel. This subset is then further extended to support evolution.

5.1. UML Profile

Because only class, sequence and state diagrams are considered, our UML profile consists of subsets of the Core, Model Management, Common Behaviour, Collaborations and State Machines packages of the UML metamodel.

To express model evolution and model consistency, we need to extend our subset of the UML metamodel. In our UML profile, horizontal and evolution consistency can be expressed by defining stereotypes for the Trace metaclass: HorizontalTrace and EvolutionTrace.²

We also need a notion of VersionedModel, which is a stereotype for the Model metaclass. It adds a tag-value pair (version, Integer) to denote the model version.

To specify the kind of models that can be related by horizontal or evolution consistency traces, the Model metaclass in the package Model Management is stereotyped to distinguish between primitive models and composite models. PrimitiveModel is a stereotyped Model that can be specialised (stereotyped) further into ClassModel, SequenceModel and StateModel (representing a class diagram, sequence diagram and state diagram, respectively). 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 to refer to the CompositeModel it belongs to.

5.2. Translation of UML Profile into DL

Our UML profile is translated 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
...
UML metaclasses are translated into Loom concepts. As an example, the translation of the metaclass Composite-Model 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. The OCL well-formedness rules of our UML profile are translated into logic 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).

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 ATM class is represented by the instance ATM-1.0 of the concept Class. Furthermore, different properties for ATM-1.0 are specified, e.g. this class has the operations getPin() and getAmountEntry() presented by getPin-1.0 and getAmountEntry-1.0 which are instances of the concept Operation. The complete translation of the metamodel into Loom code can be found in [15].

6. Experiments

To carry out our experiments, the diagrams of Figure 1 are 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 described in section 2 can be found in [15].

6.1. Inherited association conflict - illegal associations

To detect links that reference illegal associations due to the fact that the inheritance link between two classes is removed, the following predicate is used:

(defun illegal-link (?link ?assoc)
 (let* (((?list
   (retrieve (?class1 ?class2 ?class3 ?class4
     ?assocEnd1 ?assocEnd2
     ?stim ?obj1 ?obj2)
   (:and
     (assoc-assocEnd ?assoc ?assocEnd1)
     (assoc-assocEnd ?assoc ?assocEnd2)
     (has-participant ?assocEnd1 ?class1)
     (has-participant ?assocEnd2 ?class2)
     (link-stimulus ?link ?stim)
     (received-by ?stim ?obj1)
     (sent-by ?stim ?obj2)
     (instance-of-class ?obj1 ?class3)
     (instance-of-class ?obj2 ?class4))))
 (if (equalp NIL (related ?list)
 (format t "association exists, but not available through inheritance" ?assoc)))
))

First of all, the classes related through the association ?assoc and the classes linked by this association in the sequence diagram are collected in the variable list. The predicate related checks if those classes are related by inheritance. If this is not the case, we conclude that the association exists but is not available through inheritance. If this predicate is applied to the example where the inheritance link between the ATM and PrintingATM classes is removed, the obtained results indicate that, even though the association between the ATM and Class Dispenser(CashDispenser-ATM-1.0) exists in the class diagram, the PrintingATM class no longer inherits this association and cannot refer to it.

6.2. Inherited association conflict - non-existing associations

To detect links that reference non-existing associations, the following query is used:

(do-retrieve (?link ?assoc)
 (:and
   (Link ?link) ;is ?link a link
   (Link-association ?link ?assoc)
   (In-namespace ?assoc NIL)))
This predicate is applied to the example where the association between the \textit{ATM} and \textit{CashDispenser} classes (CashDispenser-ATM-1.0) is removed and it automatically detects that the association CashDispenser-ATM-1.0 does not exist anymore.

\[ I | \text{CASHDISPENSER-ATM-1.0 association does not exist in any class diagram} \]

For most of the detected inconsistencies we also provide rules to automatically resolve them. For example, it is possible to add the non-existing association between the classes \textit{ATM} and \textit{CashDispenser} with or without user interaction.

We applied all the above mentioned rules to the example of section 3. This enables us to detect and resolve all inconsistencies as specified in that section. Other, more extensive experiments are performed in [15].

7. Related work

Finkelstein \textit{et al.} [9] 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 \textit{et al.} [11] 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, of modifying logic rules and facts in the knowledge base.

A wide range of different approaches for checking consistency has been proposed in the literature. Engels \textit{et al.} [7] 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 [6, 8] 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 [5] 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 [17] 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 \textit{et al.} [10] 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, 16]. 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 \textit{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 prove interesting properties about consistency between UML models. We need to further automate the consistency maintenance process, by using a description logic tool as a basic engine for 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 also need to translate the corresponding OCL well-formedness rules. 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 [19] will allow us to provide better formal support for the round-trip engineering and model-driven architecture process.
References


