Maintaining Consistency between UML Models Using Description Logic

Jocelyn Simmonds* — Ragnhild Van Der Straeten† — Viviane Jonckers† — Tom Mens‡

* Dep. de Ciencias de la Computación, Universidad de Chile
Av. Blanco Encalada 2120, 837-0459 Santiago, Chile
jsimmond@dcc.uchile.cl

† Systems and Software Engineering Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
rvdstrae@vub.ac.be viviane@info.vub.ac.be

‡ Service de Génie Logiciel, Université de Mons-Hainaut
Av. du Champ de Mars 8, 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 their consistency, since these diagrams are subject to continuous changes due to successive refinements or evolutions. Contemporary UML tools provide unsatisfactory support for maintaining the consistency between different versions of UML diagrams. To solve this problem, an extension of the UML metamodel is developed, and a classification of inconsistency problems is proposed. Detection and resolution of consistency conflicts is expressed by means of rules in description logic. By carrying out a number of concrete experiments, we show the feasibility of the description logic formalism for the purpose of maintaining consistency between evolving UML models.

RÉSUMÉ. La conception d’un logiciel est souvent modélisée comme un ensemble de diagrammes UML. Il est essentiel de préserver leur cohérence, parce que ces diagrammes sont sujets à de fréquentes modifications, dues aux évolutions successives. Les outils contemporains pour UML fournissent un support insatisfaisant pour maintenir la cohérence entre différentes versions des diagrammes UML. Pour résoudre ce problème, une extension du méta modèle UML est développée, et une classification des conflits d’incohérence est proposée. La détection et la résolution de ces conflits est exprimée en utilisant des règles en logique de descriptions. En effectuant un certain nombre d’expériences, nous montrons la faisabilité du formalisme de logique de descriptions pour préserver la cohérence entre des modèles UML qui évoluent.

KEYWORDS: consistency maintenance, UML, description logic, model evolution.

MOTS-CLÉS : préservation de cohérence, UML, logique de descriptions, évolution des modèles.

RSTI - L’objet – 10/2004. LMO’04, pages 231 à 244
1. Introduction

A software design is typically specified as a collection of UML diagrams [Obj 03]. Different aspects of the software system are covered by different UML diagrams. Refining or evolving these diagrams may make the design model inconsistent. Therefore, it is necessary to check and ensure the consistency between UML diagrams when any of them get modified. Hence, it is crucial to provide tools that allow us to detect and resolve inconsistencies between related UML diagrams.

As illustrated in Figure 1, we can distinguish three different types of consistency. A first type, indicating consistency between different models within the same version, is called horizontal consistency [KUZ 02]. A second type, called evolution consistency indicates consistency between different versions of the same model [ENG 02c]. Finally, vertical (or refinement) consistency indicates consistency between a model and its successive refinements [KUZ 02]. As a special case, refinement can also be used to maintain consistency between a design model and the corresponding implementation.

Figure 1. Three types of consistency between UML models

In this paper however, we will restrict ourselves to design model consistency and will not deal with vertical consistency. To be able to write tools that detect and resolve design inconsistencies, we first of all need to define the notion of consistency between (evolving) UML models in a formal and precise way. To this extent, we first propose a classification of design inconsistencies in Section 2. Because the current UML metamodel [Obj 03] provides poor support for consistency preservation and software evolution (e.g., versions are not supported), we also make some small but important changes to the UML metamodel in Section 3.

To specify, detect and resolve design inconsistencies in a precise yet executable way, we propose to use the formalism of description logic (DL) [BAA 03] in Section 4. DL is a fragment of first-order predicate logic that offers a classification task that is
decidable and complete. While the satisfiability problem is undecidable in first-order logic, 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.

Section 5 discusses some concrete experiments we have carried out to deal with inconsistencies of evolving UML models. To this extent we used Loom [MAC 91], a description logic tool with an extensive query language and associated production rule system. This tool allows us to specify UML models, their evolution, consistency rules and also conflict resolution strategies in a straightforward and uniform way. Section 6 provides a summary of related work, and Section 7 gives some concluding remarks and directions for further research.

2. Classification of design inconsistencies

According to Spanoudakis [SPA 01], an inconsistency is a state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are not jointly satisfiable. This definition is directly applicable to UML models that are specified as a set of overlapping UML diagrams. While different types of diagrams reason about different aspects of the system (e.g., class diagrams deal with the static structure, sequence diagrams with the dynamic behaviour), they rely on the same model elements that can make conflicting assertions about these elements. For example, an operation may be removed in a class diagram while an instance of this class (i.e., an object) in a sequence diagram still relies on this operation to handle a message it receives from another object.

In this section, we propose a classification of design inconsistencies that can arise between class diagrams, sequence diagrams and state diagrams. Due to the nature of these diagrams, the classification will be two-dimensional.

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, and typically appear in class diagrams that describe the static structure of the system. Behavioural inconsistencies arise when the specification of the system behaviour is inconsistent, and typically appear in sequence and state diagrams that describe the dynamic behaviour of the system.

The second dimension concerns the level of the affected model. A class diagram belongs to the Class level because the model elements it represents (such as classes and associations) serve as declarations for instances (such as objects, links, transitions and events) in sequence and state diagrams belonging to the Instance level. Consequently, inconsistencies can occur at the Class level, between the Class and Instance level, or at the Instance level. These categories of observed inconsistencies are listed in Table 1, and are explained in detail in [SIM 03]. Because of space limitations, we will only explain two categories of inconsistencies in more detail.
### Table 1. Two-dimensional inconsistency table

<table>
<thead>
<tr>
<th>Behavioural</th>
<th>Structural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Structural</td>
</tr>
<tr>
<td>invocable collaboration</td>
<td>dangling (type) reference</td>
</tr>
<tr>
<td>behaviour inconsistency</td>
<td>inherited association inconsistency</td>
</tr>
<tr>
<td>observable collaboration</td>
<td>role declaration missing</td>
</tr>
<tr>
<td>behaviour inconsistency</td>
<td></td>
</tr>
<tr>
<td>Class-Instance</td>
<td></td>
</tr>
<tr>
<td>incompatible declaration</td>
<td>instance declaration missing</td>
</tr>
<tr>
<td>Instance</td>
<td></td>
</tr>
<tr>
<td>invocable behaviour inconsistency</td>
<td></td>
</tr>
<tr>
<td>observable behaviour inconsistency</td>
<td></td>
</tr>
<tr>
<td>incompatible behaviour inconsistency</td>
<td></td>
</tr>
<tr>
<td>disconnected model</td>
<td></td>
</tr>
</tbody>
</table>

**Incompatible behaviour inconsistency** occurs when the sequence of messages received by an object in a sequence diagram is inconsistent with the specification of the behaviour established by the protocol state machine of the object represented in the state diagram. A *protocol state machine* has only transitions with events referencing an operation and an empty action sequence. No action specifications exist other than for the methods. A sequence of messages is inconsistent with respect to a protocol state machine if it is not possible to find a path in the state machine that follows the order established by the sequence diagram.

**Disconnected model** inconsistency occurs when a state diagram has one or various states that are not reachable from the initial state, i.e., when there are no paths connecting the initial state to those particular states. This usually occurs when the connecting states or transitions have been removed from the diagram. It can also occur when the designer creates a disconnected state diagram, for example, by forgetting to include a transition.

We will now give a concrete and motivating example of these two inconsistencies. The example is based on the design of an automatic teller machine (ATM), originally developed by Russell Bjork for a computer science course at Gordon University. The sequence diagram in Figure 2 and the state diagram in Figure 3 show the execution of a `dispenseCash()` action performed on an ATM. Starting from these diagrams we have made some modifications leading to an inconsistency.

The state machine in Figure 3 represents a protocol state machine that indicates the permitted behaviour sequences. The ordering of the stimuli received by the instance of the *ATM* class is perceived as inconsistent with the established behaviour, since it receives a `ejectCard()` message before an `dispenseCash()` message. The protocol state machine of this class has no path of transitions modeling this specification of behaviour resulting in an *incompatible behaviour* conflict.

Suppose that the designer deleted the *AmountEntry* state and the transitions to and from this state from the state diagram shown in Figure 3 (represented by the dashed el-
To be able to support consistency for evolving UML models, we require a UML profile for model consistency, which is realised by defining new stereotypes and their associated OCL constraints. The three types of consistency explained in Figure 1 can be expressed as follows in the UML metamodel. For vertical consistency we can use the UML Refinement relationship, which is a stereotype «refine» of the Abstraction metaclass. It can be used to specify the derivation relationship between a model and a refinement of this model that includes more specific details. The UML Trace relationship, which is another stereotype «trace» of the Abstraction metaclass, can be used to express horizontal and evolution consistencies. It suffices to stereotype Trace further into HorizontalTrace (stereotype «horizontal») and EvolutionTrace (stereotype «evolution»), respectively. To support versioning of UML models, we stereotype the metaclass ModelElement into a VersionedElement (stereotype «versioned»). To this stereotype, a tag-value pair with tag name version is associated. Its value represents the version number. We deliberately do not specify the type of this value, since diffe-
rent version control tools use different ways to represent versions. The only important requirement here is that version numbers must be comparable.

For these newly defined stereotypes, we need three additional OCL constraints. Due to space limitations, we only show the second constraint in OCL and in Loom syntax (see section 4).

1) A «versioned» Model can only contain «versioned» ModelElements as ownedElements. Moreover, the version of each ownedElement must equal the version of the containing Model.

2) A «horizontal» Trace can only be specified between «versioned» clients and «versioned» suppliers that all have the same version.

```ocl
class Model
|
| +ownedElement : ModelElement[0..*]
| +namespace : Integer[0..1]
|

class ModelElement
|
| +supplier : Model[0..*]
| +supplierDependency : ClientDependency[0..*]
| +client : Model[1..*]
| +clientDependency : ClientDependency[1..*]
|

class Abstraction
|
| +stereotype : «stereotype»
|

context Abstraction inv:
let getVersion(me:ModelElement): TaggedValue =
  me.stereotype->any(name="versioned").definedTag
  ->any(tagType="version").typedValue in
  (self.stereotype->name->includes("horizontal")) implies
  (self.client->union(self.supplier))->forall(e |
    e.stereotype->name->includes("versioned") and
    (getVersion(e) = getVersion(self)) )

3) All suppliers of an «evolution» Trace must be «versioned» and must have the same version. All clients of an «evolution» Trace must be «versioned» and must have the same version. The version of the suppliers must precede the version of the clients.

4. Description logic

Description Logics (DLs) are a family of formalisms to represent the knowledge of the world by defining the concepts of the application domain and then using these
consistency 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 deliberately restricted to keep it decidable. 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. 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 design structure of the software application. For example, Calì et al. [CAL 01] translated UML class diagrams to a description logic.

Several implemented DL systems exist, such as the second generation systems Loom and Classic [BOR 89] but also third generation systems such as FaCT [HOR 98] and RACER [HAA 01]. A serious shortcoming of these third generation systems is the inadequacy of their query languages as recognized by [HOR 02]. We have selected the Loom system for carrying out our experiments because it offers reasoning facilities on concepts and individuals. 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. We translated our UML profile 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 Loom translation of the meta association between ModelElement and Model with roles namespace and ownedElement. The translation uses (inverse) roles between concepts. The role namespace between a ModelElement and a Model translates into the role namespace with as domain the concept ModelElement and as range Model. The role ownedElement translates into the role ownedElement which is the inverse role of namespace. UML metaclasses and stereotypes are translated into Loom concepts. As an example, the translation of the stereotype VersionedElement of metaclass ModelElement is given.

(Loom:rel namespace
 :domain ModelElement
 :range Model :characteristics :single-valued)
In the same way all the other classes, stereotypes, inheritance relationships, associations and attributes in the UML metamodel are translated into Loom. The complete translation of the metamodel into Loom code can be found in [SIM 03]. The OCL well-formedness rules of our UML profile are translated into logic rules.

The modeling elements of 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 anATM and aSession objects of Figure 2 are represented by the instances anATM and aSession of the concept Object respectively. Furthermore, it is specified that anATM is an instance of the class ATM. In the same way, a stimulus stim3 can be defined reifying the communication between anATM and aSession and a callaction call13 causing this stimulus to be dispatched when it was executed.

Using Loom's concept constructors and its query mechanism the HorizontalTrace stereotype can be defined and also the corresponding constraint (as specified in section 3):

```
(define-concept horizontaltrace :is
 (:and (:all client (:and versionedelement))
     (:all supplier (:and versionedelement))))
```

1. Multiple inheritance of metaclasses and stereotypes is naturally translated into the powerfull subsumption mechanism of description logics
5. Experiments

For our experiments we set up the tool chain illustrated in Figure 5. While we manually specified our UML profile in $Loom$, specific UML models expressed in the UML CASE tool $Poseidon$ are exported in XMI 1.2 format, and automatically translated into description logics format using $Saxon$, an XML translator. The logic code is then asserted into a knowledge base maintained by the $Loom$ logic reasoning engine. To detect and resolve design inconsistencies, $Loom$’s query processor is used.

We will now illustrate how we detected the inconsistencies explained in Figures 2 and 3. The queries for all other inconsistencies of Table 1, as well as more extensive experiments, can be found in [SIM 03].

To detect the *incompatible behaviour inconsistency* the ordered collection of operations dispatched by stimuli received by a certain object is collected. This is done using the following predicate:

\[
 (: \text{and} \ (\text{client} \ ?\text{horizontaltrace} \ ?\text{client})
   \ (\text{for-all} \ (?\text{supplier}) \ (:\text{implies} \ (?\text{supplier} \ ?\text{horizontaltrace} \ ?\text{supplier})
   \ (:\text{same-as} \ (?\text{version} \ ?\text{client}) \ (?\text{version} \ ?\text{supplier})))))
\]

Using this collection of operations as a guide to choose the corresponding transitions, the object’s state diagram is traversed. If there are no suitable transitions from a certain state an inconsistency is found. Remark that the traversal does not necessarily start at the initial state of the state diagram, as the behaviour depicted in a sequence diagram is usually incomplete in contrast to protocol state machines which should be as complete as possible.
Applying this function to the sequence diagram in Figure 2 and the state diagram in Figure 3 yields the following result:

Current operation: |I|CHECKIFCASHAVAILABLE
Current operation: |I|EJECTCARD
Incompatible behaviour found at state: |I|VERIFYATMBALANCE

The disconnected model inconsistency can be detected in Loom by checking if there is at least one path from the initial state to a particular state using recursive calls analogous to the previous example. DLs equipped with transitive roles can express this recursiveness. The third generation system RACER supports such an expressive DL.

The disconnected model inconsistency can be checked by RACER exploiting transitive roles and role hierarchies. For this purpose, we define a new role successor-state between two states which is transitive and as such keeps track of all the paths between different states. Next to this transitive role, we define another role direct-successor-state which is a subrole of the previous one.

All the states that are connected through a transition are connected by the direct-successor-state role and because this is a subrole of successor-state, all connected states are related through successor-state. To check if there is a path relating two states, it suffices to use the RACER built-in query individuals-related which checks if two specified individuals are related via the specified role. In this case, the individuals are initial, representing the initial state of the state diagram in Figure 3 and verifyAccountBalance, representing the verifyAccountBalance state in that state diagram.
This query returns `nil` indicating that those individuals are not related through the role `successor-state`, meaning that there is no path from the initial state of this state diagram to the `verifyAccountBalance` state.

For these two categories of inconsistencies it is possible to provide rules that resolve those conflicts but it is not feasible to automatically apply these rules. The user will have to select between the possible alternatives. In the case of incompatible behaviour inconsistency, it is only possible to alert the user about the existence of this inconsistency, the user must determine whether the problem lies in the protocol state machine or in the sequence diagram. For other detected inconsistencies we provide rules to automatically resolve them.

6. Related work

Finkelstein et al. [FIN 93] 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. [GRU 98] 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 rules and facts in the knowledge base.

A wide range of different approaches for checking consistency has been proposed in the literature. A survey of inconsistency management in software engineering is given in [SPA 01]. The logic-based approach for detecting inconsistencies is discussed in this survey. As limitations the semi-decidability of first-order logic and the computationally inefficiency of theorem proving are brought forward. In our approach DLs are used which are decidable and use optimized tableau- and automata-based algorithms.

Engels et al. [ENG 01] 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 [ENG 02a, ENG 02b] 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 model consistency is explored.

In Briand et al. [BRI 03] changes between two different versions of a UML model are identified and model elements impacted by those changes are determined using formally defined impact analysis rules. Consistency can be checked within the original
and evolved UML model. However, their work emphasizes change impact whereas our approach is focused on consistency maintenance.

Ehrig and Tsiolakis [EHR 00] 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 [TSI 01] 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. [FRA 99] use systems of linear inequalities to check consistency for multiple view software architectures.

In Cali et al. 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, [CAL 03] proves that reasoning on UML class diagrams is \( \text{EXPTIME} \) hard.

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 [ALE 00, TOV 00]. 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.

7. Conclusion

In this paper we proposed an approach based on description logic, a decidable fragment of first-order predicate logic, to detect and resolve inconsistencies between different versions of a UML design model. For research purposes, we restricted ourselves to class diagrams, sequence diagrams and state diagrams only.

As a feasibility study of our approach, we implemented a tool chain in which the Loom knowledge representation tool was used to formally specify and reason about UML models as a collection of concepts and roles. We used this tool to detect some inconsistencies we encountered in a simple ATM simulation. Logic rules were successfully used to detect and resolve inconsistencies.

A lot of future work remains to be done. From a practical point of view, the usability of our tool needs to be improved. The users should not be aware of the underlying formalism of DL. Until now, we have only validated our approach on small examples. The question remains if our approach remains feasible for larger models. We also need
to investigate how the formal properties of DL can help us to prove interesting properties about consistency between UML models. We need to incorporate other kinds of UML diagrams (such as collaboration diagrams and activity diagrams). We need to translate the metamodel and OCL constraints to logic rules in DL in an automated way. We need to extend our ideas to deal with co-evolution and consistency maintenance between UML design models and source code. This will allow us to provide better formal support for the round-trip engineering and model-driven architecture process. We also want to use our approach to detect opportunities for design improvements, and to specify restructurings to introduce them.

8. References


