

Alexander Egyed, Roberto E. Lopez-Herrejon,  
Bashar Nuseibeh, Goetz Botterweck,  
Marsha Chechik, Zhenjiang Hu (eds.)

3<sup>rd</sup> Workshop on Living with  
Inconsistencies in Software  
Development  
(LWI 2010)

Pre-proceedings

held on 21 September 2010 in conjunction with the  
25<sup>th</sup> International Conference on Automated Software  
Engineering (ASE 2010)

In software engineering, there has long been a recognition that inconsistency is a fact of life. Evolving descriptions of software artifacts are frequently inconsistent, and tolerating this inconsistency is important if flexible collaborative working is to be supported.

The workshop follows on two previous successful iterations at ICSE-97 and ICSE-01. The new iteration is motivated by the recognition of the growing importance that inconsistency management has acquired over the last few years.

Many excellent research papers have recently appeared in multiple and dispersed venues which denotes a rekindled interest in the area. Thus we want to capitalize on this renewed interest and provide a venue for researchers and practitioners to gather and exchange ideas.

We want to thank Tom Mens for giving the keynote entitled *Approaches to Software Model Inconsistency Management*.

## Program Committee

Xavier Blanc, University Pierre and Marie Curie, France  
Betty Cheng, Michigan State University, USA  
Anthony Finkelstein, University College London, UK  
Vincenzo Gervasi, University of Pisa, Italy  
John Grundy, Swinburne University of Technology, Australia  
Jochen Küster, IBM Research Zurich, Switzerland  
Julio Leite, PUC-Rio, Brazil  
Emmanuel Letier, University College London, UK  
Christian Nentwich, Model Two Zero Ltd, UK  
Bill Robinson, Georgia State University, USA  
Alessandra Russo, Imperial College London, UK  
Mehrdad Sabetzadeh, Simula School of Research & Innovation AS, NO  
Andres Silva, Universidad Politecnica de Madrid, Spain  
Andrea Zisman, City University London, UK  
Didar Zowghi, University of Technology, Sydney (UTS), Australia

## Workshop Organisers

Alexander Egyed, Johannes Kepler University Linz, Austria  
Roberto E. Lopez-Herrejon, Johannes Kepler University Linz, Austria  
Bashar Nuseibeh, Lero, Ireland and The Open University, UK  
Goetz Botterweck, Lero, Ireland  
Marsha Chechik, University of Toronto, Canada  
Zhenjiang Hu, National Institute of Informatics, Japan

This research was partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965.

We, the workshop organisers, are grateful to the members of the program committee and the reviewers. We also would like to thank the organisers of ASE 2010, in particular Tom Mens and Tefvik Bultan, the workshops chairs, for their support and the opportunity to hold LWI 2010 in conjunction with this conference.

## Table Of Contents

Approaches to Software Model Inconsistency Management. Keynote.....1 <i>Tom Mens.</i>	1
Inconsistency Detection in Distributed Model Driven Software Engineering Environments.....2 <i>Alix Mougnot, Xavier Blanc and Marie-Pierre Gervais</i>	2
Model Inconsistencies with Automated Planning.....8 <i>Jorge Pinna Puissant, Tom Mens and Ragnhild Van Der Straeten</i>	8
Tolerating Inconsistency in Feature Models.....15 <i>Bo Wang, Zhenjiang Hu, Yingfei Xiong, Haiyan Zhao, Wei Zhang and Hong Mei</i>	15
New Strategies to Resolve Inconsistencies between Models of Decoupled Tools..21 <i>Anne-Therese Körtgen</i>	21
Reasoning about Consistency in Model Merging.....32 <i>Mehrdad Sabetzadeh, Shiva Nejati, Marsha Chechik and Steve Easterbrook</i>	32
Utilizing the Relationships Between Inconsistencies for more Effective.....39 Inconsistency Resolution <i>Alexander Nöhrrer and Alexander Egyed</i>	39

# Approaches to Software Model Inconsistency Management (Keynote)

Tom Mens

Software Engineering Lab, University of Mons, Belgium  
tom.mens@umons.ac.be

**Abstract.** In this talk we focus on the problem of model inconsistency management, an active field of research in model-driven software engineering. We focus on the activity of inconsistency resolution in particular. Automating the resolution of model inconsistencies turns out to be quite a challenge. We provide an overview of a number of research approaches to inconsistency resolution that have been proposed, discuss their limitations, and propose some novel research avenues in this important area of research.

# Inconsistency Detection in Distributed Model Driven Software Engineering Environments

Alix Mougenot<sup>\*</sup>  
LIP6, UPMC - Paris  
Universitas, Paris France  
alix.mougenot@lip6.fr

Xavier Blanc  
LIP6, UPMC - Paris  
Universitas, Paris France  
xavier.blanc@lip6.fr

Marie-Pierre Gervais  
LIP6, UPMC - Paris  
Universitas, Paris France  
marie-  
pierre.gervais@lip6.fr

## ABSTRACT

Model driven development uses more and more complementary models. Indeed, large-scale industrial systems are currently developed by hundreds of developers working on hundreds of models by different distributed teams. In such a context, model inconsistency detection is gaining a lot of attention as the overlap between all these models, which are often maintained by different persons, are a common source of inconsistencies. This paper proposes a method to detect inconsistencies when models are scattered on different editing sites using partial replication. The method provides a way to check the consistency of a single view against the ones that are related to it regarding consistency. It relies on Praxis, an operation based representation of models, to determine what information needs to be collected for consistency checking and the DPraxis protocol to find where it can be.

## Categories and Subject Descriptors

D.2 [Software Engineering]:

## Keywords

Consistency, Model, View.

## 1. INTRODUCTION

Model driven development uses more and more complementary models. Indeed, large-scale industrial systems are currently developed by hundreds of developers working on hundreds of models by different distributed teams [13]. In such a context, model inconsistency detection is gaining a lot of attention as the overlap between all these models, which are often maintained by different persons and from different perspectives, are a common source of inconsistency.

<sup>\*</sup>This work was founded by the French Weapon Agency (DGA) and the *MOVIDA* ANR Project

Each software designer only partially contributes to the global design. Only some of the global design's model elements are interesting to each software designer. In [5], authors define the notion of viewpoint for addressing this concern. A viewpoint is defined by the knowledge, responsibilities and understandings of some part of the system. Each software designer has his own view of the global system to design. For work efficiency reasons each view contains only the model elements of the global design the designer is concerned with. It should be noted that views do not provide a partitioning of the global design as they have elements in common.

Views allow different software designers to concurrently access elements of the global design. Since these concurrent modifications are done from different point-of-views of the global design, inconsistencies arise between and within the views. View Consistency is then a key matter in software engineering. In order to control the consistency of global design, consistency rules are defined and checked against the global design. When an inconsistency is detected, it is reported and logged for further resolving. Consistency checks should be regularly done in order to make sure that the separation of concerns granted by the use of views does not introduce inconsistencies due to the distribution of the information. Detection of inconsistencies between views is a well known research domain (the interested reader will find a precise description in [14, 4]). Many approaches have been proposed in this domain [7, 5, 6, 3, 10, 8, 1] and many tools are available for model consistency validation (Object Constraint Language checkers, Eclipse Model Framework validation tools, Epsilon Validation Language, Praxis . . .), but except [10] they have all been proposed on single machine architectures and don't cover how to perform inconsistency detection in a distributed environment. The actual challenges of modeling concerns huge systems that are modeled by distant teams, in a distributed fashion, assuming possible disconnections and delays between the designing sites, is not answered by the state of the art.

A designer can produce, modify or delete model elements that belong to his view. During the execution of the development process, designers will collaborate. Hence, they will need to access model elements on other sites. Once a designer has accessed model elements from any other view, these model elements can be incorporated to his view as he now relies on it. Also, a designer's view always remains partial for sake of the separation of concerns. Respecting the

separation of concern principle inevitably leads to inconsistency between the different views. Inconsistencies need to be detected as soon as possible to avoid project failures. In this article, we address the problem of inconsistency detection when the views are distributed. More precisely, we want to tackle with this problem when each designer's site only holds the designer's view. The focus of this work is then to provide a method for detecting single-view inconsistencies, that are all the inconsistencies related to a particular view. In other words, we want to provide a method that only detect inconsistencies that the view's designer can understand and resolve.

We identified two problems that need to be tackled regarding single-view inconsistency detection. The first one is to be able to identify the needed information from other views. This point is handled in section 2. The second lock concerns the problem of finding in which view the interesting information is, which is dealt with in section 3.

## 2. WHAT TO GET

If no information is available about what kind of data is relevant to each inconsistency rule, the consistency check of a single view will need to download the entirety of the related views; and possibly the entirety of the views related to the related views, which can quickly end-up in downloading the entire design. The gathering of such a wasteful amount of data can be avoided using filters that can be deduced from the consistency rules' code.

In this section we briefly revisit the three elements from our previous work [1, 9, 2] on which we rely to determine which parts of a model are useful for checking a particular rule. These three elements are the *Praxis model construction* that is the model representation used in our approach, the *Praxis inconsistency rule formalism* that is used for the representation of inconsistency rules and the *consistency impact matrix* which is the data-structure used by the single-view inconsistency checker to determine the needed information for each inconsistency rule.

### 2.1 Praxis Model Construction

We propose a formalism to represent models as sequences of unitary editing operations [1, 9]. This formalism, named Praxis, is used to represent any model as well as any set of model changes. It represents models using six unitary operations that are revisited here:

- *create*( $me, mc$ ) creates a model element  $me$  instance of the meta-class  $mc$ .
- *delete*( $me$ ) deletes the model element  $me$ . A deleted element does not have properties, nor references.
- *addProperty*( $me, p, v$ ) assigns the value  $v$  to the property  $p$  of the model element  $me$ .
- *remProperty*( $me, p, v$ ) removes the value  $v$  of the property  $p$  for the model element  $me$ .
- *addReference*( $me, r, met$ ) assigns a target model element  $met$  to the reference  $r$  for the model element  $me$ .

- *remReference*( $me, r, met$ ) removes the target model element  $met$  of the reference  $r$  for the model element  $me$ .



Figure 1: Sample UML model

```

1  create(C1,class)
2  addProperty(C1,name, 'TOTO')
3  addProperty(C1,visibility, 'private')
4  addProperty(C1,isabstract, 'false')
5  create(P1,property)
6  addProperty(P1, name, 'Alice')
7  addProperty(P1, visibility, 'public')
8  addReference(C1, attribute, P1)
9  addReference(C1, ownedElement, P1)
10 addReference(P1, class, C1)
11 addReference(P1, namespace, C1)
  
```

Figure 2: Praxis illustrative operation sequence

Figure 2 is a simplified *Praxis construction sequence*  $\sigma_c$  used to produce the model of Figure 1 that consists of a private UML class named *TOTO* which owns a public attribute *Alice*. The diagram comes from the Eclipse UML2 Tools Editor that does not display visibilities.

### 2.2 Inconsistency detection rules

In Praxis, an inconsistency rule is a logic formula over the sequence of model editing operations (interested readers can refer to [1] for a more detailed description of the rules' formalism). In these rules the following predicates are used to access model editing operations within a sequence:

- *lastCreate*( $id, MetaClass$ ) is used to fetch within the sequence, the operations that create model elements. The 'last' prefix of this logic constructor means that the element is never deleted further in the sequence.
- *lastAddReference*( $id, MetaReference, Target$ ) is used to fetch within the sequence, the operations that did assign to a source model element ( $id$ ), one value ( $Target$ ) for the reference ( $MetaReference$ ). The 'last' prefix of this logic constructor means that this reference value is not removed further in the sequence (with the *remReference* or *delete* operation).
- *lastAddProperty*( $id, MetaProperty, Value$ ) is used to fetch within the sequence, the operations that did assign to a source model element ( $id$ ), one value ( $Value$ ) for its property  $MetaProperty$ . The 'last' prefix of this logic constructor means that the value is not removed further in the sequence.

Let us illustrate inconsistency rules with two examples that are defined in the class diagram part of the UML 2.1 specification [11]:

The *Ownership* rule specifies that: *An element may not*

directly or indirectly own itself.

The **Visibility** rule specifies that: *An element that has a public visibility can only be in a namespace with a public visibility.*

These two rules cover two typical aspects of inconsistency detection. The rule **Ownership** (figure 4), specifies that containments should not cycle, which is quite complex to check because containments relationships are spread over the entire design. The rule **Visibility** (figure 3) on the contrary, is less complex, and has its locality restricted to each namespace.

$$\begin{aligned} \text{Visibility} &\iff \{\exists Me, N1 \cdot \\ &\text{lastAddProperty}(Me, \text{visibility}, \text{"public"}) \wedge \\ &\text{lastAddReference}(Me, \text{namespace}, N1) \wedge \\ &\text{lastAddProperty}(N1, \text{visibility}, V) \wedge \text{not}(V = \text{"public"})\}. \end{aligned}$$

**Figure 3: Visibility rule for Praxis**

$$\begin{aligned} \text{Ownership} &\iff \{\exists X \cdot \text{Owns}(X, X)\} \text{ With} \\ \text{Owns}(A, B) &\iff \{ \\ &\text{lastAddReference}(A, \text{ownedElement}, B) \vee \{\exists Y \cdot \\ &\text{lastAddReference}(A, \text{ownedElement}, Y) \wedge \text{Owns}(Y, B)\} \} \end{aligned}$$

**Figure 4: Ownership rule for Praxis**

To illustrate Praxis rules, we present the Praxis version of rule **Visibility** and **Ownership** in figure 3 and 4. The **Visibility** rule has two variables ( $Me, N1$ ) that both correspond to identifiers of UML elements. This rule matches any Praxis model construction sequence that results in a model containing an element with a public visibility for which its namespace is not public. The sequence presented in Figure 2 corresponds to an inconsistent model regarding this rule. Launching an inconsistency detection on this sequence would detect the inconsistency **Visibility(P1,C1)**.

### 2.3 Consistency Impact Matrix

The effect that editing operations may have on inconsistency rules can be described in a two dimensional boolean matrix we called **impact matrix** [2]. This matrix was designed for incremental inconsistency detection. Its rows correspond to Praxis operations and its columns to the inconsistency rules. A 'true' in a cell of such matrix means that the corresponding editing operation may have an effect on the corresponding inconsistency rule. When an editing operation appears in a model change (a model change is also a sequence of unitary operations), the inconsistency rules marked as 'true' in the matrix for these operations have to be re-checked. In our context this matrix can be used to know which information needs to be downloaded from the other views to check a particular rule.

As there is an infinity of possible editing operations (infinity of possible parameter's values), we group them in equivalence classes to bound the matrix's size. We defined four rules to partition the editing operations for any metamodel. Two editing operations are equivalent if (1) they create a model element instance of the same meta-class, (2) they

change a reference for the same meta-reference, (3) they change values for the same meta-property, (4) they delete a model element. Thanks to such a partition, the number of equivalence classes is bound to the number of the meta-classes, plus the number of the references, plus the number of properties, plus one (the equivalent class corresponding to the deletion). The **impact matrix** is automatically derived from the metamodel and the inconsistency rules [2].

Equivalence class	OwnedElement	Visibility
$C_{Class}$	false	false
$C_{Attribute}$	false	false
$SP_{Name}$	false	false
$SP_{Visibility}$	false	<b>true</b>
$SR_{OwnedElement}$	<b>true</b>	false
$SR_{Class}$	false	false
$SR_{Attribute}$	false	false
$SR_{Namespace}$	false	<b>true</b>
$Delete$	false	false

**Figure 5: Extract of the Impact matrix for the illustrative inconsistency rules (C=Create, SP=PropertyChange, SR=ReferenceChange)**

An extract of the **impact matrix** for the illustrative inconsistency rules we previously introduced is shown in figure 5. The extract shows nine equivalence classes from the metamodel partition, of which only three have an effect on an inconsistency rule. In this matrix, regarding the **Visibility** inconsistency rule, the operation classes that may have an impact on this rule are  $SP_{Visibility}$  and  $SR_{Namespace}$ . The operations that may have an impact on this rule are those that modify a reference to the namespace of an element, which all belong to the operation class  $SR_{Namespace}$ , and those that modify the visibility of an element, which all belong to  $SP_{Visibility}$ . Note that the  $Delete$  equivalence class has no impact because deletion occurs only for model elements that are not referenced and do not reference other model elements.

Our method for single-view consistency detection uses the **impact matrix** that was originally intended for incremental inconsistency detection [2]. For checking a particular ruleset, only operations members of the operation classes that have an impact on an inconsistency rules are relevant. Knowing which operation classes are interesting reduces the amount of data that needs to be considered to the relevant part of the views.

### 3. WHERE TO FETCH

Having the information about what classes of operations are useful for checking a particular rule is not enough in practice. It does not tackle the problem of gathering data that is actually related to the view for which we want to check the consistency. Indeed, the interesting information is scattered among the views, and not all of the information that belongs to the identified operation classes is interesting to the view's designer. In this section we describe how we use group of interest tables from peer-to-peer model editing engines to identify where in the net of views is the information to consider for inconsistency detection.



### 3.1 DPraxis Partial Replication Model

Partial replications allows each model designer to restrict his workspace to its strict minimum: his view. The overlaps between the views are maintained up-to-date thanks to a complex background update protocol. We defined such a protocol, named DPraxis, for distributed model editing tools [9]. DPraxis is a peer-to-peer fully distributed model update protocol that is designed to maintain the related views of a model up-to-date without having to replicate the entirety of its content in each designer’s workspace. In this section we revisit the use of this protocol, which is used by the method for single-view inconsistency detection.

In distributed systems using replication, replicated elements are called replicas [12]. A replica is accessed in the same way as a regular element, but it can be modified by the propagation of modifications issued from other sites. The replication is handled thanks to interest groups that are used to know which sites are interested in which information. Depending on the replication protocol and the data that needs to be replicated, the size and the nature of the replication unit can vary. An interest group links one replication unit to the sites that have it.

DPraxis uses the model element as the replication unit. Consequently, views contain replicas for the parts that overlap with at least one other view, and model elements that are proper to the view. The use of the model element as the replication unit allows to fine tune the parts of the model a designer want to import in his view. As DPraxis does not allow dangling links, reference between elements are shared between views only if both the source and the target of a reference are in the site’s view. This allows to choose the perimeter of the view by not importing elements pointed by unwanted references.

The DPraxis protocol maintains a table with routing information on each site to propagate changes to the interested sites. Thanks to this table it is possible to know which elements are replicas, and to know, for each replica, on which site it is replicated. This two informations are exploited by the single-view inconsistency detection method to contact other views when checking consistency.

### 3.2 Single-view consistency detection

The following method focuses on the use of partial replication. Nevertheless, the method described here for single-view inconsistency detection could be used in centralized architectures as well. We describe in this section how to check one inconsistency rule for one view, plus the elements of other views that are related to the considered view regarding the rule to check.

#### 3.2.1 Determining Jump Points

The intent of single view inconsistency detection is to simulate an inconsistency check where the detection engine can *jump* from the original view to the related ones. To achieve this goal we propose to begin with the identification of what we call *Jump Points*. A jump point is a replicated model element from which the inconsistency detection engine would

want to continue its work in an other view. They correspond to elements for which the local view only has a partial knowledge and where it is necessary to jump to other views to complete this knowledge. Inconsistency rules navigate through the model thanks to references, which can point to other views elements that are not replicated locally. The first step of single view inconsistency detection is then to find these local elements that may point to model elements that are not known locally, and may be the source of inconsistencies.

*Definition 1. Jump Point:* An inconsistency rule’s jump point is a model element which is the source (res. the target) of a reference that points to an element outside of the local view and that can be navigated by the inconsistency rule.

In Praxis, references are accessed using the logical predicate `lastAddReference(source, reference, target)`. The **Impact matrix** presented previously can be used to know which references can be crossed by an inconsistency rules. These references correspond to *true* in the matrix for operation classes of type  $SR_{reference}$ . For instance the rule *Visibility* only navigates through relations of the class  $SR_{Namespace}$ . In DPraxis, replicas are maintained up-to-date thanks to interest group tables that can be used to know which elements are replicated, and where.

In practice we determine the jump points following this procedure:

1. Determine the references that can be crossed by the inconsistency rule by looking for *true* in the **Impact Matrix** for classes of operations that correspond to references.
2. Determine which elements are replicas using the group of interest table.
3. Confront the two informations to find all the replicas for which there is a locally crossable reference. These elements will be considered as the only interesting Jump Points.

It is difficult in practice to determine all the possible Jump Points using DPraxis because references in the view can only target elements of the view — if a reference is interesting to one view, then its target has to be imported. Therefore, there can be references of replicas that are not known to the local view. We chose to ignore such jump points because in DPraxis the user decides whether the target (res. source) of a replicated element is interesting or not. It is then reasonable to hide inconsistencies for which the user clearly stated that the reference was not relevant to him/her. We only consider as a jump point elements for which there is at least one reference of the considered type locally. Nevertheless the Jump points we ignore could be capture by looking out in the meta-model if the replica’s type can admit crossable references.

### 3.2.2 Simulating Jumps

The next step of the single-view inconsistency method is to allow inconsistency rules to follow jump points' references to other views. We chose to download data of other views locally for simulating the jumps. This choice allows to use a regular inconsistency detection engine to do the single-view inconsistency detection. Not all the information from the distant view is to be downloaded. First, only information that is interesting to the considered rule matters, which is specified in the **Impact matrix**. Second, only elements that are linked to the jump point need to be downloaded. The gathering of the unitary actions for realizing the single-view inconsistency detection can be done this way:

1. Determine Jump points following the previous procedure.
2. Contact all the sites that have a replica of a jump point for the considered rule. Ask them to send all the unitary operations that are interesting for this rule plus the operations that are necessary to fulfill their preconditions. Wait for the answer.
3. Contacted sites then recursively apply the two previous steps of the procedure as long as new jump points are discovered. Then they send back to the caller all the gathered data.

The implementation of the procedure is detailed in section 4. At the end of this procedure the calling site has gathered all the information that is interesting to the rule from the other views, and that is linked to its jump points. An inconsistency detection on the view plus the gathered data implements a single-view inconsistency detection: It detects the local inconsistencies, plus the inconsistencies that are due to the overlapping parts of the views.

### 3.3 Example

In this section we present an illustrative example of the single-view inconsistency detection.

In figure 6 is shown three sequences corresponding to three different views, where each of the views is consistent, but together make a cycle for the `ownedElement` relationship. The number correspond to a total ordering of the actions that is granted by D-Praxis.

Let us consider a single-view inconsistency detection for *View One* regarding the *Ownership* inconsistency rule.

The first step is to determine the jump points: *P1* and *P2* are replicas (they are in the routing table from figure 7). From the **Impact Matrix** in figure 5 we can deduce that this rule can only cross references from the operation class *SR<sub>OwnedElement</sub>*. There is a local *OwnedElement* reference in the view that both concerns *P1* and *P2*, thus the jump points for this view and for the rule *Ownership* are *P1* and *P2*.

The simulation of the jump needs to contact *View two* concerning *P2* and *View three* concerning *P1*. The following requests are sent in order to gather the data without cycling indefinitely:

View One	
1	create(P1,package)
2	addProperty(P1,name, 'A')
3	create(P2,package)
4	addProperty(P2, name, 'B')
5	addReference(P1, ownedElement, P2)
6	addReference(P2, namespace, P1)
View Two	
3	create(P2,package)
4	addProperty(P2, name, 'B')
7	create(P3,package)
8	addProperty(P3,name, 'C')
9	addReference(P2, ownedElement, P3)
10	addReference(P3, namespace, P2)
View Three	
1	create(P1,package)
2	addProperty(P1,name, 'A')
7	create(P3,package)
8	addProperty(P3,name, 'C')
11	addReference(P3, ownedElement, P1)
12	addReference(P1, namespace, P3)

Figure 6: Three distributed views

P1-> View One, View Three  
P2-> View One, View Two  
P3-> View Two, View Three

Figure 7: Simplified routing table

- Request from *View one* to *View two*: All actions in *SR<sub>OwnedElement</sub>* regarding *P2, P1*, plus dependencies.
  - Identified operations: 9: `addReference(P2, ownedElement, P3)`.
  - Dependencies: 7: `create(P3,package)`.
  - Recursive Call from *View two* to *View three*: *P3* is identified as a new jumping point. Requesting all actions in *SR<sub>OwnedElement</sub>* regarding *P1, P2, P3*, plus dependencies.
    - \* Identified operation: 11: `addReference(P3, ownedElement, P1)`
    - \* Dependencies: None. *P3* and *P1* are known to the caller.
    - \* Recursive Call: None, no new jump point discovered.
  - Returned operations: 7: `create(P3,package)`; 9: `addReference(P2, ownedElement, P3)`; 11: `addReference(P3, ownedElement, P1)`.
- Request from *View one* to *View three* is symmetric to the previous one, it returns the same operations.

At the end of the procedure, the *view one* has gathered:

7	create(P3,package)
9	addReference(P2, ownedElement, P3)
11	addReference(P3, ownedElement, P1)

Using both the local and the distant operations, the inconsistency detection engine will detect an inconsistency for the

rule *Ownership*, indeed  $P1, P2, P3$  are in a cycle for the relation *OwnedElement*. The procedure only gathers the minimal information that matters for the rule, and that has a link to the local data. The designer of *view one* is now aware of the cycle problem and can contact the designers responsible for the two other implicated views to resolve the problem.

### 3.4 Limitations

The method described here is based on the fact that inconsistency rules navigate through models using the relations between model elements. This condition does not hold in general, for instance a rule that counts elements of a particular type may not use references for the navigation. Rules that do not use navigation must be checked by gathering all the interesting operations from all the other views.

## 4. VALIDATION

We implemented the method in our inconsistency detection tool <sup>1</sup> that runs on top of Eclipse. The implementation adds one optimization to the described procedure. We implemented a hashing system to avoid to constantly download the same operations over and over: A hash of the known operations for each operation class is sent prior to the procedure, if the hashes matches, the procedure does not continue.

## 5. RELATED WORK

The naive method to tackle with distributed views consistency consists in downloading all the views on a central repository and then perform the inconsistency detection on the re-united global design [10]. This technique is acceptable when the latency of the technique is not an issue, or when the consistency of the entire design needs to be assessed. Nevertheless, it does not scale to large designs regarding answer speed because: First, downloading all the views on one central place is slow if the views are large. Second, the merge of all the distant view into one big model is not easy. To address this problem this paper introduces the notion of single-view inconsistency detection, which detects distributed inconsistencies in a light-weight fashion.

Unfortunately we could not find many research approaches that address the problem of inconsistency detection in distributed model. However, this general problem is not new, traceability maintenance systems in distributed developing environment that can manage dependencies and ensure traceability between documents were developed in a quite centralized fashion [7]. The consistency in this tool was managed at a code and documentation level, plus a monitor system was available for user defined dependency needs.

## 6. CONCLUSION

This paper presents the notion of single-view inconsistency detection. This method aims at detecting distributed inconsistencies in a light-weight fashion by only reasoning on elements that are relevant for both the considered inconsistency rules and the considered view. The method is based on the notion of inconsistency rules' jump point which is computed thanks to two of our previous contributions [1, 9].

<sup>1</sup><http://meta.lip6.fr>

The next step of this work is a quantitative evaluation of the method to assess its performances both empirically and theoretically.

## 7. REFERENCES

- [1] X. Blanc, A. Mougnot, I. Mounier, and T. Mens. Detecting model inconsistency through operation-based model construction. In Robby, editor, *Proc. Int'l Conf. Software engineering (ICSE'08)*, volume 1, pages 511–520. ACM, 2008.
- [2] X. Blanc, A. Mougnot, I. Mounier, and T. Mens. Incremental detection of model inconsistencies based on model operations. In *Proceedings of the 21st International Conference on Advanced Information Systems, CAISE'09*, pages 32–46. Springer, 2009.
- [3] A. Egyed. Instant consistency checking for UML. In *Proceedings Int'l Conf. Software Engineering (ICSE '06)*, pages 381–390. ACM Press, 2006.
- [4] M. Elaasar and L. Brian. An overview of UML consistency management. *Technical Report SCE-04-18*, August 2004.
- [5] A. C. W. Finkelstein et al. Inconsistency handling in multiperspective specifications. In *IEEE Trans. Softw. Eng.*, volume 20, pages 569–578. IEEE Press, 1994.
- [6] P. Fradet, D. Le Metayer, and M. Pein. Consistency checking for multiple view software architectures. In *Proc. Joint Conf. ESEC/FSE'99*, volume 41, pages 410–428. Springer, September 1999.
- [7] D. Leblang and R. Chase Jr. Computer-aided software engineering in a distributed workstation environment. *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 104–112, 1984.
- [8] T. Mens et al. Detecting and resolving model inconsistencies using transformation dependency analysis. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 200–214. Springer, October 2006.
- [9] A. Mougnot, X. Blanc, and M.-P. Gervais. D-praxis : A peer-to-peer collaborative model editing framework. In *Distributed Applications and Interoperable Systems, 9th IFIP WG 6.1 International Conference, DAIS 2009, Lisbon, Portugal, June 9-11, 2009. Proceedings*, pages 16–29, 2009.
- [10] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proc. Int'l Conf. Software Engineering (ICSE'03)*, pages 455–464, Washington, DC, USA, 2003. IEEE Computer Society.
- [11] OMG. Unified Modeling Language: Super Structure version 2.1, january 2006.
- [12] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.
- [13] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.
- [14] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380, 2001.

# Resolving Model Inconsistencies with Automated Planning

Jorge Pinna Puissant, Tom Mens  
Université de Mons  
20 Place du Parc  
7000 Mons, Belgique  
{jorge.pinnapuissant,  
tom.mens}@umons.ac.be

Ragnhild Van Der Straeten  
Vrije Universiteit Brussel  
1050 Brussel, Belgium  
Université Libre de Bruxelles  
1050 Bruxelles, Belgique  
rvdstrae@vub.ac.be

## ABSTRACT

Various approaches have been explored to detect and resolve software model inconsistencies in a generic and scalable way. In this position paper, we outline our research that aims to use the technique of *automated planning* for the purpose of resolving model inconsistencies. We discuss the scalability results of the approach obtained through several stress-tests and we propose several alternatives to the automated planning approach.

## Keywords

inconsistency resolution, UML models, automated planning, scalability

## 1. INTRODUCTION

In *model-driven software engineering (MDE)* [24, 26], model inconsistencies inevitably arise, because a (software) system description is composed of a wide variety of diverse models, some of which are developed and maintained in parallel, and most of which are subject to continuous evolution. Our research focuses on the *resolution* of inconsistencies. The inconsistency resolution activity is divided into the following steps: (1) *Select* the inconsistencies that need to be resolved; (2) Identify possible *resolution plans* to resolve the selected inconsistencies; (3) Perform a *cost-benefit analysis* of the implementation of each of these resolution plans; (4) Select and apply resolution actions, based on the previous choices [25]. We focus on how to automate step (2) of the inconsistency resolution activity: identification of possible resolution plans. To do this, we propose to use the *Automated Planning* technique from the Artificial Intelligence domain.

In this article we give an overview of different automated planning techniques (Section 3). Based on a simple case study (Section 4.1) we present an approach using a forward-chaining heuristic planner to resolve inconsistencies (Section 4.2). One of our requirements is that the time required for resolving inconsistencies has to be sufficiently small so as not to disturb the designer in his/her work. Therefore, we investigate the scalability of the approach to larger software models (Section 4.3). Based on these results we discuss ways to improve the scalability of the proposed technique (Section 4.4). We also discuss alternatives to automated planning that may be more appropriate (Section 5).

## 2. PROBLEM STATEMENT

State-of-the-art approaches on inconsistency resolution exhibit several problems. In [18], resolution rules are specified manually, which is an error-prone process. Automatic generation of inconsistency resolution actions would resolve this problem. This is what is done by Nentwich et al. [19], by generating resolution actions automatically from the inconsistency detection rules. The execution of these rules, however, only resolves one inconsistency at a time. As recognised by the authors, this may cause problems when inconsistencies and their resolution are interdependent [17]. An additional problem is the interaction of the resolutions with the syntactical constraints imposed by the modelling language. Xiong et al. [28] define a language in which it is possible to specify the inconsistency rule and the possibilities to resolve the inconsistencies. This requires inconsistency rules to be annotated with resolution information. Almeida da Silva et al. [1] propose an approach to generate resolution plans for inconsistent models. The approach is based on the extension of inconsistency detection rules with information about the causes of the inconsistency, and on the use of generator functions, which are manually written and are used to generate resolution actions. Instead of explicitly defining or generating resolution rules, a set of models satisfying a set of consistency rules can be generated and presented to the user. Egyed et al. [6] define such an approach for resolving inconsistencies in UML models. Given an inconsistency and using choice generation functions, possible resolution choices, i.e., possible consistent models, are generated. The choice generation functions are dependent on the modelling language, i.e., they take into account the syntactical constraints of the modelling language and they only consider the impact of one consistency rule at a time. Furthermore these choice generation functions need to be implemented manually.

Our aim is to tackle the problem of inconsistency resolution by generating possible resolution plans without the need of manually writing resolution rules or writing any procedures that generate choices. The approach needs to generate valid models with respect to the modelling language and needs to enable the resolution of multiple inconsistencies at once and to perform the resolution in a reasonable time. In addition, the approach needs to be generic, i.e. it needs to be easy to apply it to different modelling languages. In this article we investigate the use of *Automated Planning* for this purpose.

### 3. PLANNING TECHNIQUES

Automated planning is a technique coming from artificial intelligence research. It aims to create *plans*, which are sequences of primitive actions that lead from an initial state to a state meeting a specific predefined goal. To accomplish this, the planner decomposes the world into logical conditions and represents a state as a conjunction of literals. As input the planner needs a *planning environment*, composed of an initial state, a desired goal and a set of primitive actions that can be performed. The initial state represents the current state of the world. The goal is a partially specified state that describes the world that we would like to obtain. The actions express how each element of a state can be changed. The actions are composed of a *precondition* and an *effect*. The effect of an action is executed if and only if the precondition is satisfied.

*Classical planning* is an automated planning subset that aim to find a sequence of actions that reaches a desired state in a finite, static, deterministic and fully observable world. In general a planning approach consists of a representation language used to describe the problem and an algorithm representing the mechanism to solve the problem.

Fikes *et al* [7] developed, in 1971, a formal planning representation language called *STRIPS* (*STanford Research Institute Problem Solver*). In 1989, Pednault [21] developed a more advanced and expressive language called *ADL* (*Action Description Language*, not to be confused with Architecture Description Language). ADL has an improved expressiveness compared to STRIPS. In particular, ADL applies the open world principle: unspecified literals are considered as unknown instead of being assumed false. ADL also allows to use negative literals and disjunction, whereas STRIPS only allows positive literals and conjunctions. In recent years a standard PDDL (*Planning Domain Definition Language*) [10] has been developed for the *International Planning Competition* (IPC) of the *International Conference on Artificial Intelligence Planning and Scheduling* (ICAPS). PDDL is a generic language allowing to represent the syntax of STRIPS, ADL and other languages. Even if PDDL covers all the functionalities of these languages, the majority of planners only implement the STRIPS subset [14]. The most recent version of PDDL is version 3.0 [9]. This language is used in the competition to compare the benchmarks of different planning approaches [23].

Two main approaches exist to solve classical planning problems [14]: (1) translating the planning problem into a problem that can be solved by a different approach (e.g. a boolean satisfiability problem, a constraint satisfaction problem, or a model checking problem); (2) generating a search space (which can be either a state space, a plan space, or a planning graph) and looking for a solution plan in this space. We will focus on this second approach only. Depending on the direction in which the state space is traversed to look for a solution, we can distinguish between:

*Progression planning* is a *forward search* that starts in the initial state and tries to find a sequence of actions that reaches a goal state. The problem of this algorithm is that it does not exclude irrelevant actions. An action is considered relevant if it can achieve the goal or one of the conjuncts of

the goal.

*Regression planning* is a *backward state-space search* that starts in the goal state and searches a sequence of actions that reach the initial state. This algorithm avoids the problems of the previous one by working only with relevant actions. The problem of this algorithm is that it is not always obvious to find a possible predecessor of an action.

Another distinction can be made between *total-order* and *partial-order* planning. With the former approach, the set of actions that composes the strategy found by the algorithm is strictly linear and ordered from the initial state to the goal. This category of algorithms cannot execute different actions simultaneously and cannot take advantage of the subdivision of a goal. Instead, *partial-order* planning (POP) explores the plan-space without committing to a totally ordered sequence of actions. POP works back from the goal to the initial state and it can place two actions into a strategy without specifying which comes first. As a result, these actions can be executed in parallel and their order is unimportant because they achieve different sub-divisions of the goal [22, 23]. Neither total-order nor partial-order is efficient without a good heuristic function that estimates the distance from a state to the goal.

Many planners exist that implement some variant of a planner algorithm. In this article we use the heuristic state-space progression planner called *FF* (for “Fast-Forward Planning System” [12, 13]). It is considered by [23] as the “The most successful state-space searcher”, and was awarded for *Outstanding Performance* at the AIPS 2000 planning competition and *Top Performer* at the AIPS 2002 planning competition. FF has been chosen not only because of its performance, but also because it uses PDDL language with full ADL subset support, including positive and negative literals, conjunction and disjunction, negation, typing, and logic quantification in the desired goal. This is crucial to our approach, as will be explained in the next section.

## 4. AUTOMATED PLANNING IN ACTION

### 4.1 Case Study

Design models can be of different types (e.g. UML, Petri nets, feature models, business process models). In this article we restrict ourselves to UML class diagrams [20]. They can suffer from many kinds of inconsistencies, such as structural and behavioural inconsistencies. Figure 1 illustrates a simple class diagram containing two structural inconsistency occurrences of type “inherited cyclic composition” and two occurrences of type “cyclic inheritance” [27].

An inherited cyclic composition inconsistency arises when a composition relationship and an inheritance chain form a cycle that would produce an infinite containment of objects upon instantiation. Both occurrences, ICC1 and ICC2, of this inconsistency in Figure 1 arise with the same composition relationship, between **Vehicle** and **Amphibious Vehicle**, but with different inheritance chains. The first occurrence ICC1 appears in the inheritance chain **Vehicle** ← **Boat** ← **Amphibious Vehicle**. The second inconsistency ICC2 occurs in the inheritance chain **Vehicle** ← **Car** ← **Amphibious Vehicle**. A cyclic inheritance inconsistency arises when an inheritance chain forms a cycle. Figure 1 has two occur-

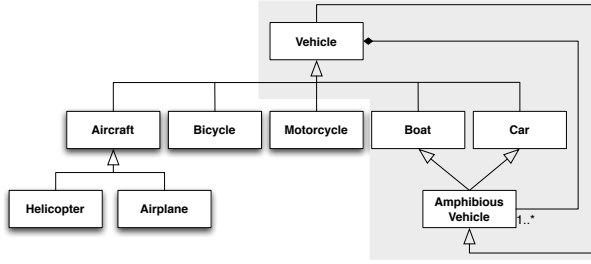


Figure 1: Class diagram with 4 inconsistency occurrences, inspired by [27].

rences **CI1** and **CI2** of this type of inconsistency. The first occurrence **CI1** forms an inheritance cycle that involves the classes **Vehicle**, **Boat** and **Amphibious Vehicle**. The second occurrence **CI2** forms an inheritance cycle that involves the classes **Vehicle**, **Car** and **Amphibious Vehicle**.

All four aforementioned inconsistency occurrences share two of the three classes that compose their respective inheritance chains: **Vehicle** and **Amphibious Vehicle**. Because of this overlap, it is possible to resolve more than one inconsistency occurrence with the same resolution action. For example, removing the composition relationship between **Vehicle** and **Amphibious Vehicle** solves the two inconsistency occurrences **ICC1** and **ICC2**. Removing the inheritance relationship between **Boat** and **Amphibious Vehicle** solves the two inconsistency occurrences **ICC1** and **CI1**. This clearly illustrates that, in order to resolve model inconsistencies in an optimal way, it is important to consider all inconsistencies simultaneously. In [17, 18], the impact of dependencies between model inconsistencies and their resolution actions were studied using the notion of critical pair analysis of graph transformation rules.

## 4.2 Planning for Inconsistency Resolution

Using the example of Figure 1, we illustrate how to create a sequence of inconsistency resolution actions with automated planning. We require as input an initial state (the inconsistent model), a set of possible actions (that change the model) and a desired goal (the consistent model). Planning requires logic conditions as input, so the whole model environment (*e.g.* model, meta-model, detection rules) is translated into a conjunction of logic literals. The syntax of PDDL is Lisp-like. Each logic literal is a tuple represented between parentheses. The tuple starts with the name of the literal, followed by pairs of variable names and their type (separated by a “-”). There are no primitive types in PDDL. More information about the PDDL syntax can be found in [8].

The **initial state** is expressed as a conjunction of literals, and represents the current world. In our case the initial state will be the inconsistent model. We can choose between three different representations of this initial state: (1) using the complete model; (2) using a partial model that contains only those elements that are involved in one or more inconsistency occurrences; (3) using a partial model that contains only those elements that are involved in a single inconsistency occurrence. We exclude option (3) as it only allows us to solve one inconsistency at a time, and it does not take into

account any dependency between inconsistency occurrences or their resolution actions.

The metamodel for our class diagram is given below using PDDL syntax. Each metamodel element is represented by a unique id through which it can be referred.

```

(Class ?id - class_id ?name - String)
(Generalisation ?id - g_id ?label - String
 ?child_class - class_id ?parent_class - class_id)
(Association_End ?id - ae_id ?class - class_id ?role - String
 ?upper_mult - Cardinal ?lower_mult - Cardinal
 ?composite - Boolean)
(Association ?id - a_id ?name - String ?ass_end_1 - ae_id
 ?ass_end_2 - ae_id)
  
```

A partial model conforming to this metamodel is given below. It contains only the elements that are involved in the inconsistency occurrences. This is illustrated by the shaded part of Figure 1.

```

(Class c1 Vehicle)
(Class c5 Boat)
(Class c6 Car)
(Class c9 Amphibious_Vehicle)
(Generalisation g4 label14 c5 c1)
(Generalisation g5 label15 c6 c1)
(Generalisation g8 label18 c9 c5)
(Generalisation g9 label19 c9 c6)
(Generalisation g10 label10 c1 c9)
(Association_End ae1 c9 role1 star one non)
(Association_End ae2 c1 role2 one one yes)
(Association a1 ass1 ae1 ae2)
  
```

The **set of actions** that can be performed to change a model are represented in terms of a *precondition* that must hold before the execution and the *action* to execute. In our approach, inspired by [2], the set of actions corresponds to the elementary operations (basically, create, modify and delete) of the different types of model elements that can be derived from the metamodel. These elementary operations, combined with the logic literals of the metamodel, allow us to compute the list of all possible actions. As an example, the specification of **modify\_Association\_Name** is given below.

```

(:action modify_Association_Name
 :parameters (?id - id ?name - String ?ass_end_1 - ae_id
 ?ass_end_2 - ae_id ?new_name - String)
 :precondition (Association ?id ?name ?ass_end_1 ?ass_end_2)
 :effect (when (not (= ?name ?new_name))
 (and (not (Association ?id ?name ?ass_end_1 ?ass_end_2))
 (Association ?id ?new_name ?ass_end_1 ?ass_end_2)))
 )
  
```

The **desired goal** is a partially specified state, represented as a conjunction of literals using logic quantification. It specifies the objective that we want to reach, namely a consistent model. To represent this consistent model we can use two alternatives: (1) the negation of the inconsistency detection rules; (2) or the negation of the inconsistency occurrences. An inconsistency detection rule is a conjunction of literals representing a pattern that, if matched in the model, detects inconsistency occurrences.

The inherited cyclic composition inconsistency detection rule using the PDDL syntax is given below. Observe that it only specifies an inheritance chain involving three classes. PDDL syntax does not allow to express transitive closure to make the rule more generic.

```

(exists (?a - class_id ?b - class_id ?c - class_id)
 (and
 (exists (?g - g_id ?Label - g_label)
 (Generalisation ?g ?Label ?c ?a))
 (exists (?g - g_id ?Label - g_label)
 (Generalisation ?g ?Label ?b ?c))
 )
  
```

```

(exists (?ae - ae_id ?role - ae_role
        ?upper - upper_cardinal ?lower - lower_cardinal)
 (Association_End ?ae ?a ?role ?upper ?lower yes))
(exists (?ae - ae_id ?role - ae_role
        ?upper - upper_cardinal ?composite - boolean)
 (Association_End ?ae ?b ?role ?upper one_1 ?composite))
))

```

The advantage of using alternative (1) above is that it can be used to detect and resolve inconsistency occurrences at the same time. Alternative (2) will only be able to resolve inconsistency occurrences that have already been identified previously. On the other hand, as we will see later, alternative (1) suffers from severe scalability problems. In both alternatives we use logic negation to express the fact that we do not want inconsistencies in the model. Because negation of the conjunction of literals is used we need a planning approach that allows the use of disjunction and negative literals in the goal. This is one of the main reasons why we have selected FF as a planning tool for our experiments.

The **plan** is a sequence of actions that reaches the desired goal. It is generated automatically by the domain independent planning algorithm. A complete resolution plan that solves the four inconsistency occurrences of the motivating example is shown in Figure 2. Remark that our approach prohibits the generation of a resolution plan that leads to ill-formed models (i.e., models that do not conform to their metamodel).

<pre> delete_Generalisation :     (Generalisation g10 label10 c1 c9) modify_Association_End_Lower_Multiplicity :     from: (Association_End ae1 c9 role1 star one non)     to:   (Association_End ae1 c9 role1 star zero non) </pre>
--

Figure 2: Complete resolution plan that resolves all four inconsistency occurrences.

### 4.3 Scalability Study

There are different ways in which to specify the input for the automated planning algorithm. To specify the initial state, we can either use the complete model or we can restrict the search space by using a partial model that contains only those elements that are involved in the inconsistency occurrences. To specify the desired goal, we can choose between using the negation of the inconsistency detection rules or using the negation of the inconsistency occurrences themselves.

In order to assess which of the above four choices produces the best results, we compared the timing results of each considered possibility. In order to remove noise, each experiment was executed 10 times and the average time and standard deviation was computed. All experiments were carried out on a 64-bit Apple MacBook with 2.4 GHz Intel Core 2 Duo processor and 4GB RAM, 2.9GB of which were available for the experiment.

The experiments using the complete model as initial state and the negation of the inconsistency detection rules as desired goal and using the partial model as initial state and the negation of the inconsistency detection rules as desired goal, ran out of memory. Using the complete model as initial state and the negation of the inconsistency occurrences

as desired goal, the resolution plan of Figure 2 was generated in 14.84 seconds on average with a very low standard deviation of 0.09 seconds. Using a partial model as initial state, and the negation of the inconsistency occurrences as desired goal, the resolution plan of Figure 2 was generated in 0.268 seconds on average, with a standard deviation of 0.004 seconds.

To verify whether the proposed approach scales up to larger models, we have stress-tested both of the successful experiments (using the negation of the inconsistency occurrences as desired goal). Again each experiment was executed 10 times and the average time and standard deviation was computed.

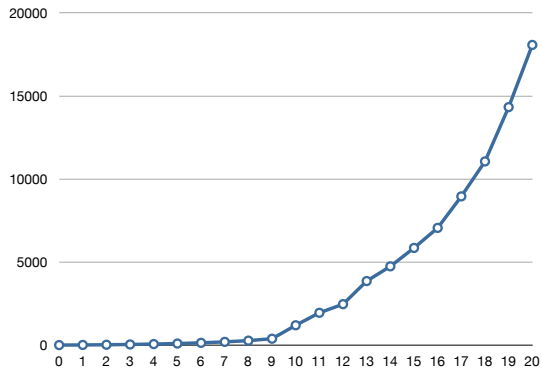
First, we artificially augmented the size of the motivating example of Figure 1 by adding an increasing number of isolated classes to the model (from 1 class to 20 classes). Since these classes are unrelated to the inconsistency occurrences that the algorithm needs to resolve, the algorithm is still able to find the same resolution plan and the partial model is left untouched. However, the time it takes to generate a plan increases as the model size increases.

Figure 3a illustrates the timing results if we use the *complete model* as initial state. It takes only 15 seconds for our initial example, but it takes more than 5 hours for the model with 20 more added classes. A regression analysis reveals an *exponential* relation with coefficient of determination  $R^2 = 0.982$ , indicating a very good fit of the regression model. Two other candidate regression models we verified had a lower goodness of fit: 0.977 for a quadratic polynomial model and 0.884 for a power curve. These results show that using a complete model as initial state does not scale up to larger models.

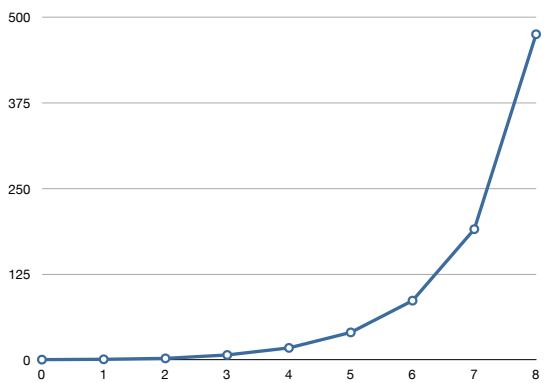
Secondly we studied the timing results when the size of the *partial model* increases. The motivating example of Figure 1 contains an inheritance chain of classes for the two types of considered inconsistencies. We artificially augmented the size of the model by increasing the length of the inheritance chains involved in the inconsistency occurrences. We did this gradually, by adding between one and eight intermediate superclasses, and computing the timing results for each partial model.

Figure 3b shows the timing results of carrying out this experiment. The figure shows a strong increase in time to compute the resolution plan as the size of the partial model increases. The standard deviation was always below 2%, and less than 0.4% on average. A regression analysis reveals an *exponential* growth (with coefficient of determination  $R^2 = 0.995$ ) in the time needed to find a resolution plan. Two other regression models we verified had a lower goodness of fit: 0.927 for a quadratic polynomial model and 0.949 for a power curve.

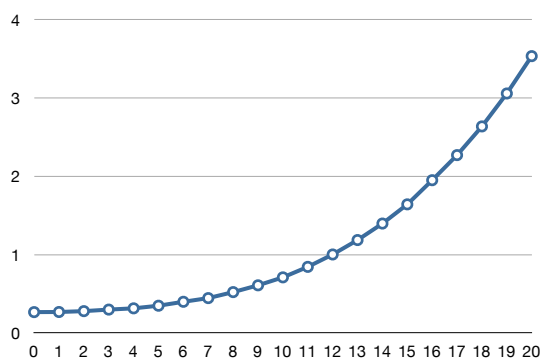
We also verified whether the number of inconsistency occurrences to be resolved affected the timing results. To achieve this, we reduced the desired goal by generating plans that resolve only 2 or 3 inconsistency occurrences, respectively. In all of these cases we found an *exponential* growth in time. We obtained a goodness of fit  $R^2 = 0.991$  for resolving 2 inconsistency occurrences, and  $R^2 = 0.992$  for resolving 3



(a) Using the complete model as initial state.



(b) Adding intermediate superclasses to the partial model.



(c) Adding class attributes to the partial model.

Figure 3: Scalability timing results (the y-axis represents the time in seconds).

inconsistency occurrences.

Finally, we verified whether the size of the metamodel affects the timing results. To achieve this, we added a new element to the metamodel:

```
(Attribute ?id - attribute_id ?class - class_id
?Name - String ?Type - Type)
```

This requires to add three new actions, to create, modify and delete attributes, respectively. It did not affect the timing results as long as attributes are not used in the initial state and the desired goal.

As a next step, we increased the size of the initial state by adding 1 to 20 attributes to the existing classes of the model. The desired goal was not modified. The standard deviation was 0,7% on average. The results are shown in Figure 3c. For an initial state with 1 attribute added the time was 0.27 seconds. After adding 20 attributes it was 3.5 seconds. A regression analysis revealed a *quadratic polynomial* with a goodness of fit  $R^2 = 0.994$ . Two other regression models we verified were an exponential model with  $R^2 = 0.982$  and a power model with  $R^2 = 0.763$ .

#### 4.4 Discussion

The exponential timing results obtained through the experiments described in the previous section, indicate that the approach is not usable in practice. Using the approach to resolve inconsistencies one by one would be feasible because the partial model and desired goal will remain relatively small. This is not a good solution, because it does not take full advantage of automated planning. In addition, inconsistency occurrences and their resolution actions are often interdependent. Another important limitation we encountered is the *expressiveness* of the PDDL syntax. It does not offer important features such as transitive closure, primitive types, numbers. In addition, literals cannot be modified (they have to be deleted and added again). A third limitation of our approach is that, currently, we generate only a single resolution plan. The resolution of several inconsistencies can give rise to several different resolution plans, i.e., different sequences of resolution actions leading to possibly different consistent models.

Several improvements to the approach can be envisaged. A first improvement is to adapt the planning algorithm so that it generates several resolution plans among which the model designer could choose. The scalability problem could be addressed by implementing a *domain-specific planner* that can be optimized by making it more specific and more performant for the specific problem we want to tackle. In addition, since we are not constrained by the PDDL syntax, this would solve the problems of expressiveness we encountered. The timing results could be improved by using *regression planning* as opposed to *progression planning* [23], as used by *FF*. Progression planning depends mainly on the size of the initial state and it does not exclude irrelevant actions. Regression planning works only with relevant actions. Because of this, the search space will be significantly smaller. Further experiments are needed to verify whether regression planning will be more appropriate for our needs.



## 5. BEYOND PLANNING

Since the automated planning approach does not meet our expectations, we would also like to study other techniques coming from the domain of artificial intelligence for the purpose of resolving modeling inconsistencies in an automated way.

Logic-based approaches have been used for different but related purposes in inconsistency resolution. Marcelloni and Akist [15, 16] used *fuzzy logic* to cope with methodological inconsistencies in design models. It remains to be seen whether this approach can be generalised to resolve any kind of model inconsistency. Castro et. al. [5] used *logic abduction* to detect and resolve inconsistencies in source code. Some preliminary results we carried out to apply this approach to resolve inconsistencies in design models appeared promising, but further work is necessary to assess whether the approach scales up and works in practice. Almeida da Silva et al. [1] implemented a Prolog program to generate resolution plans for inconsistent models. The approach is promising but still requires a lot of manual encoded input to specify the generator functions and the causes of the inconsistencies.

Harman [11] advocates the use of search-based approaches in software engineering. This includes a wide variety of different techniques and approaches such as metaheuristics (e.g. variable neighborhood search [3, 4]), local search algorithms, automated learning, genetic algorithms [23]. We believe that these techniques could be applied to the problem of model inconsistency management, because it satisfies at least three important properties that motivate the need for search-based software engineering: the presence of a large search space, the need for algorithms with a low computational complexity, and the absence of known optimal solutions.

In order to assess the adequacy of all these different approaches to inconsistency management, there is also an urgent need to define benchmarks allowing to compare them. Such a benchmark should contain at least a set of shared case studies on which to evaluate each approach; as well as a set of clearly identified criteria enabling the comparison of approaches and their quality.

## 6. CONCLUSION

In this article, we explored the use of *automated planning*, a logic-based approach originating from artificial intelligence, for the purpose of resolving model inconsistencies. We are not aware of any other work having used this technique for this particular purpose. The results of our experiments reveal that the approach is feasible but suffers from various scalability problems. We have discussed ways in which the scalability can be improved. We have also discussed alternative search-based techniques that may deal with inconsistency resolution in a scalable way.

**Acknowledgements.** This work has been partially supported by (i) the F.R.S. – FNRS through FRFC project 2.4515.09 “Research Center on Software Adaptability”; (ii) research project AUWB-08/12-UMH “Model-Driven Software Evolution”, an *Action de Recherche Concertée* financed by the *Ministère de la Communauté française - Direction générale de l’Enseignement non obligatoire et de la Recherche scientifique, Belgium*; (iii) Avec le soutien de Wallonie

- Bruxelles International et du Fonds de la Recherche Scientifique, du Ministère Français des Affaires étrangères et européennes, du Ministère de l’Enseignement supérieur et de la Recherche dans le cadre des Partenariats Hubert Curien.

## 7. REFERENCES

- [1] M. A. Almeida da Silva, A. Mougenot, X. Blanc, and R. Bendraou. Towards automated inconsistency handling in design models. In *CAiSE 2010, Lecture Notes in Computer Science*. Springer, 2010.
- [2] X. Blanc, A. Mougenot, I. Mounier, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Proc. Int’l Conf. Software Engineering (ICSE)*, volume 1, pages 511–520, 2008.
- [3] J. Brownlee. Variable neighbourhood search. Technical Report CA-TR-20100206-1, The Clever Algorithms Project <http://www.CleverAlgorithms.com>, February 2010.
- [4] G. Caporossi and P. Hansen. Variable Neighborhood Search for Extremal Graphs 1. The AutoGraphiX System. *Discrete Math.*, 212:29 – 44, 2000.
- [5] S. Castro, J. Brichau, and K. Mens. Diagnosis and semi-automatic correction of detected design inconsistencies in source code. In *IWST ’09: Proceedings of the International Workshop on Smalltalk Technologies*, pages 8–17, New York, NY, USA, 2009. ACM.
- [6] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Proc. Int’l Conf. Automated Software Engineering*, pages 99–108. IEEE, 2008.
- [7] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *2nd International Joint Conference on Artificial Intelligence.*, pages 608–620, 1971.
- [8] A. Gerevini and D. Long. BNF description of PDDL 3.0. <http://zeus.ing.unibs.it/ipc-5/>, October 2005.
- [9] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3 : The language of the fifth international planning competition. Technical report, Department of Electronics for Automation, University of Brescia, Italy, August 2005.
- [10] M. Ghallab, A. Howe, C. Knoblock, and D. McDermott. PDDL — the planning domain definition language. Technical Report DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut, 1998.
- [11] M. Harman. Search based software engineering. In *Computational Science - ICCS 2006*, volume 3994/2006 of *Lecture Notes in Computer Science*, pages 740–747. Springer Berlin / Heidelberg, 2006. Workshop on Computational Science in Software Engineering (CSSE’06).
- [12] J. Hoffmann. FF: The Fast-Forward Planning System. *The AI Magazine*, 2001.
- [13] J. Hoffmann and B. Nebel. The FF Planning System: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
- [14] S. Jiménez Celorrio. *Planning and Learning under Uncertainty*. PhD thesis, Universidad Carlos III de Madrid, 2010.

- [15] F. Marcelloni and M. Aksit. Leaving inconsistency using fuzzy logic. *Information and Software Technology*, 43(12):725 – 741, 2001.
- [16] F. Marcelloni and M. Aksit. Fuzzy logic-based object-oriented methods to reduce quantization error and contextual bias problems in software development. *Fuzzy Sets and Systems*, 145(1):57 – 80, 2004. Computational Intelligence in Software Engineering.
- [17] T. Mens and R. Van Der Straeten. Incremental resolution of model inconsistencies. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *Algebraic Description Techniques*, volume 4409 of *Lecture Notes in Computer Science*, pages 111–127. Springer-Verlag, 2007.
- [18] T. Mens, R. Van Der Straeten, and M. D’Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *Proc. Int’l Conf. Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, October 2006.
- [19] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proc. 25th Int’l Conf. Software Engineering*, pages 455–464. IEEE Computer Society, May 2003.
- [20] Object Management Group. Unified modeling language: Super structure version 2.1, january 2006.
- [21] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *1st International Conference on Principles of Knowledge Representation and Reasoning (KR’89)*, pages 324–332, 1989.
- [22] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In *3rd International Conference on Principles of Knowledge Representation and Reasoning (KR’92)*, pages 103–114, 1992.
- [23] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.
- [24] D. C. Schmidt. Guest editor’s introduction: Model-driven engineering. *IEEE Computer*, pages 25 – 31, February 2006.
- [25] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World scientific, 2001.
- [26] T. Stahl and M. Völter. *Model Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.
- [27] R. Van Der Straeten. *Inconsistency management in model-driven engineering: an approach using description logics*. PhD thesis, Vrije Universiteit Brussel, 2005.
- [28] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In H. van Vliet and V. Issarny, editors, *Proc. ESEC/FSE 2009*, pages 315–324. ACM, 2009.

# Tolerating Inconsistency in Feature Models

Bo Wang  
Key Laboratory of High  
Confidence Software  
Technologies  
(Ministry of Education)  
Peking University, China  
wangbo07@sei.pku.edu.cn

Zhenjiang Hu  
GRACE Center  
National Institute of  
Informatics  
Tokyo, Japan  
hu@nii.ac.jp

Yingfei Xiong  
Generative Software  
Development Lab  
The University of Waterloo  
Waterloo, Canada  
yingfei@swen.uwaterloo.ca

Haiyan Zhao  
Key Laboratory of High  
Confidence Software  
Technologies  
(Ministry of Education)  
Peking University, China  
zhhy@sei.pku.edu.cn

Wei Zhang  
Key Laboratory of High  
Confidence Software  
Technologies  
(Ministry of Education)  
Peking University, China  
zhangw@sei.pku.edu.cn

Hong Mei  
Key Laboratory of High  
Confidence Software  
Technologies  
(Ministry of Education)  
Peking University, China  
meih@pku.edu.cn

## ABSTRACT

Feature models have been widely adopted to reuse the requirements of a set of similar products in a domain. When constructing feature models, it is difficult to always ensure the consistency of feature models. Therefore, tolerating inconsistencies is important during the construction of feature models. The usual way of tolerating inconsistencies is to find the minimal unsatisfiable core. However, identifying the minimal unsatisfiable core is time-consuming, which decreases itself the practicability.

In this paper, we propose a priority based approach to tolerating inconsistencies in feature models efficiently. The basic idea of our approach is to find the weaker unsatisfied constraints, while keeping the rest of the feature model consistent. Our approach tolerates inconsistencies with the help of priority based operations while building feature models. To this end, we adopt the constraint hierarchy theory to express the degree of domain analysts' confidence on constraints (i.e. the priorities of constraints) and tolerate inconsistencies in feature models. Experiments have been conducted to demonstrate that our system can scale up to large feature models.

## Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software—*Domain engineering*

## Keywords

Feature Model, Constraint Hierarchy, Tolerate Inconsistency

## 1. INTRODUCTION

Feature models [6, 7] have been widely adopted to reuse the requirements of a set of similar products in a domain. During the process of products reuse, specific products that satisfy all the constraints are derived from feature models. Inconsistent feature models contain contradictory constraints that cannot be satisfied at the same time, leading to no valid products derivable from them [13]. However, it is difficult to always ensure the consistency of feature models, during the construction of feature models. Therefore, tolerating inconsistencies is important when constructing feature models.

The usual way of tolerating inconsistencies is to find the minimal unsatisfiable core in inconsistent feature models. However, identifying the minimal unsatisfiable core is time-consuming [9], which decreases itself the practicability.

In this paper, we propose a priority based approach to tolerating inconsistencies in feature models, and report an implementation of a system that not only automatically tolerates inconsistencies by identifying weaker unsatisfied constraints, but also supports domain analysts to handle the tolerated inconsistencies, with the help of priority based operations. To this end, we adopt the *constraint hierarchy theory* [5], a known practical theory in user interface construction, to express the degree of domain engineers' confidence on constraints (i.e. the priorities of constraints) and tolerate inconsistencies in feature models. The main contributions of our paper are summarized as follows:

- We show the importance of the constraint hierarchy theory in tolerating inconsistencies in feature models, and we adopt it to divide a feature model into the *consistent feature model* part and *pending constraint set* part, which will help tolerate inconsistencies in feature models.
- We make the first attempt of conducting a constraint hierarchy system<sup>1</sup> for tolerating inconsistencies in feature models, through adapting and extending an existing incremental algorithm-SkyBlue [10, 11].

<sup>1</sup>See <http://sei.pku.edu.cn/~wangbo07/> for more details.

- We have conducted the experiments on our system, which demonstrates that our approach scales up to very large feature models.

The rest of this paper is organized as follows. Section 2 introduces some preliminary knowledge on feature models, constraint hierarchy and SkyBlue. Section 3 gives an overview of our approach. Section 4 amplifies our approach. Section 5 illustrates the scalability of our approach. Section 6 discusses the related work, and Section 7 concludes the paper and highlights the future work.

## 2. PRELIMINARIES

In this section, we first describe feature models, followed by the introduction to the constraint hierarchy theory and SkyBlue. All these three serve as the fundamental supports for tolerating inconsistencies in feature models.

### 2.1 Feature Model

A feature model organizes the requirements of the products of a domain, in terms of features and the relationships between them. A simplified feature model of the mobile phone domain [3], which adopts our meta model of feature models [15], is shown in Fig. 1.

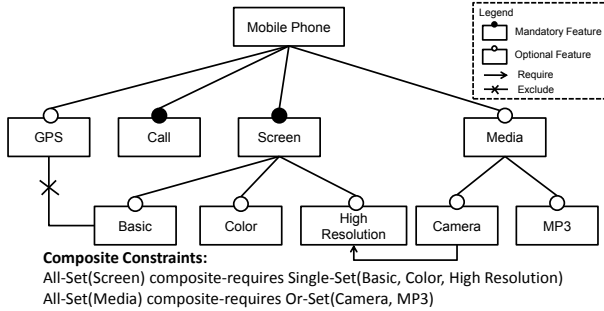


Figure 1: A simplified example of the mobile phone domain

A feature is a software characteristic with sufficient user or customer value, which essentially denotes a cohesive set of individual requirements [14]. In feature models, if a feature is bound (i.e. selected and implemented in a product), so is its parent. A *mandatory* feature should be bound if its parent is bound. An *optional* feature can be unbound (i.e. deselected and not implemented in a product), even if its parent is bound.

There are three kinds of *simple constraints* on two features, namely *requires*, *m-requires*, and *excludes*. If feature *A* *requires* feature *B*, it indicates that *B* must be bound when *A* is bound. If feature *A* *m-requires* feature *B*, it means that *A* and *B* should be bound or unbound at the same time. If feature *A* *excludes* feature *B*, it indicates that they cannot be bound at the same time.

There are three kinds of *predicates* on a set of features, namely *All*, *Alternative* and *Or*. Predicates *All*, *Alternative*, and *Or* indicate these predicates are true only if all, only one, and at least one features are bound in their feature sets, respectively. For instance, *Single-Set (Basic, Color, High Resolution)* indicates that this predicate is true when only one kind of screens can be chosen in a product.

Based on the predicates, there are three kinds of *composite constraints* on two feature sets, *composite-requires*, *composite-m-requires*, and *composite-excludes*. For example, given *All-Set(Media)* *composite requires Or-Set(Camera, MP3)*, if *All-Set (Media)* is true, *Or-Set (Camera, MP3)* must be true. For the details of the composite constraints, see sub-section 4.1.

Products are derived from a feature model by binding and unbinding constraints. A valid derived product must satisfy all the constraints in the feature model. A feature model contains inconsistencies if no valid products can be found to satisfy all the constraints in this feature model [13]. These inconsistencies are caused by the contradictory constraints in feature models.

### 2.2 Constraint Hierarchies and SkyBlue

When a solver is used to check inconsistent models, it is not enough for the solver to just signal the detected inconsistencies. The *constraint hierarchy theory* [5] provides a way to handle the detected inconsistencies through maintaining constraint hierarchies. A constraint hierarchy contains a set of constraints, each assigned with a priority, indicating the importance of the constraint. Given an inconsistent model, a constraint solver makes sure that stronger constraints are satisfied, through unsatisfying the contradictory weaker constraints.

SkyBlue is an incremental constraint solver that uses local propagation to maintain the constraint hierarchies. It has been successfully applied in many GUI systems. SkyBlue requires that methods can be derived from constraints (explained later in this sub-section), and thus is not applicable to some kinds of constraints. One important finding of our approach is that the constraints in feature models satisfy the prerequisite of SkyBlue (with minor extension) and thus can enjoy the performance boost of SkyBlue (see Section 4).

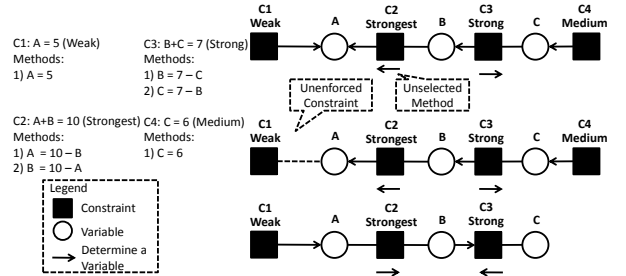


Figure 2: A simple example for SkyBlue

The input of SkyBlue is a set of variables and the constraints on these variables. The output of SkyBlue is a set of values that satisfy stronger constraints and leave the contradictory weaker constraints unsatisfied.

In SkyBlue, each constraint is equipped with one or more *methods*. SkyBlue satisfies a constraint by selecting one of its methods and executing the selected method. SkyBlue *enforces* a constraint by choosing one method for this constraint and *revoke* a constraint by choosing no methods for this constraint. A constraint is *enforced* if it has a selected method, otherwise, it is *unenforced*. The variables and the constraints form the constraint graph. The constraint graph, together with the selected methods, forms the method graph.

The output of SkyBlue, the value set for variables, is calculated through constructing and executing a *locally-graph-better* (called LGB) method graph. A method graph is LGB if there are no method conflicts and there are no unenforced constraints that could be enforced by revoking one or more weaker constraints (and possibly changing the selected methods for other enforced constraints with the same or stronger strength) [10].

As a simple example, the method graphs in Fig. 2 has four constraints  $C1$ ,  $C2$ ,  $C3$  and  $C4$  on three variables  $A$ ,  $B$  and  $C$ . Each constraint has one or more methods to make the constraint hold (for instance, two methods are given to satisfy  $C3$  by either calculating  $B$  from  $C$  or calculating  $C$  from  $B$ ). To satisfy every constraint, SkyBlue tries to select a method from each constraint, as shown in the upper right of Fig. 2, but there is a method conflict (inconsistency): variable  $A$  is determined by two methods, namely,  $A=5$  and  $A=10-B$ . To resolve this conflict, we have to revoke some weaker constraint to enforce the stronger constraints. SkyBlue finds the strong constraints that can be enforced, while leaving the weaker constraints unenforced by constructing LGB method graph. The LGB method graph of this example is shown in the middle right of Fig. 2, where  $C1$  is revoked. After executing the selected methods in the LGB method graph,  $A$  equals to 9,  $B$  equals to 1, and  $C$  equals to 6, which satisfy the three stronger constraints, namely  $C2$ ,  $C3$  and  $C4$ .  $C1$  may be reenforced automatically when its contradictory constraints are deleted. For example, if  $C4$  is deleted, a new LGB method graph is constructed, in which constraint  $C1$  is reenforced by selecting method “ $A$  equals to 5”, as shown in the lower right of Fig. 2.

### 3. APPROACH OVERVIEW

In this section, we first give an overview of our approach, and then we use an example to illustrate how to tolerate inconsistencies in feature models.

#### 3.1 Feature Model Inconsistency Tolerance

In our approach, a feature model is divided into two parts, namely the *consistent feature model* part (called CFM part) and the *pending constraint set* part (called PCS part). The PCS contains weaker constraints that conflict with some stronger constraints in the CFM. If the PCS is empty, the feature model is consistent. Domain analysts work on the CFM to construct the feature model and work on the PCS to handle the tolerated inconsistencies. An overview of the inconsistency tolerance is shown in Fig. 3.

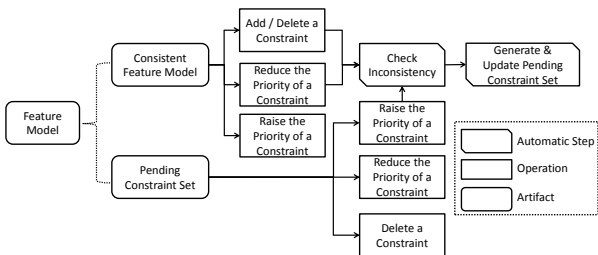


Figure 3: Feature model inconsistency tolerance

We divide a feature models into the CFM and the PCS through constructing LGB method graphs. The CFM consists of the enforced constraints in the LGB method graph,

and the PCS consists of the unenforced constraints in the LGB method graph. After a new LGB method graph is constructed, the CFM and the PCS are updated.

1. If the constructed LGB method graph does not contain any unenforced constraints, the PCS is empty and the CFM contains all the constraints in the feature model. At this moment, the feature model is consistent.
2. If the constructed LGB method graph contains one or more weaker unenforced constraints, the constraints in the PCS are replaced with these unenforced constraints and the constraints in the CFM are replaced with the enforced constraints in the LGB method graph. At this moment, the feature model is inconsistent.

Four kinds of operations on the CFM are provided to help domain analysts construct feature models. When constructing feature models, domain analysts can add a constraint with priority into the CFM or delete a constraint from it. Domain analysts can change priorities of constraints when constructing feature models.

There are three conditions on which the enforced constraints in the CFM may become unenforced and thus are put into the PCS: 1) their priorities are reduced; 2) the priorities of their contradictory weaker constraints in the PCS are raised; 3) some contradictory stronger constraints are added. When these conditions are met, we generate a new LGB method graph to update the CFM and the PCS.

Three kinds of operations on the PCS are provided to help domain analysts handle the tolerated inconsistencies. If the domain analysts become more confident about a constraint in the PCS, he can raise its priority. The possibility of reenforcing this constraint become larger as its priority rises. If the domain analysts become less confident about a constraint, he can reduce its priority. The possibility of enforcing this constraint becomes smaller as its priority decreases. If domain analysts believe some constraints do not represent the correct relationships among the features, they can delete them from the in pending constraint set.

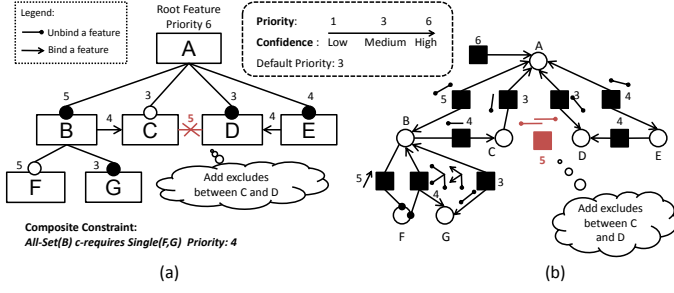
There are three conditions on which the unenforced constraints in the PCS can be re-enforced again, and thus are put into the CFM: 1) their priorities are raised; 2) their contradictory stronger constraints in the CFM are deleted; 3) the priorities of their contradictory stronger constraints in the CFM are reduced. When these conditions are met, we generate a new LGB method graph to update the CFM and the PCS.

#### 3.2 An Example

To demonstrate how we tolerate inconsistencies in feature models, let us see how to find the CFMs and the PCSs, and handle the tolerated inconsistencies in the feature model in Fig. 4.

Suppose all the constraints have been added into the feature model except “feature  $C$  excludes feature  $D$ ” (the red part in Fig. 4). The feature model is consistent before adding “feature  $C$  excludes feature  $D$ ”, since the LGB shown in Fig. 4(b) contains no unenforced constraints. At this moment, the PCS is empty. Note that even some variables are determined by more than one method in the LGB method graph, no conflict happens, because these variables are set to the same value (see Section 4 for more detail).

When the “exclude” constraint is added and enforced, a LGB method graph, in which the constraints “Mandatory  $D$ ”



**Figure 4: An example of feature model inconsistency tolerance**

and “feature  $E$  requires feature  $D$ ” are revoked, is generated. The PCS consists of these two revoked constraints.

Domain analysts can delete the constraint “feature  $D$  requires feature  $E$ ” if they believe the “require” constraint does not represent the correct relationship between feature  $D$  and  $E$ . If domain analysts have more confidence on the “Mandatory  $D$ ” than before, they raise its priority to 5. Then our approach will try to enforce it by constructing a new LGB. In the new LGB method graph, only “feature  $B$  requires feature  $C$ ” is revoked. The PCS is updated, and it only contains the “require” constraint.

## 4. TOLERATE INCONSISTENCIES IN FEATURE MODELS

In this section, we will describe how we adopt the constraint hierarchy theory by revising and extending SkyBlue to tolerate inconsistencies in feature models.

### 4.1 Map Feature Models to Constraint Graphs

To use SkyBlue to detect and tolerate inconsistencies, the first thing is to map the elements of feature models to the elements of SkyBlue constraint graphs.



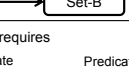
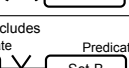

The mapping consists of two steps: 1) each feature of the feature model is mapped to a variable of the SkyBlue constraint graph; 2) each constraint of the feature model is mapped to a SkyBlue constraint (called SBC) that is represented by a set of methods.

SkyBlue cannot be generalized to derive methods from some “inequality-like” constraints. But feature models are different from this general case. In feature models, each feature can have only two states: 1) bound; 2) unbound. Therefore, it is possible to derive methods for constraints in feature models, through combinations of the states of “certain” features. Concrete rules for the mapping from feature models to constraint graphs are listed in Tables 1 and 2.




Binding a feature ( $Bind(feature)$ ) sets the bind state of the feature *bound*. Unbinding a feature ( $Unbind(feature)$ ) sets the bind state of the feature *unbound*. *Predicate* on a feature set represents the value of the predicate of a feature set. In our approach, simple constraints can be represented by a composite constraint. For example, “feature  $A$  excludes feature  $B$ ” can be represented as “*All-Set(A) composite-excludes All-Set(B)*”.

In Table 2, each kind of group predicates is associated with a set of methods that can be executed to set the predicate *True* or *False*. These predicate methods, together with the composite constraint methods, can map a com-

**Table 1: Methods for constraints**

Relationship	Number of Methods	Methods
Mandatory 	2	{Bind(A), Bind(B)} or {Unbind(A), Unbind(B)}
Optional 	2	{Bind(A)} or {Unbind(B)}
Composite-Requires 	2	{Predicate(Set-A) = False} or {Predicate(Set-B) = True}
Composite-M-requires 	2	{Predicate(Set-A) = False, Predicate(Set-B) = False} or {Predicate(Set-A) = True, Predicate(Set-B) = True}
Composite-Excludes 	2	{Predicate(Set-A) = False} or {Predicate(Set-B) = False}

**Table 2: Methods for predicates**

Predicate	Value	Number Of Methods	Methods
All 	True	1	{Bind(A <sub>1</sub> ), Bind(A <sub>2</sub> ) ... Bind(A <sub>n</sub> )}
	False	n	{Unbind(A <sub>1</sub> )} or {Unbind(A <sub>2</sub> )} or ... {Unbind(A <sub>n</sub> )}
Alternative 	True	n	{Bind(A <sub>1</sub> ), Unbind(A <sub>2</sub> ), Unbind(A <sub>3</sub> ) ... Unbind(A <sub>n</sub> )} or ... {Bind(A <sub>n</sub> ), Unbind(A <sub>1</sub> ), Unbind(A <sub>2</sub> ) ... Unbind(A <sub>n-1</sub> )}
	False	1+(n <sup>2</sup> -n)/2	{Unbind(A <sub>1</sub> ), Unbind(A <sub>2</sub> ) ... Unbind(A <sub>n</sub> )} or Any two of the features in the group are bound
Or 	True	n	{Bind(A <sub>1</sub> )} or {Bind(A <sub>2</sub> )} or ... {Bind(A <sub>n</sub> )}
	False	1	{Unbind(A <sub>1</sub> ), Unbind(A <sub>2</sub> ) ... Unbind(A <sub>n</sub> )}

posite constraint to an SBC. For example, given a composite constraint “*All-Set(A,B) composite-excludes Alternative-Set(C,D)*”, methods are generated through combination of the states of the features in the two sets. The four derived methods are { $Unbind(A)$ }, { $Unbind(B)$ }, { $Unbind(C)$ ,  $Unbind(D)$ }, and { $Bind(C)$ ,  $Bind(D)$ }.

### 4.2 Construct LGB Method Graphs

In our approach, we divide a feature model into the CFM and the PCS, and provide priority-based operations through constructing LGB method graphs. To construct LGB method graphs for feature models’ tolerance, we have to extend and revise SkyBlue through: 1) redefining method conflicts; 2) specializing the execution process.

An LGB method graph is constructed under the following conditions: 1) a new constraint is added/deleted to the CFM; 2) the priority of a constraint in the CFM/PCS is changed. The pseudo codes are shown below.

*Add/delete a constraint in CFM*

```

ConstructCFM(Constraint SBC, Boolean isAdd){
  If(isAdd){
    ConstructLGB(Constraint SBC)
  }
  Else {
    UnenforcedCnsSet =
      collectUnenforcedConstraints();
    While(UnenforcedCnsSet != null){
      unenforcedCn = UnenforcedCnsSet.get();
      ConstructLGB(SBC);
    }
  }
}

```

### Changing a constraint's priority

```

ChangePriority(Constraint SBC, Priority p){
    oldPriority = SBC.priority;
    SBC.priority = p;
    If (oldPriority < p){
        If (SBC.selectedMethod == null)
            ConstructLGB(SBC);
    }
    Else If (oldPriority > p){
        If (SBC.selectedMethod != null)
            ConstructLGB(SBC);
    }
}

```

Constructing an LGB method graph involves enforcing the constraints in the constraint graph. To enforce a constraint, we select a method for it, change the methods of the constraints with the same or stronger priorities, or revoke one or more weaker constraints. This process is called constructing a *method vine* or *mvine*. When an mvine for the newly-added SBC is built, the SBC is successfully enforced.

Note that each time a constraint is successfully enforced (i.e. an mvine is constructed), one or more weaker constraints may be revoked. To construct an LGB method graph, these revoked constraints are added to the unenforced constraint set. Then our algorithm repeatedly tries to enforce all the constraints in the unenforced constraint set by constructing mvines for these constraints, until none of the constraints can be enforced. This process terminates because of the finite number of constraints. The pseudo code of constructing an LGB method is shown below.

### Construct an LGB method graph

```

ConstructLGB(Constraint SBC){
    //clean the unenforced constraint
    //set before enforce the newly-added SBC}
    clearUnenforcedCnSet();
    addToUnenforcedCnSet(SBC);
    While(UnenforcedCnSet != null){
        unenforcedCn = UnenforcedCnSet.get();
        buildMvine(unenforcedCn, unenforcedCnSet);
    }
}

```

SkyBlue uses a backtracking depth-first search to build mvines. The pseudo code of building a mvine is shown as follows:

### Build a Mvine for a unenforced constraint

```

buildMvine(Constraint root){
    While (root has methods){
        Method m = getMethod();
        If (!checkConflicts()){
            return true;
        }Else{
            Constraint cn = getConflictsConstraint();
            If (cn weaker than root){
                revokeConstring(cn);
                return true;
            }Else{
                If (buildMvine(cn))
                    return true;
            }
        }
    }
    return false; //start backtrack
}

```

To apply the SkyBlue to detect and tolerate inconsistencies in feature models, our algorithm redefines *method conflict* and revises the SkyBlue algorithm for building mvines. In SkyBlue, a conflict happens when a variable is determined by more than one method. In our approach, the variables

in the constraint graph can only be *bound* and *unbound*. Therefore, even if a variable is determined by more than one method, it may not cause a conflict (e.g. see variable *B* in Fig. 4). A conflict happens only when this variable is set to different value by different methods.

In SkyBlue, if an LGB method graph contains directed cycles, it is not possible to find an execution sort to satisfy all the variables in the LGB method graph. However, our approach can just execute all the methods to satisfy all the constraints, because all the methods set a variable to a fixed value.

## 5. PERFORMANCE

In this section, we investigate whether our approach can scale up to large feature models. To evaluate the scalability, we randomly generate feature models<sup>2</sup> and tolerate the inconsistencies in the generated feature models. We choose to use generated models because none of the large models are publicly available. The generated feature model contains a root feature. We can specify the number of the subtrees that are connected to the root feature, the height of the subtrees, the number of the child feature for each non-leaf feature in the subtrees, the number of the constraints. The percentage of the variability of features are: Mandatory (25%) and Optional (75%). The priorities of constraints are randomly set between 1 to 5.

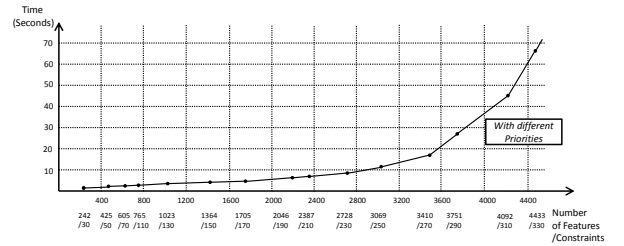


Figure 5: Experiments results for randomly generated feature models

The environment for our experiments is a Win 7 PC with a 2.66 GHz CPU, 2GB memory and the result is shown in Fig. 5. A mandatory feature or optional feature brings constraints with their parents, *m-requires* and *requires*, respectively. The constraints showed in the results are the constraints explicitly modeled into the feature models, they do not contain the simple constraints that are brought with the *Mandatory* and *Optional* feature.

In our approach, we check inconsistency and generate the PCS incrementally. For example, in the second case, 425 mandatory or optional features are added (each bring a constraint), and 50 constraints are explicitly modeled, we generate the PCS 475 times in total and cost 1.2s in all. The results indicate that our approach can scale up to large feature models.

## 6. RELATED WORK

Feature models are first proposed by Kang et al. [7] in the feature-oriented domain analysis (FODA) method. Since

<sup>2</sup>See <http://sei.pku.edu.cn/~wangbo07/> for our system and the feature model random generation algorithm.

then many researches focus on the detection of inconsistencies in feature models [3]. Maßen and Lichter [13] proposed a deficiency framework of feature model. They point out that inconsistency is one of the most severe deficiencies in feature models. Mannion et al. [8] was the first to use propositional formulas to find inconsistencies. Batory [2] proposed an approach to detecting deficiencies with SAT Solver. Benavides et al. [4] were the first to use constraint programming for analysis on feature models. Our previous work [16] focused on how to analyze feature models using BDD.

However, these approaches do not focus on how to find the unsatisfied constraints and tolerate inconsistencies in feature models. Balzer [1] pointed out the importance of tolerating inconsistencies, when the inconsistencies cannot be fixed. Trinidad et al. [12] focus on the explanation of inconsistencies in feature models, which helps find unsatisfied constraints. Nakajima et al. [9] propose some heuristics rules to find the unsatisfied core. However, these approaches do not provide explicit support to handle the tolerated inconsistencies and the scalability of these approaches is also not clear. Zowghi et al. [17] propose an approach to handling inconsistencies as a consequence of evolution changes performed on requirements specification, while our approach focuses on the inconsistencies in feature models.

## 7. CONCLUSION AND FUTURE WORK

In this paper, we adopt the constraint hierarchy theory and extend the constraint solver-SkyBlue to implement a system that can effectively tolerate inconsistencies in feature models. The feature model is divided into two parts, consistent feature model part and the pending constraint set part, through building LGB method graphs. Domain analysts can construct the feature model by working on the CFM while handling the tolerated inconsistencies that are expressed explicitly by the PCS. Three operations are defined and supported by our system, with the purpose of helping domain analysts handle the tolerated inconsistencies. Our future work will focus on investigating the applicability of our approach.

## 8. ACKNOWLEDGMENTS

The authors would like to thank Hiroshi Hosobe (NII, Japan) for introducing Delta/Skyblue to us. This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320701, the National High Technology Research and Development Program of China (863) under Grant No. 2009AA01Z139, the Natural Science Foundation of China under Grant No. 60703065, 60873059, and the National Institute of Informatics (Japan) Internship Program.

## 9. REFERENCES

- [1] R. Balzer. Tolerating inconsistency. In *ICSE*, pages 158–165, 1991.
- [2] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.
- [3] D. Benavides, S. Segura, and A. R.-R. Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 2010.
- [4] D. Benavides, P. Trinidad, and A. R. Cortés. Using constraint programming to reason on feature models. In *SEKE*, pages 677–682, 2005.
- [5] A. Borning, B. N. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.
- [6] K. Czarnecki, S. Helsen, and U. W. Eisencker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
- [7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU-SEI, November 1990.
- [8] M. Mannion. Using first-order logic for product line model validation. In *SPLC*, pages 176–187, 2002.
- [9] S. Nakajima. Semi-automated diagnosis of foda feature diagram. In *SAC '10*, pages 2191–2197, New York, NY, USA, 2010. ACM.
- [10] M. Sannella. The skyblue constraint solver and its applications. In *PPCP*, pages 258–268, 1993.
- [11] M. Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *ACM Symposium on User Interface Software and Technology*, pages 137–146, 1994.
- [12] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883 – 896, 2008. Agile Product Line Engineering.
- [13] T. von der Maßen and H. Lichter. Deficiencies in feature models. In *Workshop on Software Variability Management for Product Derivation, in Conjunction with SPLC*, 2004.
- [14] B. Wang, W. Zhang, H. Zhao, Z. Jin, and H. Mei. A use case based approach to feature models' construction. *IEEE International Conference on Requirements Engineering*, 0:121–130, 2009.
- [15] W. Zhang, H. Mei, and H. Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requir. Eng.*, 11(3):205–220, 2006.
- [16] W. Zhang, H. Yan, H. Zhao, and Z. Jin. A bdd-based approach to verifying clone-enabled feature models' constraints and customization. In *ICSR*, pages 186–199. Springer, 2008.
- [17] D. Zowghi and R. Offen. A logical framework for modeling and reasoning about the evolution of requirements. In *RE*, page 247. IEEE Computer Society, 1997.



# New Strategies to Resolve Inconsistencies between Models of Decoupled Tools

Anne-Thérèse Körtgen  
Department of Software Engineering  
RWTH Aachen University  
Aachen, Germany  
koertgen@se-rwth.de

## ABSTRACT

Maintaining consistency between models in development processes is a challenging task. Though in software engineering many case tools exist which allow the editing of models and include code generators for reverse or round trip engineering, the source code is edited with specialized development environments. The same holds for other models, such as requirements specifications or architectures. The *editing tools* are often *decoupled* from each other. Additionally, modifications of the models are performed simultaneously.

In this paper, we introduce strategies to resolve inconsistencies by so-called *repair actions*. The novelty of the strategies is that they specify different concurrent ways of coming to a consistent state, not knowing anything about the modifications a user had applied. Concrete repair actions are derived at runtime based on the consistency rules and the inconsistent parts of the models only. The maintenance tool can maintain additional models' consistency without developing specialized repair actions. One new strategy proposed in this paper in contrast to other approaches is to *heal* the violated consistency rule or try a similar one.

## Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Corrections, Version control*; D.2.12 [Software Engineering]: Interoperability—*Data mapping, Distributed objects*; K.6.3 [Management of Computing and Information Systems]: Software Management—*Software Development*

## Keywords

Inconsistency Management, Model Synchronization, Incremental Transformation, Model-driven Development

## 1. INTRODUCTION

Development processes in engineering disciplines tend to be highly complex. As a result, the product to be developed needs to be modeled from multiple interdependent perspectives and on different levels of abstraction. The results of development activities like requirements definitions, software architectures, or implementations in software engineering are stored in *models*. Different models may overlap, so when they are edited independently, they may impose constraints on the system to be developed which are not satisfiable any more. Models are then said to be *inconsistent* to each other [29]. The task is to *maintain* the *inter-model consistency* after editing.

In literature consistency maintenance of models are also called *inconsistency management* [8] or *model synchronization* [11] and involves the following tasks [23]: (i) *consistency* of dependent models must be *checked*, (ii) *inconsistencies* must be *categorized*, and (iii) inconsistencies must be *handled*, i.e. resolved, ignored, or identified only. Reestablishing consistency is also called *reconciliation* [28].

Usually, valid statements within model overlappings are formalized by *consistency rules* [23]. For example, a consistency rule declares that for each class definition in a UML class diagram a corresponding class definition must exist in the source code. If a class exists in the diagram but not in the source code, then the consistency rule is violated. With the specification of consistency rules *dependencies* between two or more models are therefore also specified. Thus, when one model is changed and a consistency rule is violated, then consistency can be reestablished by changing dependent models analogously (propagation).

Many approaches face the problem of inconsistencies by offering one *integrated case tool* which performs the tasks of inconsistency management by direct propagation of changes to dependent models such as e.g., EclipseUML [25] or Rational Software Delivery Platform [19]. But often there are several reasons why changes cannot be propagated directly to depending models: (i) when different *specialized heterogeneous tools* for a task are used or (ii) when developers are distributed and *edit* different copies *in parallel*, so that an integrated tool cannot observe incremental changes.

*Version control systems* cannot be used after parallel editing of the models when dealing with graphical models [9], especially if they are stored within binary data: If they are stored as text files, changes cause often an overall rearrangement

of the text lines. *Model transformation approaches* [32, 4] transform a source model into a target model. Mostly, these transformations work batch-wise, i.e., a target model is generated completely from a source model, thus, changes on the target model get overridden.

Therefore, consistency maintenance should resolve an inconsistency by propagation of changes on one model to depending models *incrementally* and *bidirectionally*. Incrementality means that only local changes are made in a model to handle one inconsistency so that other changes on the model which were done in the meantime are not overwritten. Also inconsistency resolution should be made only *on request*, meaning that *inconsistencies must be tolerated* [23]. For example, the user’s changes on a model create inconsistencies but the editing process should not be interrupted, thus, resolving has to wait until the end of the editing process. As an additional requirement, *interaction* with the user is required when resolution is ambiguous.

*Triple graph grammars* (TGG) [27] were developed for handling consistency maintenance problems of two models which are modeled as graphs. The idea is to store fine-grained  $n : m$ -relationships in a *correspondence graph* between a *source* and a *target graph* and to specify analogous operations on the three models in so-called *triple rules*. By applying triple rules, we may modify coupled models synchronously, taking their mutual relationships into account. But also the operations on one model can be performed independently and based on the triple rule specification one can maintain consistency by applying the corresponding operations on the other model. TGG approaches basically work incrementally and bidirectionally.

A problem is that triple rules need to be monotone, i.e. they only produce new nodes and edges and are not allowed to delete any. Thus, only inconsistencies caused by addition of new nodes in one of the models can be resolved by transforming them into the dependent model (by applying remaining operations of a triple rule). We would like to call inconsistencies of that kind *category 1 inconsistencies*. But modifications on one of the models may affect nodes which are related to nodes of the other model. It can then happen that the relationship defined by the triple rule which was applied before is not present any more. We would like to call inconsistencies of that kind *category 2 inconsistencies*.

In this paper, we introduce strategies to resolve category 2 inconsistencies by so-called *repair actions*. The novelty of the strategies is that they specify different concurrent ways of coming to a consistent state, not knowing anything about the modifications a user had applied. Concrete repair actions are derived at runtime based on the consistency rules and the inconsistent parts of the models only. The maintenance tool can maintain additional models’ consistency without developing specialized repair actions. One new strategy proposed in this paper in contrast to other approaches is to *heal* the violated consistency rule or try a similar one.

This paper is structured as follows: Section 2 presents a scenario where the maintenance tool is applied. Section 3 introduces some basic concepts of the underlying data struc-

tures, the modeling of consistency rules, and the definition of consistency in this setting. In Section 4, the strategies for finding and resolving category 2 inconsistencies are presented. Section 5 describes shortly the implementation of the maintenance tool and the resolving strategies. Related work is discussed in Section 6 and, finally, Section 7 concludes the paper.

## 2. SCENARIO

While the underlying concepts of the maintenance tools are fairly general and domain-independent, we focus on software engineering and introduce a sample scenario where UML class diagrams and source code are maintained consistent. For designing the structure of a system, UML class diagrams are well-suited as they abstract from the implementation details of the system and they are part of static structure diagrams in the UML. Heterogeneous tools may be used for creating UML class diagrams and source code, respectively. In the following, we assume that the UML class diagrams are maintained by Eclipse [31] and source code is edited in Visual Studio [20]. We chose to use these applications for our case study because they were successfully used in many software projects as modeling tool as well as integrated software development environment and they are very widely known.

Figure 1 illustrates how the tool assists in maintaining consistency when changes are made simultaneously. Three different versions of the sample UML class diagram and the corresponding Visual Studio (VS) object model are shown above and below the dashed line, respectively. The tool maintains a data structure which contains links for connecting the two models. These links are represented by ellipses which are located on the dashed line. Dotted lines are used to indicate the objects participating in a link<sup>1</sup>. The connections of a link to objects in both models base on template specifications for corresponding object patterns, e.g. that UML classes correspond to code classes, UML associations correspond to attributes in the code owned by corresponding classes, or UML methods correspond to code methods also both owned by corresponding classes. Furthermore, arrows located on the right indicate the directions of change propagations (structural and attribute changes, respectively). The figure illustrates a re-design process consisting of three steps described in the following.

Initially, the VS object model and the corresponding UML class diagram are consistent to each other, thus, links already exist between the objects (see left version in the figure). They each contain the classes `Control` and `DataAccessObject` and the methods `getDbItem` and `getFileItem` which are elements of latter class. The association `data` is represented in the object model as attribute referencing the corresponding class object as type.

Secondly, both models are edited simultaneously (see middle version in the figure). Changes are marked with `new` or `*` in the figure for newly added objects or changed objects, respectively. To the VS object model only the helping function `helper` is added. To the class diagram two further

<sup>1</sup>Please note that this is a simplified notation. Some details of the link data structure introduced later are omitted and also not every link is shown.

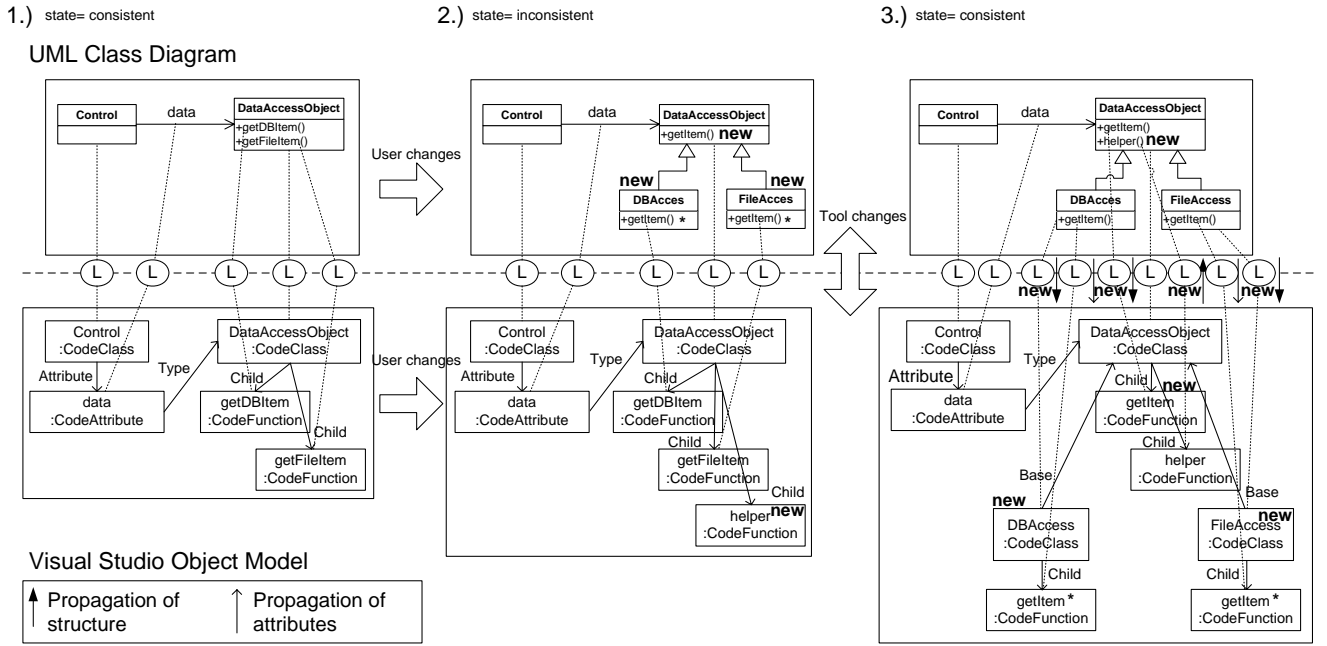


Figure 1: Recovering consistency between design and implementation

classes DBAccess and FileAccess are added and the methods of their super class are distributed among these classes. Additionally, the method names are generalized to `getItem`. The models are now inconsistent but the links still exist. We see that inconsistencies caused by newly added objects are category 1 inconsistencies while those caused by changed objects are category 2 inconsistencies.

Finally, the maintenance tool is used to synchronize the parallel work which results in the right versions of the models in the figure. The newly added objects are added to the respective other model. The category 2 inconsistencies within the methods correspondence relation are the non-valid attribute condition, i.e. the method names are not equal, and the owning classes of the UML and code methods do not correspond to each other, respectively. To resolve the inconsistencies the name change in the UML model is propagated to the code model and the shift of the methods to different classes in the UML model results in rearrangement of the references between the class and method objects. What we see here is, that the basic template specification of corresponding methods are still valid. Please note that there might be compile errors in the source code after the synchronization.

The example presented above demonstrates the functionality of the maintenance tool, but it does not show how this is achieved. In the next sections, the underlying conceptual approach is presented. The implementation is introduced in Section 5.

### 3. CONCEPTS OF THE APPROACH

In our approach, we use typed, directed graphs to represent the models between which relationships exist. Graphs consisting of nodes and edges are well suited for representing complex data with manifold relationships in a natural way.

Also, inter-model relationships can be expressed easily with nodes and edges referencing related nodes of the graphs. As another advantage we can use the theory of *graph transformations* to define modifications on the graphs. In the following, the data structures, construction of consistency rules, and the notion of consistency in this setting are presented.

#### 3.1 Models as Triple Graphs

For using TGG in complex scenarios where two dependent models are edited by different tools we create graphs from these models (via tool wrappers) and denote one as *source* and the other as *target model*. Operations on the models performed by the respective tools are represented as graph transformations. The data structure storing links between inter-dependent models is called *integration document* which is also modeled as graph. The framework creates an integrated graph consisting of the three graphs to simplify specification of the triple rules and their application.

In the following, we want to use UML object diagrams to represent typed graphs and to model triple rules as well. The graph schemes, therefore, are modeled as UML class diagrams. To be able to model integration scenarios, we provide an extension of the UML meta-model [3] by stereotypes. To illustrate the graph modeling, we use the example in Figure 2 which shows the graph representation of the UML class diagram, the VS Object model, and the integration document from the left situation of Figure 1 with only one method object.

To model source and target graphs we use the stereotypes `Increment` for nodes and `Incr2Incr` for edges. Source and target graphs have an underlying graph scheme equivalent to the respective models's meta-models. This means the graph schemes of UML class diagrams and source code which we

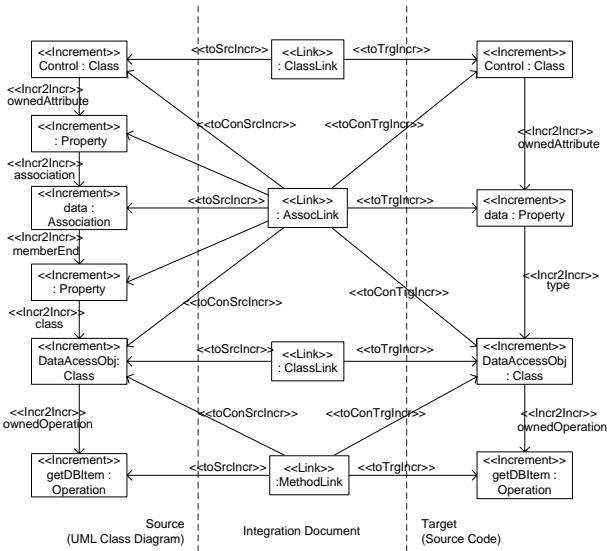


Figure 2: Triple graph as UML object diagram

want to maintain consistent do not differ much from the meta-model of UML class diagrams specified by the OMG in [24]. This makes sense, since they both reflect the structure of the system. In general, the graph schemes of the models are different. The tool wrappers realize the concrete type mapping, that is for example the concept of a Class instance in the graph representation of the UML model is mapped to an *Ecore Class* and that of the VS Object Model is mapped to an instance of the type *CodeClass*. In Figure 2, we see that source and target graph contain objects of concepts of the UML meta-model.

The integration document objects use the stereotype *Link* to mark the links. Links refer to increments in source or target model via *UMLLinks* (instances of associations). The increments have to be distinguished according to the graphs they are contained in, either *source* or *target* graph. Therefore, stereotypes for *UMLLinks* from links to increments appear twice, one for increments of the source and one for increments of the target graph. This is indicated by suffixes *SrcIncr* and *TrgIncr* of the stereotypes' names. Dependent on whether the increments play the role of a normal or context increment, they are connected to a link via *UMLLinks* with stereotype *toSrcIncr*, *toTrgIncr*, *toConSrcIncr*, or *toConTrgIncr* respectively. Increments may play the role of so called *context increments* for a link when they define some kind of existent-dependency for a link, i.e. a link relates increments only if the context increments are connected to these increments within the models.

The integration document also has an underlying graph scheme which models the link types and how they are related to classes of source and target graph schemes. In Figure 2, we see that links of type *ClassLink* and *AssocLink* are represented which are used to relate classes or associations in both models. In this example, the association (source) is mapped to a property (target).

### 3.2 Consistency Rules as Triple Rules

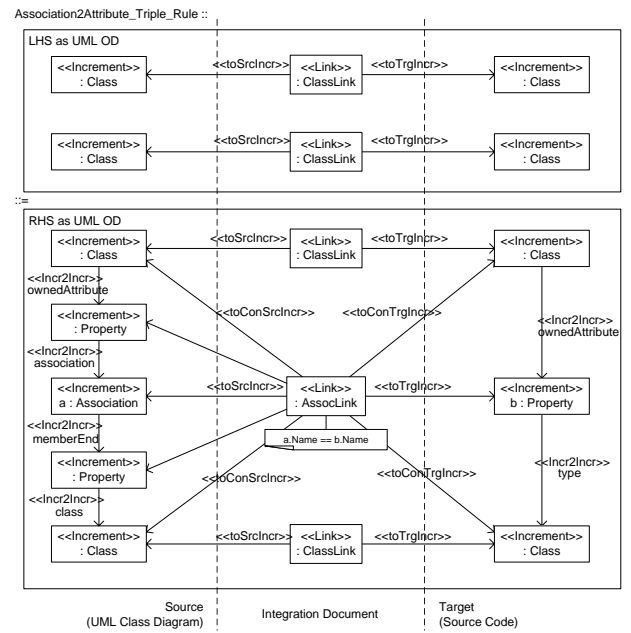


Figure 3: Triple rule (synchronous) creating an association *a* in a UML class diagram (left) and a corresponding attribute *b* with the same name in a source code (right)

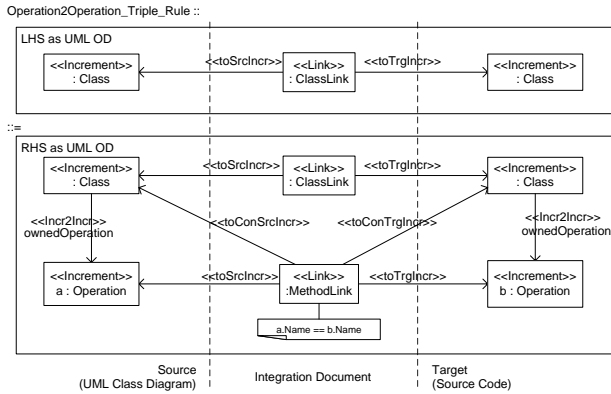
As mentioned, the connections of a link to objects in both graphs base on template specifications for corresponding graph patterns. These are defined in triple rules which are also modeled as object diagrams. To give an example, the triple rule which defines the connection pattern of the association link from Figure 2 is shown in Figure 3.

The triple rule defines in the first place how to relate two classes within a UML class diagram with an association (source) and to simultaneously add in the source code (target) an attribute to the class corresponding to the association's source class whose type is of the class corresponding to association's target class.

The left-hand-side (LHS) of the rule is shown above and specifies that the two classes in each model already exist and are related to each other via links in the integration document. The right-hand-side (RHS) shown below specifies the situation after the rule has been applied meaning that the association in the source graph, the attribute in the target graph, the link in the integration document mapping both onto each other, and the edges embedding these increments in the existing graphs are created. This triple rule also specifies an *attribute equation* shown in the comment attached to the new link. It states that the increments *a* and *b* have the same names. The RHS is the template which declaratively specifies the connection pattern for a consistent link.

A rule can also specify restrictions on attributes of the increments, for example the name of a class *aClass* could be restricted by *aClass == 'User'*. If the restrictions or equations appear on the LHS they restrict graph pattern matching allowing only those increments to be matched which fulfill

these constraints. If they appear on the RHS, the constraints have to be fulfilled in any case. When there is no value specified within the rule such as in the triple rule shown above, a dialog is prompted to enter a value.



**Figure 4: Triple rule (synchronous) creating an operation a in a UML class diagram (left) and a corresponding operation b with the same name in a source code (right)**

As a further example, Figure 4 shows the triple rule which is applied in the scenario for creating the methods `getDBItem` and `getFileItem` in source and target graph and the links between them.

Please note: if asynchronous rules are derived from triple rules which have equations, attribute values are transferred into the other model. For example, the forward transformation of the triple rule shown above sets the attribute's name to the association name (`b.name := a.name`), vice versa for the backward transformation (`a.name := b.name`).

### 3.3 Definition of Consistency

TGG base on Pair Graph Grammars (PGG) [26]. Pratt defined consistency of two graphs based on pair rules: "Two graphs are consistent iff both graphs can be created by synchronously applying corresponding graph grammar rules on corresponding nodes."

Of course in real life scenarios a rule modeler has to model the triple rules for two models edited by (as a general rule) commercial tools and so it cannot be assumed that he specifies the complete TGG covering each possible change a tool can make on one of its models and relates it to changes the other tool should make so that the models remain consistent. One can assume that the tools create valid models according to the models' meta-models so that inner-model consistency must not be checked.

Although in practice we do not have a complete TGG we can keep the definition of consistency of Pratt. We only regard those structures in the models which *can* be created by one of the triple rules. Other increments are ignored; they also do not make the models inconsistent as there is no triple rule that would define corresponding increments in the other model.

## 4. INCONSISTENCY MANAGEMENT

Our tool performs the three tasks mentioned in [23]. In this section, we only present the check and resolving strategies for category 2 inconsistencies. A detailed description of the consistency check and resolution of category 1 inconsistencies can be found in [2].

The main issues we are dealing with are that we cannot record the changes an external tool performs on a model and also that we cannot assume that a rule modeler specified for each operation on a model an equivalent operation on a dependent model. So, instead of parsing the graph and finding out the operations which were performed on a model and to perform equivalent operations on a dependent model, we decided to take only the already available triple rules into account and those parts in the model which make a triple rule inconsistent.

It is so, that a triple rule specifies declaratively in its RHS the correspondences of two patterns in source and target graph. Thus, if the rule is applied, there exist instances of these patterns (subgraphs) in source and target graph and a link exists in the integration document referencing the nodes of these subgraphs. An inconsistency according to a triple rule is, therefore, that the patterns do not match anymore or that attribute conditions are violated. Thus, resolution means to maintain the graphs so that a link referring to nodes in source and target graphs is valid according to an RHS of any triple rule, not necessarily the one which was formerly applied.

It is important to note that a repair action is allowed to change only nodes and edges that were originally created by the applied rule. That does not hold for context increments. Therefore, a repair action never causes new inconsistencies for links referenced as context. But of course, other links having a repaired link in their contexts could be damaged.

### 4.1 Inconsistency Check (Category 2)

For existing links we check if there were modifications on their referenced increments by doing pattern matching using the RHS of the triple rules applied before. Each link in the integration document has an associated state. When a link has been newly created by applying a rule, its state is initially set to *consistent*. In the analysis, for each link it is checked whether increments originally referenced by the link are still present in source and target graphs. If the RHS of the triple rule does not match any more due to modifications of source or target graphs, the state of the link is changed to *damaged*. For handling the inconsistency all reasons why the link got damaged are collected during analysis. Here, all possibilities for a link to be damaged are listed:

**Increments deleted** Increments of the source or target graph which take part in a link have been deleted, resulting in dangling references of the link. As an example, an attribute of a class could have been deleted.

**Attribute values changed** Attribute *values* of source or target increments of a link have been *changed* so that attribute conditions do not hold any more. This would be the case if the attribute of a class in the source code has been renamed resulting in an inconsistency of the Associa-

tion2Attribute-rule because the name of the corresponding association in the UML class diagram of that attribute and the attribute's name are no longer equal.

**Edges deleted** *Edges* of the source or target graph being involved in a link have been *deleted*, so that the patterns on both sides are no longer valid. For example, an attribute of a class in the source code has got another type so that the edge typeRef from the attribute to the old type is deleted.

**Context damaged** If a link gets damaged all links referring to it as context get damaged, too. For example, a link L1 mapping an association onto an attribute refers to two links as context links, i.e. those links mapping two classes in each graph onto each other. If one of those links gets damaged, L1 is damaged, too.

## 4.2 Resolution of Inconsistencies (Category 2)

We now present what can be done in general if damaged links have been found and list in the following sections graph transformation strategies. We call these strategies *repair types* because we create concrete instances from these strategies, *repair actions*, to repair a concrete damaged link with specific damages. Applied to a damaged link, a repair action brings the link back to a consistent state which means that it resolves the inconsistency.

### 4.2.1 Delete all increments involved

The most primitive procedure is to *delete the link and all the increments* on both sides, which are involved in the integration situation. It is obvious that this is suitable only in situations where all increments of a link have been deleted by the user in one of the graphs or where one of the main increments has been deleted. If some increments of a link in one of the graphs are deleted and others still exist, the deletion could have been part of a restructuring activity, thus, deleting all increments is not the reaction the user expected. Still, it is a valid repair strategy, but one which should never be executed without prompting the user before.

### 4.2.2 Remove the link and integrate again

The next simple possibility is to *only delete the damaged link* and to leave the increments on both sides unchanged and to make them available for other transformation rules. Most of the time this also does not lead to the desired behavior of the tool. The modifications resulting in the damaged link were probably done on purpose. The link, although damaged, may contain valuable information to be used, most importantly, to determine which parts of the graphs may be affected by the modification which damaged the link. For example, in the scenario the Operation2Operation rule from Figure 4 had been applied and the user rerouted the method getItem from the class DataAccessObject to another class DBAccess, then the rule did not match anymore. The edge ownedOperation from the class DataAccessObject is missing. But instead of deleting the link and all references the desired repair action is to reroute the ownedOperation reference of the corresponding method in the source code from the class DataAccessObject to the class DBAccess.

### 4.2.3 Undo changes of the user

Another option is to restore consistency by *removing the cause for the inconsistency*. For instance, missing increments or edges may be created. This option is desirable only in those cases where the operation causing the damage was carried out accidentally, because it would be undone. For attribute values, the attribute conditions of the triple rule can be used to propagate the change.

### 4.2.4 Define new rule

This is a trivial solution to handle a damage as for the new situation a rule is defined (induced) and attached to the link.

### 4.2.5 Conserve the applied rule

As the alternatives to repair inconsistencies presented so far are not very useful from the practical point of view, more specific repair types [16] based on knowledge about the original rules which created the links have to be considered. The main priority when repairing links is to *conserve the rule* that has been originally *applied* to create the link or to find a similar one doing only small adaptations (see below).

Conserving the applied rule in general means to create a situation where the damaged link refers to increments in source and target graphs so that all required conditions of that rule including graph pattern matching are fulfilled. As there can be multiple reasons for a link to get damaged each single damage has to be *healed* when trying to conserve the applied rule. We present in the following sub-strategies to heal a single occurrence of a damage of the same type, e.g. if an increment is missing, then a strategy/action is to replace it. If multiple increments are missing, then there are actions for each increment to replace it. Of course all occurrences of all damage types must be healed which results in building actions of the same and different sub-strategies.

A repair can be done by changing or not changing source and target graphs. Not changing means that only references within the integration document are adapted assuming that the user established a consistent state by himself. So, conserving the applied rule strategy exists in two variants. We explain what has to be addressed when a rule should be conserved with and without doing any changes on the graphs.

#### Conserve the applied rule (changing):

- **Create increment** If an increment is deleted it can be recreated, thus, this (single) damage is removed, resulting in an *undo*-like operation of the user's changes. Note, this is not a real undo such as in the **Undo changes repair type**, because attribute values of the recreated increment are not reestablished as before the deletion of the increment.
- **Create edge** If an edge is deleted it can be recreated as above. This also is not a real undo, because an edge which was deleted due to the deletion of an incident increment which is not part of the rule pattern is not recreated by this repair.
- **Propagate attribute value** If an attribute value changed so that the condition of an attribute equation of the applied rule does not hold, this repair type changes

attribute values so that the violated attribute equations hold again. For example, in the scenario of Figure 1 the rule which maps two methods onto each other is applied two times. It states that methods *a* from the source and *b* from the target graph referred by a link must have the same names, here `getDBItem` and `getFileItem`. The user afterwards changed both methods' names in the UML model having the role of method *a* to `getItem`, there are two possible actions: (i) `b.name := a.name` or (ii) `a.name := b.name`. The first would correspond to a propagation but the system is not able to determine which value (that of *a* or *b*) had changed by the user, so it suggests both alternatives.

- **Change attribute value according to restriction** If an attribute value is changed so that the condition of an attribute restriction is violated, this repair changes the attribute value so that the violated attribute restriction holds again. For example, if the restriction is `aClass.name == 'User'` and `aClass.name` differs from this value, then the action `aClass.name := 'User'` is proposed. If the restriction is `a.card < 10` and `a.card` is greater than 10, then the action `a.card := 10 - 0,1` is proposed, where it must be said that the percentage which is additionally subtracted can be configured by the user. Note that, for a greater than (>) relation, a percentage is added up.
- **Include alternative context** If increments and edges of the *context* of a damaged link are missing, this repair type adapts references in the integration document from the damaged link to existing increments and edges in source or target graph (in one step) to make the context valid again for the damaged link. Nevertheless, the repair type is allowed to reassign edges from the non-context increments to the new context so that finally changes in the graphs are made. This repair is applied twice in the scenario of Figure 1 where the method links first refer to the same context increment `DataAccessObject` in the source code and after the repair to the alternative context increments `DBAccess` and `FileAccess`, respectively.

**Conserve the applied rule (not changing):** Not doing any changes on the graphs but to heal each single damage is only possible for deleted increments and edges as they can be replaced with already existing increments and edges by adapting the references in the integration document. Violated attribute conditions cannot be reestablished as this would mean to change at least one value in the graphs.

- **Include alternative increment** A deleted increment can be replaced by an *alternative increment* which exists in the respective graph if it is not used by another link. This increment must fulfill all conditions stated in the triple rule, as for example attribute restrictions or connections to other increments via edges.
- **Include alternative edge** A deleted edge is only a cause for a damage if its two incident increments still exist. If not, then this damage is handled by alternative increment (see above). To find an alternative edge for the deleted one means that either one of the

incident increments or both are replaced with alternatives which exist in the graphs if they are not used by another link. As for the alternative increment repair type (see above) these increments have to fulfill all requirements of the rule.

- **Include alternative context** This repair type is equal to the changing version (see above) not being allowed to do any change on the graphs. Thus, a valid context is included only if it is already connected as required to the non-context part of the link.

#### 4.2.6 Apply a similar rule

The following repair types do not conserve the originally applied rule but *substitute* it with a *similar* one.

**Apply a subset rule (changing):** A triple rule is a subset rule of another triple rule if its RHS is a subset of the RHS of the other rule. If an applied rule is inconsistent, one can try to look for a subset rule which is still consistent in the given situation. The applicability of a subset rule is likely as a subset rule has less conditions which may not have been affected by the modifications of the user. If there exists such a subset rule which is not damaged, then increments which not appear in the subset rule but by the damaged rule are deleted. The situation after the repair is as if the subset rule had been applied.

**Apply another rule from the same rule group (changing):** Two rules are in the same rule group if they are ambiguous for a pattern in a graph which means the pattern can be represented in a dependent graph in two ways. If an applied rule is damaged it is likely to look for an *alternative rule* of the same rule group which is applicable and only do little adaptations to the graph; in case that some preconditions for applicability of the rule are not fulfilled, their validity can be enforced with small changes on the graph. For example, required nodes and edges for the application of a different rule can be created and nodes and edges of the formerly applied rule which are not used by the alternative rule can be deleted.

**Apply another rule from the same rule group (not changing):** The same is possible without doing any changes on the graphs.

**Apply another rule keeping the main increments:** Like alternative rule repair, this repair action searches for another possible rule which can be applied. But instead of taking the whole pattern into account, it just searches for source and target main increments, as a minimal requirement for an alternative rule application if they still exist.

## 5. IMPLEMENTATION

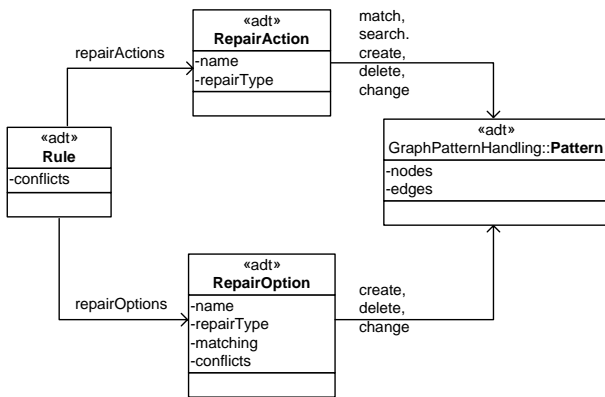
The strategies are implemented in a framework which was built within the IMPROVE project (1997-2009) [21] at RWTH Aachen University to rapidly construct consistency maintenance tools [17] for specific models and editing tools.

The framework works tool- and model-independent on a generic graph-based data structure; the model editing tools are plugged in the framework by using wrappers which provide the required graph views on the models to the frame-

work. Triple rules also base on the generic data structure and have to be defined for each pair of models to maintain consistently. Parameterized by the rules the framework is able to check, to categorize, and to maintain the consistency incrementally, bidirectionally, and with the work of [2, 1] they operate interactively.

For concrete models specific repair actions (i.e. graph transformations) for a damaged link are constructed after the damage check at runtime. The repair actions represent alternatives to make the link consistent again. They can be selected and prioritized so that the framework can be employed for various models and various phases of the development process with different requirements on resolution. We focus on the implementation of repair actions here, how they are built and executed.

A repair action is a concrete instance of a repair strategy for a category 2 inconsistency of a damaged link realized as graph transformation with a LHS and a RHS. The LHS and RHS of a repair action are modeled with `Pattern` objects which can be then interpreted at runtime by methods for pattern search and transformation by a graph transformation engine.



**Figure 5: Data structure to store repair actions and options in the integration document**

Figure 5 presents the data structure to store repair actions in the integration document. We need store this information there because of the interactivity of the tool. A user can select among multiple options and can stop the tool during consistency maintenance. The class `Rule` models the application of a triple rule and is connected to the respective link it has created. If a rule pattern no longer matches, a set of `RepairAction` and `RepairOption` objects are created for that rule. A `RepairAction` contains multiple `Pattern` objects: one is searched (`search`), one models the remaining matching subgraph (`match`), one defines the subgraph which is going to create (`create`), one models the pattern of the subgraph to be deleted (`delete`), and one specifies the pattern of the subgraph to be changed (`change`). We can say that the `search`, `match`, `delete`, and `change` patterns form the LHS and the `create` pattern forms the RHS. When repair actions are created pattern nodes and edges must be added to the respective `Pattern` objects.

```

1 List<RepairAction> AlternativeNode(List<
    RepairAction> ras, Pattern
    rsPattern, Pattern dmgPattern) {
2 List<RepairAction> retRas = ras;
3 foreach (RepairAction ra in ras)
4     Pattern search = ra.GetSearchPattern();
5     Pattern create = ra.GetCreatePattern();
6     foreach (PatternNode patNode in
    rsPattern.GetNodes())
7         if (dmgPattern.ContainsKey(patNode) &&
    !rsPattern.IsContext(patNode)) {
8             //search alt. nodes
9             search.AddNode(patNode);
10            //Edges of alt. nodes
11            foreach (PatternEdge edge in
    rsPattern.GetEdges(patNode))
12                if (!IntDocEdgeTypes.ContainsKey(edge
    .type))
13                    //search in model
14                    search.AddEdge(edge);
15            else
16                //create in Int.Doc.
17                create.AddEdge(edge);
18        }
19 return retRas;
20 }
  
```

**Figure 6: Creation of the repair action for searching for alternative nodes (simplified, in C#)**

When repair actions are constructed it is not known if they really are applicable, i.e. if the search pattern has a match in the graph. If a match is found, then a `RepairOption` object is created which denotes a really applicable repair action. All options are offered to the user. For each match of the search pattern found by the graph pattern search engine, the search pattern is enhanced with concrete matching node and edges ids of the graph and is now part of one `RepairOption` as matching. The repair option only references `create`, `delete`, and `change` patterns of the corresponding repair action.

To construct a repair action for conserving the originally applied rule, the patterns are extended step-wise according to the sub strategies. For example, the enhancement to search for an alternative context bases on the patterns which are enhanced with pattern nodes and edges to search for alternative nodes. Repair actions without sub strategies can be constructed independently in one step and retrieve the pattern of the rule which is still intact, the *remainder* pattern.

To demonstrate how repair actions are created we present as examples the implementation of the alternative nodes and context sub strategies. The methods describe how the different pattern objects are enhanced to match alternative nodes and an alternative context and replace the missing nodes and damaged context of a link.

Figure 6 shows the method which retrieves as input parameters a list<sup>2</sup> of already created repair actions `ras`, the pattern

<sup>2</sup>Please note that creation of alternative edge repair is done



of the rule `rsPattern`, and a damage pattern `dmgPattern`. The damage pattern contains the missing nodes and edges as well as those nodes which violate attribute conditions or restrictions. In this method, all repair actions of `ras` are extended. Alternative nodes and edges for the missing nodes and their incident edges are searched, i.e. they are added to the search pattern (lines 8 to 14). When the current repair action is applied all edges from the integration document to the alternative nodes are created, i.e. those edges are added to the create pattern (line 17).

Next, the repair actions are enhanced for searching for an alternative context. The method shown in Figure 7 is the changing (c) version and retrieves as input parameters the repair actions `ras` of the previous step. In this method, each repair action is doubled (line 7). One version (`ra`) searches for an alternative context group of a missing context node only in the graph of the missing node with pattern `search`. Nodes of the context are only added to the pattern, if the graph role is equal to the graph role of the missing node (lines 21 and 22). The other version (`ra2`) searches for an alternative context group in all three graphs with pattern `search2`. Nodes of the context are all added to this pattern (line 20). The edges of the alternative context are searched in `ra` only if they belong to the same graph as the missing context (line 27), `ra2` searches all edges between the context nodes (line 25). Edges from the context nodes which points to other nodes in the same graph as the missing node are searched in both versions (lines 28 to 30). Those in the other graphs are created (line 32) and edges to the former context are deleted (line 33) but only for `ra2`. The repair action `ra` only reassigns edges within the integration document (lines 34 to 36).

To give an example, we present in Figure 8 the repair action `ra2` which reassigns the operation `getDBItem` to the other class `DBAccess` which plays the role of the alternative context here. The search (LHS) pattern consists of the remaining graph with the current node ids. The nodes `DataAccessObj` in source and target graph as well as the `ClassLink` node between them are the context group which has to be replaced as the `ownedOperation` edge between `DataAccessObj` and `getDBItem` in the source graph is missing. The search pattern is extended by an additional context group, only edges between that nodes are added and the `ownedOperation` edge in the source graph where the edge is missing. The nodes and edges of the gluing graph  $K$  ( $K = \text{LHS} \cap \text{RHS}$ ) of the LHS and the RHS are not changed. The create pattern is the pattern  $\text{RHS} \setminus K$  and the delete pattern is  $\text{LHS} \setminus K$  and in this example the edges in the integration document and the target graph.

Generating repair actions at runtime is suitable because they cannot be all modeled beforehand foreseeing all possible changes a user can make. But this approach also implies performance problems when at runtime a set of repair actions is generated, most of them not being applicable as required, e.g., required nodes and edges of the search pattern are not present in the graph.

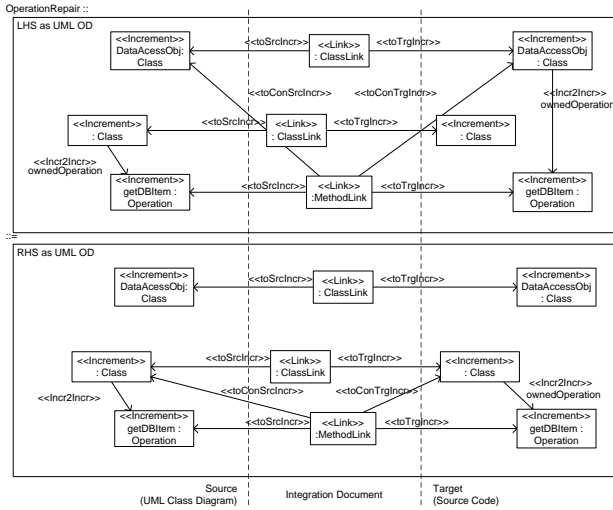
Therefore, we optimized the process by introducing *phases* before and that multiple repair actions are generated, i.e.  $3^e$  where  $e$  is the number of missing edges.

```

1 List<RepairAction> AlternativeContext_c(
    List<RepairAction> ras, Pattern
    rsPattern, Pattern dmgPattern) {
2 List<RepairAction> retRas = ras;
3 foreach (RepairAction ra in ras) {
4     Pattern search = ra.GetSearchPattern();
5     Pattern create = ra.GetCreatePattern();
6     Pattern delete = ra.GetDeletePattern();
7     RepairAction ra2 = new RepairAction();
8     retRas.Add(ra2);
9     Pattern search2 = new Pattern(search);
10    Pattern create2 = new Pattern(create);
11    Pattern delete2 = new Pattern(delete);
12    ra2.SetSearchPattern(search2);
13    ra2.SetCreatePattern(create2);
14    ra2.SetDeletePattern(delete2);
15
16    foreach (PatternNode patNode in
        rsPattern.GetNodes())
17        if (dmgPattern.ContainsKey(patNode) &&
            rsPattern.IsContext(patNode)) {
18            Pattern ctxtGroup = rsPattern.
                GetContextGroup(patNode);
19            foreach (PatternNode ctxtNode in
                ctxtGroup) {
20                search2.Add(ctxtNode);
21                if (ctxtNode.graphRole.Equals(patNode
                    .graphRole))
22                    search.Add(ctxtNode);
23                foreach (PatternEdge edge in
                    rsPattern.GetEdges(ctxtNode))
24                    if (ctxtGroup.ContainsKey(edge.
                        GetSourceNode()) && ctxtGroup.
                            ContainsKey(edge.
                                GetTargetNode())) {
25                        search2.AddEdge(edge);
26                        if (edge.graphRole.Equals(patNode.
                            graphRole))
27                            search.AddEdge(edge); }
28                    else if (edge.graphRole.Equals(
                        patNode.graphRole)) {
29                        search.AddEdge(edge);
30                        search2.AddEdge(edge); }
31                    else {
32                        create2.AddEdge(edge);
33                        delete2.AddEdge(edge);
34                        if (IntDocEdgeTypes.ContainsKey(
                            edge.type))
35                            create.AddEdge(edge);
36                            delete.AddEdge(edge);}}}}
37    return retRas;
38 }

```

Figure 7: Creation of the repair action for searching for an alternative context (simplified, in C#)



**Figure 8: Repair action which reassigns the method to another class**

where only repair actions for a set of predefined repair types are generated and tested. Only if no valid repair action could be found in one phase, repair actions of another set of repair types are tested in the next phase. This proved acceptable runtime behavior. The sets of repair types and the order of their execution can be configured by the user. Additionally, a set of repair types can be specified which should be always tested.

## 6. RELATED WORK

There are many approaches handling inconsistencies with *graph transformations*. One is, only delete and attribute propagations or link deletions are supported as with our approach, e.g. in [14, 37, 13]. Another is, inconsistent situations are defined as graph patterns which are searched in the host graph and graph transformations for their resolution are specified such as in [15, 38, 10, 35, 34, 7] to name a few. This procedure is very laborious and will never cover all cases. Additionally, [37] supports the completion of a consistency pattern which is analogous to the changing version of the rule conserving strategy.

Since not all kinds of inconsistencies like behavioral inconsistencies and resolution rules can be expressed easily as graph transformation rules, there are similar approaches [18, 36, 33] which use logic-based rules to detect and resolve inconsistencies exemplified with UML class and sequence diagrams. Also in these approaches, each inconsistency and each resolution has to be defined in advance.

A similar dynamic approach is proposed by [22, 6, 5] within a repair framework where consistency rules are expressed by first order logic formulae. In contrast to the other approaches, repairs are fully generated from these formulae. For a violated formula a set of sets of repair actions is generated, each set of repair actions representing an alternative. A repair action adds, deletes, or changes one model to fix the violated formula or subformula. The alternatives are presented to the user who selects one for execution. This

approach is similar to ours, but differs in some ways: the generated repair actions cannot create model elements and the modifications the user had done on one model are known and used for the generation. Thus, this generation approach is not immediately ready for inconsistency resolution of models which are edited by external tools.

## 7. CONCLUSIONS

In this paper, novel strategies for resolving inconsistencies between graph-based models taking into account only consistency rules specified as triple rules and the damaged sub-graphs are presented. The operations which were performed on a model and lead to the inconsistencies are not considered as it is assumed that the models are edited by external tools. Also, resolution is done on request, thus tolerating inconsistencies. A main principle in resolving inconsistencies presented here is to conserve the applied rule or to apply a similar one and do only small adaptations. As discussed, we think that this is a good option in practice. Not presented in this paper, but nevertheless mentionable is that based on the presented strategies multiple alternative repair actions are derived for one damaged link, even multiple repair actions for the strategy to conserve the applied rule or to apply a similar one. Not all are applicable, i.e. required increments are not available, so that only applicable repair actions are suggested to the user. In the end, the user picks the repair action which fits best. This should not be decided by the tool.

## 8. REFERENCES

- [1] BECKER, S. M. *Integratoren zur Konsistenzsicherung von Dokumenten in Entwicklungsprozessen*. Berichte aus der Informatik. Shaker Verlag, Aachen, Germany, 2007. Doktorarbeit, RWTH Aachen University.
- [2] BECKER, S. M., HEROLD, S., LOHMANN, S., AND WESTFECHTEL, B. A Graph-Based Algorithm for Consistency Maintenance in Incremental and Interactive Integration Tools. *Software and Systems Modeling (SoSyM)* 6, 3 (2007), 287–315.
- [3] BECKER, S. M., AND WESTFECHTEL, B. UML-based Definition of Integration Models for Incremental Development Processes in Chemical Engineering. *Integrated Design and Process Science: Transactions of the SDPS 8:1* (2004), 49–63.
- [4] CZARNECKI, K., AND HELSEN, S. Classification of Model Transformation Approaches. In *Proc. of the Workshop on Generative Techniques in the Context of Model Driven Architecture (OOPSLA 2003)* (2003).
- [5] EGYED, A. Fixing Inconsistencies in UML Design Models. In *Proc. of the 29<sup>th</sup> Intl. Conf. on Software Engineering (ICSE 2007)* (2007), IEEE Computer Society, pp. 292–301.
- [6] EGYED, A., LETIER, E., AND FINKELSTEIN, A. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *ASE* (2008), IEEE, pp. 99–108.
- [7] ENDERS, B. E., HEVERHAGEN, T., GOEDICKE, M., TRÖPFNER, P., AND TRACHT, R. Towards an Integration of Different Specification Methods by Using the ViewPoint Framework. *Transactions of the SDPS 6, 2* (2002), 1–23.

- [8] FINKELSTEIN, A., SPANOUDAKIS, G., AND TILL, D. Managing interference. In *Joint Proc. of the 2<sup>nd</sup> Intl. Software Architecture Workshop (ISAW-2) and Intl. Workshop on Multiple Perspectives in Software Development (Viewpoints 1996) on SIGSOFT 1996 Workshops* (1996), ACM, pp. 172–174.
- [9] FÖRTSCH, S., AND WESTFECHTEL, B. Differencing and Merging of Software Diagrams - State of the Art and Challenges. In *Proc. of the 2<sup>nd</sup> Intl. Conf. on Software and Data Technologies (ICSOF 2007)* (Setubal, Portugal, 2007), J. Filipe, B. Shishkow, and M. Helfert, Eds., INSTICC.
- [10] HAUSMANN, J. H., HECKEL, R., AND SAUER, S. Extended Model Relations with Graphical Consistency Conditions. In Jézéquel et al. [12], pp. 61–74.
- [11] HWAN, C., KIM, P., AND CZARNECKI, K. Synchronizing cardinality-based feature models and their specializations. In *ECMDA-FA* (2005), A. Hartman and D. Kreische, Eds., vol. 3748 of *LNCS*, Springer-Verlag, pp. 331–348.
- [12] JÉZÉQUEL, J. M., HUSSMANN, H., AND COOK, S., Eds. *Proc. of the 5<sup>th</sup> Intl. Conf. on The Unified Modeling Language (UML 2002)* (2002), vol. 2460 of *LNCS*, Springer-Verlag.
- [13] JOUAULT, F., ALLILAIRE, F., BÉZIVIN, J., AND KURTEV, I. Atl: A model transformation tool. *Science of Computer Programming* 72, 1-2 (2008), 31–39.
- [14] KÖNIGS, A. *Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation*. PhD thesis, Technische Universität Darmstadt, Fachbereich Elektrotechnik und Informationstechnik, January 2009. Dissertation.
- [15] KÖNIGS, A., AND SCHÜRR, A. Multi-Domain Integration with MOF and extended Triple Graph Grammars [online]. In *Language Engineering for Model-Driven Software Development* (Dagstuhl, Germany, 2005), no. 04101 in Dagstuhl Seminar Proc., IBFI.
- [16] KÖRTGEN, A. *Modellierung und Realisierung von Konsistenzsicherungs Werkzeugen für simultane Dokumentenentwicklung*. Berichte aus der Informatik. Shaker Verlag, Aachen, Germany, 2009. Doktorarbeit, RWTH Aachen University.
- [17] KÖRTGEN, A., BECKER, S. M., AND HEROLD, S. A Graph-Based Framework for Rapid Construction of Document Integration Tools. In *Proc. of the 11<sup>th</sup> World Conf. on Integrated Design & Process Technology (IDPT '07)* (2007), SDPS, p. 13 pp.
- [18] LIU, W., EASTERBROOK, S., AND MYLOPOULOS, J. Rule Based Detection of Inconsistency in UML Models. In Jézéquel et al. [12], pp. 106–123.
- [19] LONG, E. Transform, edit, and reverse-engineer a UML Model into Java Source Code. Tech. rep., IBM Corporation, 2007.
- [20] MICROSOFT CORPORATION. MSDN Library - Visual Studio SDK, 2008.
- [21] NAGL, M., AND MARQUARDT, W., Eds. *Collaborative and Distributed Chemical Engineering Design Processes: Better Understanding and Substantial Support Results of the CRC IMRPOVE*, vol. 4970 of *LNCS*. Springer-Verlag, 2008.
- [22] NENTWICH, C., EMMERICH, W., AND FINKELSTEIN, A. Consistency Management with Repair Actions. In *Proc. of the 25<sup>th</sup> Intl. Conf. on Software Engineering (ICSE 2003)* (2003), IEEE Computer Society, pp. 455–464.
- [23] NUSEIBEH, B., EASTERBROOK, S., AND RUSSO, A. Leveraging Inconsistency in Software Development. *Computer* 33, 4 (2000), 24–29.
- [24] OMG. UML 2.0: Infrastructure, V2.1.2. online, 2007.
- [25] OMONDO EUROPA. EclipseUML Studio, 2007.
- [26] PRATT, T. W. Pair Grammars, Graph Languages and String-to-Graph Translations. *Computer and Systems Sciences* 5, 6 (1971), 560–595.
- [27] SCHÜRR, A. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of the 20<sup>th</sup> Intl. Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)* (1995), vol. 903 of *LNCS*, Springer-Verlag, pp. 151–163.
- [28] SPANOUDAKIS, G., AND FINKELSTEIN, A. Reconciling Requirements: A Method for Managing Interference, Inconsistency and Conflict. *Annals of Software Engineering* 3 (1997), 433–457.
- [29] SPANOUDAKIS, G., AND ZISMAN, A. Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering*, S. K. Chang, Ed., vol. 1. World Scientific Publishing Co, 2001, pp. 329–380.
- [30] STEVENS, P., WHITTLE, J., AND BOOCH, G., Eds. *Proc. of the 6<sup>th</sup> Intl. Conf. on The Unified Modeling Language (UML 2003)* (2003), vol. 2863 of *LNCS*, Springer-Verlag.
- [31] THE ECLIPSE FOUNDATION, 2008.
- [32] TRATT, L. Model Transformations and Tool Integration. *Software and Systems Modelling (SoSym)* 4:2 (2005), 112–122.
- [33] VAN DER STRAETEN, R. *Inconsistency Management in Model-driven Engineering: an Approach using Description Logics*. PhD thesis, Vrije Universiteit Brussel, 2005.
- [34] VAN DER STRAETEN, R., AND D’HONDT, M. Model refactorings through rule-based inconsistency resolution. In *Proc. of the 2006 ACM symposium on Applied computing (SAC 2006)* (New York, NY, USA, 2006), ACM, pp. 1210–1217.
- [35] VAN DER STRAETEN, R., AND MENS, T. Incremental Resolution of Model Inconsistencies. In *Proc. of 18<sup>th</sup> Intl. Workshop of Recent Trends in Algebraic Development Techniques (WADT 2006)* (2007), vol. 4409 of *LNCS*, Springer-Verlag, pp. 111–126.
- [36] VAN DER STRAETEN, R., MENS, T., SIMMONDS, J., AND JONCKERS, V. Using Description Logics to Maintain Consistency Between UML Models. In Stevens et al. [30], pp. 326–340.
- [37] WAGNER, R. *Inkrementelle Modellsynchronisation*. PhD thesis, Universität Paderborn, Institut für Informatik, Fachgebiet Softwaretechnik, 2009. Dissertation.
- [38] WAGNER, R., GIESE, H., AND NICKEL, U. A Plug-In for Flexible and Incremental Consistency Management. In Stevens et al. [30].

# Reasoning about Consistency in Model Merging

Mehrdad Sabetzadeh<sup>†</sup> Shiva Nejati<sup>†</sup> Marsha Chechik<sup>‡</sup> Steve Easterbrook<sup>‡</sup>

<sup>†</sup>Simula Research Laboratory  
Oslo, Norway  
{mehrdad, shiva}@simula.no

<sup>‡</sup> University of Toronto  
Toronto, Canada  
{chechik, sme}@cs.toronto.edu

## ABSTRACT

Models undergo a variety of transformations throughout development. One of the key transformations is merge, used when developers need to combine a set of models with respect to the overlaps between them. A major question about model transformations in general, and merge in particular, is what consistency properties are preserved across the transformations and what consistency properties may need to be re-checked (and if necessary, re-established) over the result. In previous work [18], we developed a technique based on category-theoretic colimits for merging sets of inter-related models. The use of category theory leads to the preservation of the algebraic structure of the source models in the merge; however, this does not directly provide a characterization of the (in)consistency properties that carry over from the source models to the result, because consistency properties are predominantly expressed as logical formulas. Hence, an investigation of the connections between the “algebraic” and “logical” properties of model merging became necessary.

In this paper, we undertake such an investigation and use techniques from finite model theory [9] to show that the use of colimits indeed leads to the preservation of certain logical properties. Our results have implications beyond the merge framework in [18] and are potentially useful for the broad range of techniques in the graph transformation and algebraic specification literature that use colimits as the basis for model manipulations.

## Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications

## General Terms

Design, Verification

## Keywords

Inconsistency, Merge, Logical Preservation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

## 1. INTRODUCTION

In the past several years, we have been studying the problem of model integration, particularly in situations where the models are originating from distributed sources of information. Many activities in model-based development fall under the umbrella of integration. These include (1) *merging*, used to build a global view of a set of overlapping perspectives (e.g., [23, 20, 18, 10]); (2) *composition*, used to assemble a set of autonomous but interacting components that run sequentially or in parallel (e.g., [2, 5, 6]); and (3) *weaving*, used in aspect-oriented development to incorporate cross-cutting concerns into a base system (e.g., [22]).

Our position towards the integration problem has been that the integration operators (e.g., merge, compose, weave) must tolerate inconsistency. That is, the operators must work for *any* given set of models, even when the models are inconsistent. This position is motivated by two well-known observations: First, immediate resolution of inconsistency can be disruptive in projects where ambiguities and conflicts tend to occur frequently [11]. Second, maintaining consistency at all times can be counter-productive because it may lead to premature commitment to design decisions that have not yet been sufficiently analyzed [12].

In light of our position, it is important to understand how different consistency properties are affected by the integration operations. Specifically:

- If all the source models are consistent with respect to a given consistency property, will the integrated model be consistent with respect to that property as well?
- If there is an inconsistency in the source models, will the inconsistency necessarily carry over to the integrated model?

Answering the first question is interesting to enable compositional reasoning about consistency. Answering the second question is useful for understanding the nature of an inconsistency. In particular, inconsistencies that are due to incomplete information in the individual source models can be automatically resolved in the integrated models when the source models are complementary and address each other’s areas of incompleteness. For example, an abstract class with no descendants in a UML class diagram might be seen as inconsistent. But this class might be inherited from in other models and hence the overall view might still be consistent. In contrast, cyclic inheritance in a UML class diagram cannot be resolved in the integrated model (unless the integrated model omits information from the source models).

Since consistency rules are often described in logical languages (e.g., first order logic), we are interested in studying how different integration operators preserve the logical properties of models. In general, property preservation is a powerful tool for reasoning about model transformations. The main question that property preservation tackles is the following: If a property (formula)  $\varphi$  in some logic holds over a model  $M$ , will  $\varphi$  also hold over a model  $M'$  derived from  $M$  via some transformation?

In this paper, we discuss the logical property preservation characteristics of our merge operator in [18]. The merge operator is based on category theory which has been widely used as a theoretical basis for characterizing model merging. In a categorical setting, merge is typically performed by computing a colimit – an algebraic construct for combining a set of objects interrelated by a set of mappings. While colimits provide an effective and mathematically precise way for merge, their pure algebraic characterization is not directly applicable for reasoning about the logical properties of model merging. Specifically, given a property  $\varphi$  expressed in a particular logic, one cannot readily determine from the definition of colimit whether  $\varphi$  is preserved from the source models to the merged model.

We use techniques from finite model theory [9] to show that colimits indeed preserve a certain class of logical properties. The logical language we use as the basis for our work is first order logic extended with least fixpoints. Extension with fixpoints is important, because standard first order logic cannot express properties that require the computation of reachability. For example, acyclic inheritance for UML class diagrams is not expressible in standard first order logic. Our results have implications beyond our merge framework in [18] and are potentially useful for the broad range of techniques in the graph transformation [16] and algebraic specification [1] communities that use colimits as the basis for model manipulations.

The remainder of this paper is structured as follows: In Section 2, we provide background information on our merge algorithm and present the logical preliminaries for our work. In Section 3, we describe our general logical preservation results for colimits; and in Section 4, we use these general results to reason about the preservation of some logical expressions that are frequently used in consistency rules. We conclude in Section 5 with a summary and directions for future work.

## 2. BACKGROUND

### 2.1 Structural Model Merging

We first briefly review our merge operator. For more information, see [18]. The operator hinges on three abstractions: *models*, *mappings*, and *interconnection diagrams*. Each model is described as a graph, and each mapping – as a binary relation over two models equating their corresponding elements. Mappings preserve type information, i.e., they do not equate elements that have different types. Further, they preserve structure, i.e., if a mapping  $R$  maps an edge  $e$  to an edge  $e'$ , it must also map the source and target of  $e$  to the source and target of  $e'$ , respectively.

The third abstraction, the interconnection diagram, captures a set of models and a set of known or hypothesized mappings between them. An example interconnection diagram is shown in Figure 1. In this example,  $M_1, \dots, M_4$

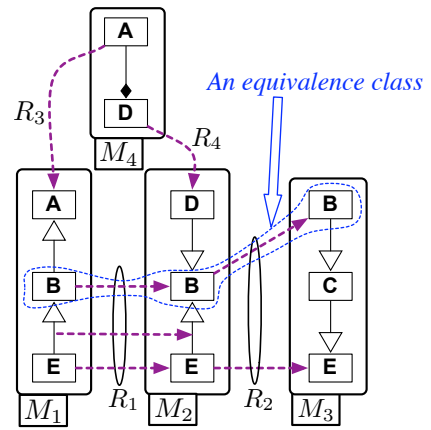


Figure 1: Example interconnection diagram

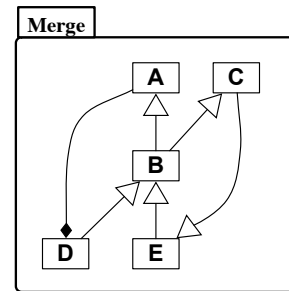


Figure 2: Merge of the diagram in Figure 1

are simple UML class diagrams with their overlapping parts specified through four mappings,  $R_1, \dots, R_4$  (depicted using directed dashed lines). A simpler example with just two models,  $M_1, M_2$ , and one mapping,  $R$ , is shown in Figure 3(a). A third example is given in Figure 4(a), where the shared parts of two models,  $M_2, M_3$ , are captured by a third model  $M_1$  and two mappings  $R_1, R_2$ .

The input to the merge algorithm is an interconnection diagram  $D = \langle M_1, \dots, M_i, R_1, \dots, R_j \rangle$ . The algorithm works by unifying elements in  $M_1, \dots, M_i$  that fall into the same equivalence class induced by  $R_1, \dots, R_j$ . As an example, we have delineated by thin dashed lines one of the several equivalence classes in Figure 1. Note that each unmapped element in the input models falls into a distinct equivalence class of its own.

For convenience, in the example shown in Figure 1, we used a consistent vocabulary for naming the elements of  $M_1, \dots, M_4$ , hence defining  $R_1, \dots, R_4$  based on name equalities. In general, models may not have a common vocabulary, and mappings are not necessarily based on vocabulary similarities (e.g., see the examples in Figures 3(a) and 4(a)).

The merged model has exactly one element corresponding to each equivalence class. Since mappings denote equality of mapped element pairs and hence are symmetric, the directionality of mappings is ignored in the computation of equivalence classes.

Figure 2 shows the resulting merge for the interconnection diagram of Figure 1. The merge provides interesting insights about how consistency properties can be broken across merge. For example, we may have consistency rules that check for multiple or cyclic inheritance in UML class diagrams. Obviously, these rules are satisfied over the in-

dividual source models in Figure 1, but the global view of the system (i.e., the merge) is inconsistent. In particular, in Figure 2: B has two parents; and B, C, E form a cycle. In Section 4, we provide a systematic explanation of what properties of the source models carry over to the merge and what properties do not.

## 2.2 Logical Background

First Order (FO) logic is one of the most commonly used logical languages for expressing consistency rules [11] and is used as the basis for our work here. FO by itself is not expressive enough to describe properties that involve reachability or cycles. To address this limitation, one can add to FO a least fixpoint operator, obtaining the least fixpoint logic (LFP) [9]. Below, we first formally define the notion of relational structure and FO. We then define the concept of least fixpoint and show how FO can be extended into LFP. Our exposition follows the standard approach in finite model theory (e.g., see [9]).

**Definition 2.1 (relational structure)** A *(relational) structure* is an object  $\mathfrak{A} = (A, R_1, \dots, R_m)$ , where  $A$  is a nonempty set,  $m$  is a natural number,  $R_1, \dots, R_m$  are abstract relation symbols with associated arities  $k_1, \dots, k_m$  (nonnegative integers), and each  $R_i$  is a  $k_i$ -ary relation on  $A$ .

The set  $A$  is called the *universe* of  $\mathfrak{A}$ . The sequence of relation symbols  $R_1, \dots, R_m$  together with corresponding arities  $k_1, \dots, k_m$  comprise the *vocabulary* of  $\mathfrak{A}$ . We usually denote a vocabulary by  $\sigma$ . Relation  $R_i^{\mathfrak{A}}$  is called the *interpretation* of a relation symbol  $R_i$  in  $\mathfrak{A}$ .

FO formulas in a vocabulary  $\sigma$  are built up from atomic formulas using negation, conjunction, disjunction, and existential and universal quantification:

$$\begin{aligned} \varphi ::= & x = y \mid R(x_1, \dots, x_n) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \\ & \exists x\varphi(x) \mid \forall x\varphi(x) \end{aligned}$$

In the above,  $x, y$  and  $x_1, \dots, x_n$  are variables,  $R$  is an  $n$ -ary relation symbol in  $\sigma$ , and  $\varphi_1$  and  $\varphi_2$  are formulas.

Given a set  $U$ , let  $\mathcal{P}(U)$  denote its powerset. A set  $X \subseteq U$  is said to be a *fixpoint* of a mapping  $F : \mathcal{P}(U) \rightarrow \mathcal{P}(U)$  if  $F(X) = X$ . A set  $X \subseteq U$  is a *least fixpoint* of  $F$  if it is a fixpoint, and for every other fixpoint  $Y$  of  $F$ , we have  $X \subseteq Y$ . The least fixpoint of  $F$  is denoted by  $\mathbf{lfp}(F)$ . Least fixpoints are guaranteed to exist only if  $F$  is *monotone*. That is,

$$X \subseteq Y \text{ implies } F(X) \subseteq F(Y).$$

We now add a least fixpoint operator to FO. Suppose we have a vocabulary  $\sigma$ , and an additional relation symbol  $R \notin \sigma$  of arity  $k$ . Let  $\varphi(R, x_1, \dots, x_k)$  be a formula with vocabulary  $\sigma \cup \{R\}$ . For a structure  $\mathfrak{A}$  with vocabulary  $\sigma$ , the formula  $\varphi(R, \vec{x})$  yields a mapping  $F_\varphi : \mathcal{P}(A^k) \rightarrow \mathcal{P}(A^k)$  defined as follows:

$$F_\varphi(X) = \{\vec{a} \mid \mathfrak{A} \models \varphi(X/R, \vec{a})\}$$

The notation  $\varphi(X/R, \vec{a})$  means that  $X$  is substituted for  $R$  in  $\varphi$ . More precisely, if  $\mathfrak{A}'$  is a  $(\sigma \cup \{R\})$ -structure expanding  $\mathfrak{A}$ , in which  $R$  is interpreted as  $X$ , then  $\mathfrak{A}' \models \varphi(\vec{a})$ .

To ensure that  $F_\varphi$  is monotone, we impose certain restrictions. Given a formula  $\varphi$  that may contain a relation symbol

$R$ , we say that an occurrence of  $R$  is *negative* if it is under the scope of an odd number of negations, and *positive*, otherwise. We say that a formula is *positive in  $R$*  if there are no negative occurrences of  $R$  in it, i.e., either all occurrences of  $R$  are positive, or there are none at all.

**Lemma 2.2** [3] *If  $\varphi(R, \vec{x})$  is positive in  $R$ , then  $F_\varphi$  is monotone.*

**Definition 2.3 (least fixpoint logic)** [9] The *least fixpoint logic (LFP)* extends FO with the following formula building rule:

- if  $\varphi(R, \vec{x})$  is a formula positive in  $R$ , where  $R$  is  $k$ -ary, and  $\vec{t}$  is a tuple of terms, where  $|\vec{x}| = |\vec{t}| = k$ , then

$$[\mathbf{lfp}_{R, \vec{x}}\varphi(R, \vec{x})](\vec{t})$$

is a formula, whose free variables are those of  $\vec{t}$ .

The semantics is defined as follows:

$$\mathfrak{A} \models [\mathbf{lfp}_{R, \vec{x}}\varphi(R, \vec{x})](\vec{a}) \quad \text{iff} \quad \vec{a} \in \mathbf{lfp}(F_\varphi).$$

**Example 2.4 (reachability)** Consider graphs whose edge relation is  $E$ , and let

$$\varphi(R, x, y) = E(x, y) \vee \exists z (E(x, z) \wedge R(y, z)).$$

Reachability, i.e., the transitive closure of  $E$ , is characterized by the formula

$$\psi(x, y) = [\mathbf{lfp}_{R, x, y}\varphi(R, x, y)](x, y).$$

That is,  $\psi(a, b)$  holds over a graph  $G$  iff there is a path from  $a$  to  $b$  in  $G$ .

## 2.3 Homomorphisms and Preservation of Logical Properties

Our merge framework embeds each source model into the merge through a homomorphism. The existence of these homomorphisms leads to the preservation of certain consistency properties. Below, we review the theoretical results underlying our discussion of property preservation in Section 3. We begin with a definition of homomorphism:

**Definition 2.5 (homomorphism)** Let  $\mathfrak{A} = (A, R_1^{\mathfrak{A}}, \dots, R_m^{\mathfrak{A}})$  and  $\mathfrak{B} = (B, R_1^{\mathfrak{B}}, \dots, R_m^{\mathfrak{B}})$  be structures in the same vocabulary. A *homomorphism* from  $\mathfrak{A}$  to  $\mathfrak{B}$  is a function  $h : A \rightarrow B$  such that  $h(R_i^{\mathfrak{A}}) \subseteq R_i^{\mathfrak{B}}$ , i.e., if  $(a_1, \dots, a_{k_i}) \in R_i^{\mathfrak{A}}$  then  $(h(a_1), \dots, h(a_{k_i})) \in R_i^{\mathfrak{B}}$  for every  $1 \leq i \leq m$ .

The first result about property preservation under homomorphisms, dating back to the 1950's, is the Los-Tarski-Lyndon Theorem:

**Theorem 2.6 (homomorphism preservation theorem)** (e.g., see [14, 15]) *A first order formula is preserved under homomorphisms on all structures (finite and infinite) if and only if it is equivalent to an existential positive formula, i.e., a formula without negation and universal quantification.*

The existential positive fragment of FO is denoted  $\exists\text{FO}^+$ . For our purposes, we are interested in finite structures only, and like many classical mathematical logic results that fail in the finite case (e.g., compactness), there is the danger that the above result may fail as well when restricted to finite structures. Fortunately, this is not the case.

**Theorem 2.7 (h. p. t. in the finite)** [15] *A first order formula is preserved under homomorphisms on finite structures if and only if it is equivalent to an  $\exists FO^+$  formula.*

The forward direction of the if-and-only-if (i.e., sufficiency) in the above result can be extended to the existential positive fragment of LFP, denoted  $\exists LFP^+$ .

**Lemma 2.8** *Every  $\exists LFP^+$  formula is preserved under homomorphisms on finite structures.*

A proof of the above lemma is provided in the appendix. The lemma is the basis for the results we present in Section 3.

### 3. GENERAL PRESERVATION RESULTS

Our merge framework offers three key features [18]:

*F1* Merge yields a family of mappings, in our case graph homomorphisms, one from each source model onto the merged model. This feature ensures that the merge does not lose information, i.e., it represents all the source models completely.

*F2* The merged model does not contain any unmapped elements, i.e., every element in the merged model is the image of some element in the source models. This feature ensures minimality, i.e., the merge does not introduce information that is not present in or implied by the source models.

*F3* Merge respects the mappings in the source system, i.e., the image of each element in the merged model remains the same, no matter which path through the mappings in the source system one follows. This feature ensures non-redundancy. More precisely, if a concept appears in more than one source model, only one copy of it appears in the merged model.

From *F1* and Lemma 2.8 (in Section 2.3), it follows that the result of our merge procedure preserves the existential positive fragment of LFP.

**Theorem 3.1** *If an existential positive LFP formula  $\varphi$  is satisfied by some source model  $M$ , any merge in which  $M$  participates satisfies  $\varphi$  as well.*

By *F2* and the above theorem, we obtain the following result regarding preservation of universal properties.

**Theorem 3.2** *Let  $\varphi(x)$  be an existential positive LFP formula with a free variable  $x$ . If the formula  $\psi = \forall x \varphi(x)$  is satisfied by all the source models,  $\psi$  is satisfied by any merge of the models as well.*

Notice that Theorem 3.2 allows the introduction of *only one* universal quantifier. To gain intuition on what happens when additional universal quantifiers are introduced, consider the system in Figure 3(a) and let the relation  $E(x, y)$  denote the graph edge relation. Both models in Figure 3(a) are complete graphs and hence satisfy the property  $\forall x \forall y \text{Node}(x) \wedge \text{Node}(y) \Rightarrow E(x, y)$ <sup>1</sup>. However, the property is violated over the resulting merge shown in Figure 3(b),

<sup>1</sup>The property uses implication and hence has negation. But the negation can be resolved, because every element in the universe that is not a node is an edge. Therefore, the property is equivalent to  $\forall x \forall y \text{Edge}(x) \vee \text{Edge}(y) \vee E(x, y)$ .

because there is no edge from node  $a$  to node  $d$  and vice versa. The general observation here is that, when there is more than one universal quantifier, universally quantified variables can be assigned values from non-shared parts of *different* source models. In such a case, property satisfaction over the individual source models may not extend to the merge.

Currently, we do not know whether *F3* leads to further property preservation results. This is an issue that we plan to investigate in future work.

### 4. PRESERVATION OF (IN)CONSISTENCY

In previous work [19], we identified three general types of expressions commonly used in structural consistency properties. These are:

- *Compatibility expressions*, used for ensuring compatibility of the type of an edge with the types of its endpoints.
- *Multiplicity expressions*, used for defining a minimum and a maximum number for edges of a given type incident to a node.
- *Reachability expressions*, used for checking existence of paths of edges of a given type between two nodes.

Below, we use results of Section 3 to reason about the preservation of these expressions.

*Compatibility Properties.* Preservation of compatibility properties can be established directly through algebraic means, but it is interesting to see if the same can be done through logical means. For example, in a class diagram, an edge of type “implements” must relate a class to an interface; otherwise, the diagram would not be well-formed. This property can be formalized as follows:

$$\mathbf{C1} = \forall e (\text{Edge}(e) \wedge \text{Type}(e, \text{“implements”}) \Rightarrow \mathbf{Compatible}_{\text{class,interface}}(e))$$

where **Edge** is the set of edges of the graph representing a class diagram, **Type** is a binary relation between the set of edges and different types of relations between classes, and  $\mathbf{Compatible}_{\text{class,interface}}(e)$  is a constraint that verifies whether the source and target nodes of edge  $e$  are of type **class** and **interface**, respectively. The general form of this constraint can be formalized using an existential positive formula as follows:

$$\mathbf{Compatible}_{\alpha,\beta}(e) = \exists n (\exists m (\text{Source}(e, n) \wedge \text{Target}(e, m) \Rightarrow \text{Type}(n, \alpha) \wedge \text{Type}(m, \beta)))$$

where **Source** and **Target** are binary relations, respectively giving the source and target node for a given edge.

The sub-formula **Type**( $e$ , “implements”) of **C1** appears in negated form, but the negation can be resolved, knowing that (1) the set of types is fixed and, (2) every element has a type. More precisely, if the set of types is  $\{t_1, \dots, t_n\}$ , the formula  $\neg \text{Type}(e, t_\ell)$  can be replaced with  $\bigvee_{i \neq \ell} \text{Type}(e, t_i)$ . Hence, by Theorem 3.2, **C1** is preserved.

*Multiplicity Properties.* One can show through simple counterexamples that *none* of the following lift from the source models to the merge:



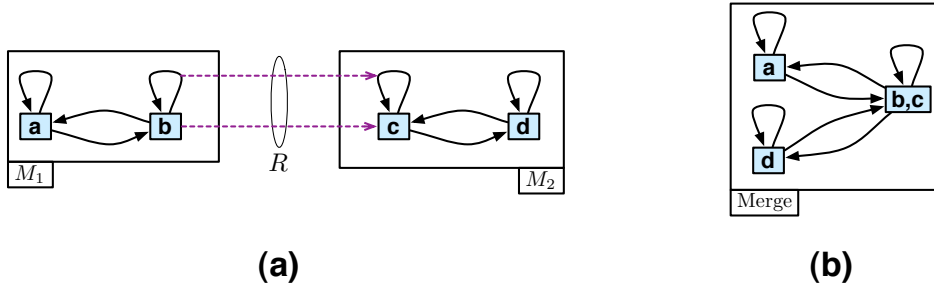


Figure 3: Illustration for violation of universal properties

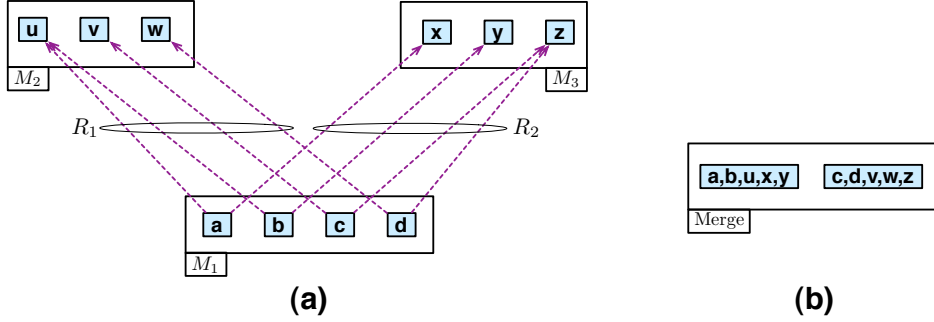


Figure 4: Illustration for violation of multiplicity properties

- There exists at least  $c$  elements satisfying  $\varphi$ .
- There exists exactly  $c$  elements satisfying  $\varphi$ .
- There exists at most  $c$  elements satisfying  $\varphi$ .

It is easy to see why the “exactly” and “at most” cases do not get preserved, noting that merge normally has more information than any of the source models. To understand why the “at least” case is not preserved, note that homomorphisms (and functions as well) are not necessarily one-to-one, and can therefore *shrink* the number of elements satisfying a property. For example, consider the system of models in Figure 4(a) and its merge in Figure 4(b). For simplicity, the models are discrete graphs, i.e., sets, and their mappings are functions. Although  $M_1, M_2, M_3$  all satisfy the property “there exists at least three (distinct) nodes”, the merge has only two nodes, hence violating the property. It is important to mention that the flexibility to fuse together multiple elements of the same source model is not an undesirable feature and is indeed valuable when one needs to perform an abstraction during merge [7].

*Reachability Properties.* An interesting consequence of Theorem 3.1 is the preservation of paths in the merge. Recall that we gave a formalization for the reachability property in Example 2.4. To see how this can be used for reasoning about consistency, consider the following consistency rule over class diagrams: “Every abstract class must have a concrete implementation”. This rule is formally expressed as follows:

$$\mathbf{C2} = \forall c ((\text{Type}(c, \text{“class”}) \wedge \text{Abstract}(c)) \Rightarrow \exists c' (\text{Concrete}(c') \wedge \text{Reachable}_{\text{extends}}(c', c)))$$

where  $\text{Reachable}_{\text{extends}}(x, y)$  holds iff a path from  $x$  to  $y$

made up of edges of type “extends” exists. Using the argument we gave when discussing preservation of compatibility properties, we know that the negation of  $\text{Type}(c1, \text{“class”})$  can be resolved. Further,  $\neg \text{Abstract}(c1)$  can be replaced with a positive property, say,  $\text{Concrete}(c1)$ . It now follows from Theorem 3.2 that  $\mathbf{C2}$  is preserved.

Note that our results can be used for reasoning about preservation of inconsistency as well. For example, consider the following rule:

$$\mathbf{C3} = \exists c ((\text{Type}(c, \text{“class”}) \wedge \text{Reachable}_{\text{extends}}(c, c)))$$

This rule holds over a class diagram  $M$  when the inheritance hierarchy in  $M$  is cyclic, i.e.,  $M$  is inconsistent. By Theorem 3.1, we can conclude that any merged model that has  $M$  as a source model satisfies  $\mathbf{C3}$  as well, and hence is also inconsistent.

## 5. CONCLUSION

In this paper, we showed that the use of algebraic colimits for model merging leads to the preservation of certain logical properties. We used our results to formally reason about the preservation of consistency properties across merge.

Based on our recent survey of existing model merging techniques [17], algebraic theories, including category theory, lattice theory, and formal concept analysis, are increasingly being used for characterizing model integration problems. What makes these theories particularly attractive is the level of abstraction to which they lead, allowing the merge process to be described in a flexible and highly generic way. At the same time, one must account for the fact that merge is often an intermediate step for activities such as behavioural synthesis [23, 21], reasoning over global behaviours of systems [4], and data integration and exchange [8, 13].



To facilitate these activities, it is crucial to be able to reason about the preservation of semantic properties (including consistency properties) of the source models in merge. Doing so requires establishing proper connections between the algebraic techniques used in model integration and the logical techniques used in the activities named above. This is a non-trivial task but is an essential step toward making model integration more effective in practice.

For future work, we would like to provide a full logical characterization of colimits. In particular, the logical implications of  $F3$ , described in Section 3, is unknown to us at the moment and need to be revisited in the future. Further, it may be possible to trade off development flexibility in favour of a broader class of preserved properties, e.g., by using more constrained mappings for relating models, or by placing restrictions on the patterns used for interconnecting the models. We leave an elaboration of these topics to future investigation. Lastly, we would like to explore the application of our results for checking the consistency of model manipulations in graph transformation and algebraic specification approaches that are based on colimits.

## 6. REFERENCES

- [1] E. Astesiano, H. Kreowski, and B. Krieg-Brueckner, editors. *Algebraic Foundations of Systems Specification*. Springer-Verlag, Secaucus, NJ, USA, 1999.
- [2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [3] B. Davey and H. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [4] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 411–420, 2001.
- [5] J. Hay and J. Atlee. “Composing Features and Resolving Interactions”. In *SIGSOFT ’00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 110–119, 2000.
- [6] M. Jackson and P. Zave. “Distributed Feature Composition: a Virtual Architecture for Telecommunications Services”. *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998.
- [7] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: The state of the art. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, editors, *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings. IBFI, 2005.
- [8] M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the 21st Symposium on Principles of Database Systems*, pages 233–246, 2002.
- [9] L. Libkin. *Elements Of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
- [10] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. “Matching and Merging of Statecharts Specifications”. In *ICSE ’07: Proceedings of the 29th International Conference on Software Engineering*, pages 54–64, 2007.
- [11] C. Nentwich, W. Emmerich, and A. Finkelstein. “Consistency Management with Repair Actions”. In *ICSE ’03: Proceedings of the 25 International Conference on Software Engineering*, pages 455–464, 2003.
- [12] B. Nuseibeh, S. Easterbrook, and A. Russo. “Making Inconsistency Respectable in Software Development”. *The Journal of Systems and Software*, 58(2):171–180, 2001.
- [13] L. Popa, Y. Velegrakis, R. Miller, M. Hernández, and R. Fagin. Translating web data. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 598–609, 2002.
- [14] E. Rosen. Some aspects of model theory and finite structures. *The Bulletin of Symbolic Logic*, 8(3):380–403, 2002.
- [15] B. Rossman. Existential positive types and preservation under homomorphisms. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 467–476, 2005.
- [16] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: Foundations*, volume 1. World Scientific, River Edge, NJ, USA, 1997.
- [17] M. Sabetzadeh. *Merging and Consistency Checking of Distributed Models*. PhD thesis, University of Toronto, 2008.
- [18] M. Sabetzadeh and S. Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requirements Engineering Journal*, 11(3):174–193, 2006.
- [19] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. “Consistency Checking of Conceptual Models via Model Merging”. In *RE ’07: Proceedings of 15th IEEE International Requirements Engineering Conference*, pages 221–230, 2007.
- [20] S. Uchitel and M. Chechik. “Merging Partial Behavioural Models”. In *SIGSOFT ’04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 43–52, 2004.
- [21] S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 188–197, 2001.
- [22] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi. “An Expressive Aspect Composition Language for UML State Diagrams”. In *MoDELS ’07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, pages 514–528, 2007.
- [23] J. Whittle and J. Schumann. “Generating Statechart Designs from Scenarios”. In *ICSE ’00: Proceedings of 22nd International Conference on Software Engineering*, pages 314–323. ACM Press, May 2000.

## APPENDIX

### A. PROOF FOR LEMMA 2.8

Let  $\mathfrak{A} = (A, R_1^{\mathfrak{A}}, \dots, R_m^{\mathfrak{A}})$  and  $\mathfrak{B} = (B, R_1^{\mathfrak{B}}, \dots, R_m^{\mathfrak{B}})$  be a pair of relational structures over vocabulary  $\sigma = (R_1, \dots, R_m)$ . Let  $h : \mathfrak{A} \rightarrow \mathfrak{B}$  be a homomorphism. We show that for every  $\varphi \in \exists LFP^+$  and for every  $\vec{a} \in A^k$

$$\mathfrak{A} \models \varphi(\vec{a}) \Rightarrow \mathfrak{B} \models \varphi(h(\vec{a}))$$

where  $h(\vec{a}) = (h(a_1), \dots, h(a_k))$ .

The proof for  $\varphi \in \exists FO^+ \cap \exists LFP^+$  follows from Theorem 2.7. Below, we provide a proof for least fixpoint formulas.

Let  $\varphi(\vec{x}) = [\mathbf{lfp}_{R, \vec{y}} \alpha(R, \vec{y})](\vec{x})$ . By the definition of  $\mathbf{lfp}$ , for every structure  $\mathfrak{A}$ , the formula  $\varphi$  yields a mapping  $F_{\alpha, \mathfrak{A}} : \mathcal{P}(A^k) \rightarrow \mathcal{P}(A^k)$  defined as follows:

$$F_{\alpha, \mathfrak{A}}(X) = \{\vec{a} \mid \mathfrak{A} \models \alpha(X/R, \vec{a})\}$$

By Definition 2.3 and Knaster-Tarski's fixpoint theorem, for every  $\vec{a} \in A^k$  we have:

$$\vec{a} \in \bigcup_{i=0}^{\infty} F_{\alpha, \mathfrak{A}}^i(\emptyset) \Leftrightarrow \mathfrak{A} \models [\mathbf{lfp}_{R, \vec{y}} \alpha(R, \vec{y})](\vec{a})$$

We first prove by induction that  $h(F_{\alpha, \mathfrak{A}}^i(\emptyset)) \subseteq F_{\alpha, \mathfrak{B}}^i(\emptyset)$ .

**Base case:** Let  $\vec{a} \in F_{\alpha, \mathfrak{A}}(\emptyset)$ . Then,  $\mathfrak{A} \models \alpha(\emptyset, \vec{a})$ . Since  $\mathfrak{A}$  is a substructure of  $\mathfrak{B}$  by  $h$  and since  $h(\emptyset) = \emptyset$ , we have  $\mathfrak{B} \models \alpha(\emptyset, h(\vec{a}))$ . Thus,  $h(\vec{a}) \in F_{\alpha, \mathfrak{B}}(\emptyset)$ .

**Inductive step:** Let  $\vec{a} \in F_{\alpha, \mathfrak{A}}^i(\emptyset)$ . Then,  $\mathfrak{A} \models \alpha(F_{\alpha, \mathfrak{A}}^{i-1}(\emptyset), \vec{a})$ . Since  $\mathfrak{A}$  is a substructure of  $\mathfrak{B}$  by  $h$ , we have  $\mathfrak{B} \models \alpha(h(F_{\alpha, \mathfrak{A}}^{i-1}(\emptyset)), h(\vec{a}))$ . Thus,  $h(\vec{a}) \in F_{\alpha, \mathfrak{B}}(h(F_{\alpha, \mathfrak{A}}^{i-1}(\emptyset)))$ . By the inductive hypothesis and since  $F_{\alpha, \mathfrak{B}}$  is monotone,  $h(\vec{a}) \in F_{\alpha, \mathfrak{B}}^i(\emptyset)$ .

Thus,

$$h(\bigcup_{i=0}^{\infty} F_{\alpha, \mathfrak{A}}^i(\emptyset)) \subseteq \bigcup_{i=0}^{\infty} F_{\alpha, \mathfrak{B}}^i(\emptyset) \quad (1)$$

Therefore,

$$\begin{aligned} \mathfrak{A} \models \varphi(\vec{a}) & \Leftrightarrow \\ (\text{By assumption } \varphi(\vec{a}) = [\mathbf{lfp}_{R, \vec{y}} \alpha(R, \vec{y})](\vec{a})) & \\ \mathfrak{A} \models [\mathbf{lfp}_{R, \vec{y}} \alpha(R, \vec{y})](\vec{a}) & \Leftrightarrow \\ (\text{By Definition 2.3 and Knaster-Tarski's Theorem}) & \\ \vec{a} \in \bigcup_{i=0}^{\infty} F_{\alpha, \mathfrak{A}}^i(\emptyset) & \Leftrightarrow \\ (\text{Since } h \text{ is homomorphism}) & \\ h(\vec{a}) \in h(\bigcup_{i=0}^{\infty} F_{\alpha, \mathfrak{A}}^i(\emptyset)) & \Rightarrow \\ (\text{By (1)}) & \\ h(\vec{a}) \in \bigcup_{i=0}^{\infty} F_{\alpha, \mathfrak{B}}^i(\emptyset) & \Leftrightarrow \\ (\text{By Definition 2.3 and Knaster-Tarski's Theorem}) & \\ \mathfrak{B} \models [\mathbf{lfp}_{R, \vec{y}} \alpha(R, \vec{y})](h(\vec{a})) & \Leftrightarrow \\ (\text{By definition of } \mathbf{lfp}) & \\ \mathfrak{B} \models \varphi(h(\vec{a})) & \end{aligned}$$

# Utilizing the Relationships Between Inconsistencies for more Effective Inconsistency Resolution

Alexander Nöhrer  
Institute for Systems Engineering and  
Automation  
Johannes Kepler University Linz, Austria  
alexander.noehrer@jku.at

Alexander Egyed  
Institute for Systems Engineering and  
Automation  
Johannes Kepler University Linz, Austria  
alexander.egyed@jku.at

## ABSTRACT

During software modeling, engineers are prone to making mistakes. State-of-the-art tool support can help detect these mistakes and point to inconsistencies in the model. They even can generate fixing actions for these inconsistencies. However state-of-the-art approaches process inconsistencies individually, assuming that each single inconsistency is a manifestation of an individual defect. This paper presents our vision of the next steps in inconsistency resolution. We believe that inconsistencies are merely expression of defects. That is, inconsistencies highlight situations under which defects are observable. However, a single defect in a software model may result in many inconsistencies and a single inconsistency may be the result of multiple defects. Inconsistencies may thus be related to other inconsistencies and we thus believe that during fixing, one should consider the clusters of such related inconsistencies. The main benefit of clustering inconsistencies is that it becomes easier to detect the defect the bigger the cluster. This paper discusses the idea in principle, provides some qualitative aspects of its benefit, and gives an outlook on how we plan to realize our vision.

**Categories and Subject Descriptors:** I.6.4 Simulation and Modeling; Model Validation and Analysis

**General Terms:** Algorithms, Human Factors, Verification.

**Keywords:** User Guidance, Grouping and Clustering, Inconsistencies

## 1. INTRODUCTION

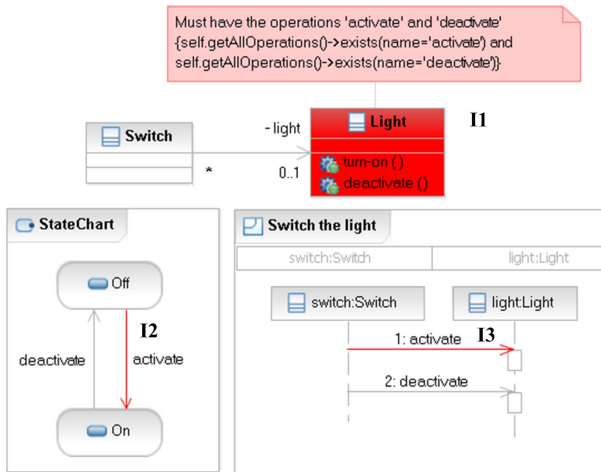
C. W. Johnson and C. Runciman already wrote in 1982 [6] “It is important to distinguish between an error diagnosis and error reporting. Correct error diagnosis must rely upon the programmer as it may depend upon intentions that are not expressed in his program. The compiler’s job is correct error reporting using a form and content of reports most likely to help the programmer in error diagnosis. We can compare error reports to the symptoms of a sick patient: the location at which the error is detected is not necessarily its source.”

This is analogous to the modeling world, where inconsistencies are the symptoms (rules and/or constraints that are violated) which are caused by defects (the sources of symptoms that need fixing) in the model. It is thus the job of the designer to identify the defects by exploring the choices for fixing the inconsistencies – one of these choices (if complete) for each inconsistency inevitably must involve fixing a defect. Thus, inconsistencies are the sheer symptoms

of a defect, but usually involve other model elements that when changed could also resolve the inconsistency. In accordance, fixing inconsistencies individually could mean fixing the symptoms only but not the defects (i. e., much like temperature-lowering medication merely “fixes” the symptom – the fever – but not the cause – an infection). Much like a good doctor attempts to identify all symptoms about a sickness to then hypothesize about the cause, a good software modeler should identify relationships among inconsistencies to reason about the cause(s) (defects) for these inconsistencies. Perhaps a key difference here: software models typically contain many defects.

A lot of research has been conducted to avoid and help detect and correct inconsistencies. The issue that inconsistencies are not self-contained is largely ignored in literature but of essential importance, it is far more important to fix the cause than just the symptoms. After all, the goal of engineers is not just to resolve one inconsistency at a time but in the end to get a consistent model. In order to get a consistent model, all defects have to be resolved. Of course some inconsistencies can only be resolved by fixing the underlying defects, but those defects often cause additional inconsistencies at other locations. In certain situations this even could be reversed, meaning that several defects cause the same inconsistency. An example for such a situation would be a requirement change in a already consistent model. This requirement change could require a set of model changes conflicting with the present requirements. As a consequence the first change would introduce an inconsistency without being the defect itself, instead the other model elements that are required to be changed are the defects. This also relates to the need of tolerating inconsistencies [1], since preventing them in this case would significantly change the typical work flow. So the challenge lies in determining where the defects are located and how to fix them, not just their symptoms, whilst not being too concerned with not causing new inconsistencies since they could be required to achieve the engineer’s goals.

We present our vision and proposed approach of how to exploit interrelations between inconsistencies for resolving them in this paper. It is our believe that using this information will result in fewer, concrete fixes and provide guidance to engineers for locating the defects. Additionally we address open issues and discuss steps that in our opinion have to be taken to provide some insights in the qualitative aspects of this approach. However we are ignoring techniques for including semantic analysis of constraints and generally information on how the inconsistencies were created for the



**Figure 1: Several Inconsistencies in an UML model of a Light and a Switch**

time being, at this point we just plan to investigate if inconsistencies are related and how this fact can be used to our advantage before combining it with other technologies.

This paper is structured as follows: In Section 2 we describe the scenario and problem we address. This is followed by the vision of how we want to tackle the problem in Section 3. In Section 5 we discuss the state-of-the-art and related work. In Section 4 we describe in detail how we plan to realize our vision. Finally we draw a conclusion and give an outlook to future work in Section 6.

## 2. SCENARIO AND PROBLEM

During modeling, engineers are prone to making mistakes. State-of-the-art tool support can help to detect these mistakes and point to inconsistencies in the model. For example, Figure 1 shows a simple UML model describing a `Light` with a `Switch` containing three inconsistencies:

- I1** A violated model constraint ( $C1$ ) that states that the `Light` class has to have at least two operations named `activate` and `deactivate`.
- I2** A violated meta-model constraint ( $C2$ ) that states that state-chart actions must be defined as an operation in the owner's class. In this case the owner of the state-chart class is `Light` and `Light` has no `activate` operation defined.
- I3** A violated meta-model constraint ( $C3$ ) that states that collaboration message actions must be defined as an operation in receiver's class. In this case the receiver's class is `Light` which has no `activate` operation defined.

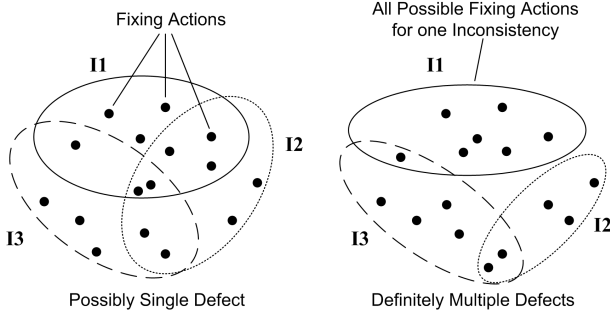
Typical tool support for fixing these inconsistencies will only look at each inconsistency individually and generate fixing actions for each of them [4, 5, 8, 7]. Current state of the art has produced interesting solutions for suggesting fixes to inconsistencies. There is the pioneering work of Nentwich et al. [8] that demonstrated how to generate fixing actions for inconsistencies (one inconsistency at a time). They distinguished abstract and concrete fixes where abstract fixes

in essence identified the locations where to fix (e.g., change the name of the `turn-on` method) and concrete actions in addition identify how to fix that location (e.g., change the name of the `turn-on` method to `activate`). However, the fixing of inconsistencies also has side effects onto other design constraints: negative side effects if the fixing causes new inconsistencies or positive side effects if the fixing of an inconsistency also fixes other inconsistencies. While the existence of these side effects has been widely published, to date they have not been exploited much. Existing state of the art, either attempts to minimize negative side effects (a heuristic that is not at all guaranteed to be the right strategy) or focuses only on visualizing them. If the goal is to avoid negative side effects (i.e., avoid additional inconsistencies) then possible fixes for the above model would be:

- I1** The first inconsistency could be fixed by adding the operation `activate` to the class `Light` ( $F1$ ), or changing its operation `turn-on` to `activate` ( $F2$ ). Of course the model constraint  $C1$  could be changed to fit the model ( $F3$ ).
- I2** The second inconsistency would also be fixed with fixes  $F1$  and  $F2$ . Additional fixes would be to change the inconsistent action in the state-chart to `turn-on` ( $F4$ ) or to `deactivate` ( $F5$ ), or simply remove it ( $F6$ ). And again also the meta-model constraint  $C2$  could be changed ( $F7$ ).
- I3** The third inconsistency again could be fixed with fixes  $F1$  and  $F2$ . Additional fixes would be to change the inconsistent message in the collaboration diagram to `turn-on` ( $F8$ ) or to `deactivate` ( $F9$ ), or simply remove it ( $F10$ ). And of course the meta-model constraint  $C3$  could be changed ( $F11$ ).

Looking at these inconsistencies individually results in several equally viable fixing actions. To choose one of those fixing actions for each individual inconsistency automatically is not reasonable as all are viable. Clearly, the decision on how to fix an individual inconsistency should be made by the software engineer. Since the example given is a fairly small and comprehensible model, it should be no challenge for an engineer to figure out that all inconsistencies can be resolved by either choosing fixing action  $F1$  or  $F2$ . In larger, more complex models, such "ideal" fixing actions cannot be identified manually as easily, and as a consequence automated support is needed. Nonetheless, even the "ideal" fixing actions with the least number of changes and/or the fewest negative side effects are not necessarily the fixes the software engineers intends. For example, if a model is fully consistent but a requirement change requires a set of model changes, then the first change likely causes inconsistencies because the set of model changes are not yet finished. The fix with the least number of changes and fewest negative side effects is then often a simple undo to restore the initial, correct state – a minimal, consistent solution, however, clearly an incorrect one with respect to the intended requirements change.

Even though this paper focuses on modeling with the UML, we previously also investigated the concept of tolerating conflicts in other domains. In [9] we discussed different fixing strategies in the domain of decision-making and product-line engineering especially. From our experiences in these domains, the same basic principles discussed in this paper are valid. If decision makers pick conflicting decisions, those



**Figure 2: Examples of different Interrelations between Inconsistencies**

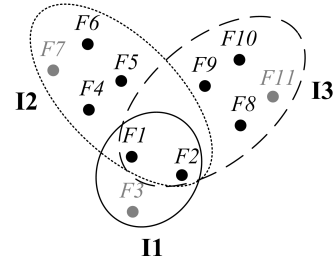
decisions are often involved in several related rule violations in the decision model (symptoms).

### 3. VISION

As mentioned in the introduction, fixing inconsistencies should not focus on fixing them individually because they are only the symptoms of defects. Software modelers should identify relationships among inconsistencies to reason about the cause(s) (defects) for these inconsistencies.

Looking at a group of inconsistencies instead of a single one provides more information about the defect and should prove to be useful in reasoning about possible fixes: 1) by reducing the number of possible fixes and thus more easily identifying the defect(s) at hand and 2) by understanding how many defects are involved and what combination of changes are necessary fix them. The latter aspect in particular is challenging because fixes for defects do not necessarily involve single changes to a model but may require sets of changes. While the set of changes is a subset of the changes of the individual inconsistencies, the combinatorial explosion in what combination of changes of the individual inconsistencies to consider would be hard to decide manually. A fixing action in this scenario thus involves all the changes to fix the defect(s) involved and thus all its/their inconsistencies. Depending on the number of inconsistencies that are investigated at a time, this could mean a significant scalability improvement when only searching for fixing actions with a number of changes considerably smaller than the number of inconsistencies. In addition, the search for fixes that involve single changes only, could be used to determine if inconsistencies are related and if one or more defects caused the inconsistencies under investigation respectively.

Figure 2 shows two sets of inconsistencies that are different in terms of their relationships. An ellipse represents the fixing actions of a single inconsistency which are depicted as black dots. Fixing actions that are located in the overlapping areas of the ellipses are fixing actions shared among several inconsistencies. These fixing actions could fix all inconsistencies involved – however, it is not necessarily true that there is a single concrete fix for every fixing action in case the fixing action is abstract. On the left hand side of Figure 2, a scenario with three inconsistencies and an overlap among all of them is shown. In this case, there exist two possible fixing actions to resolve all three inconsistencies (overlap among all three ellipses). This implies that those inconsistencies are related, however it does not necessarily imply that these fixes are indeed the only correct



**Figure 3: Overview of Fixes for the Example from Figure 1**

fixes the software engineer should consider. In other words the cause for those three inconsistencies is possibly a single defect in which case one of the two fixing actions must be taken. However, if multiple defects cause the inconsistencies then other combinations of fixing actions are also possible. On the right hand side a different scenario is shown. We can see that both **I1** and **I2** are related to **I3** but they are in no relation to each other. In this scenario it is safe to say that there are at least two defects as there exists no single fixing action that could resolve all three inconsistencies at hand.

In the example described in Section 2, all three described inconsistencies are related, they all can be fixed by either choosing fixing actions *F1* or *F2* as is apparent in Figure 3 (assuming the constraints are correct and therefore fixes *F3*, *F7*, and *F11* are irrelevant and grayed out). In this example, we even have the special situation that fixing **I1** in any case resolves all three inconsistencies. Additionally if the designer decides that there are multiple defects, the number of combined fixing actions is reduced by the fact that *F1* and *F2* respectively always have to be part of the solution. Furthermore, choosing *F2* to change the operation name from **turn-on** to **activate** in the class **Light**, excludes fixes *F4* and *F8* as they would require the operation **turn-on** to be present in the class **Light**.

To summarize we think that calculating fixes for more than one inconsistency at a time, especially if they are interrelated, is highly beneficial. Possible positive effects are:

1. A reduction in the number of possible fixing actions, through reasoning with more facts in the knowledge base (notice that knowledge about relationships among inconsistencies reduces the number of fixing actions).
2. As a consequence an increase in scalability since the calculation can be cut-off as soon as a combined solution is not achievable any more.
3. More precise fixing actions since the impact of the changes onto a larger amount of model elements is already considered.
4. Supporting designers by unburden them of having to know about interrelations when choosing a fixing action.

### 4. PROPOSED APPROACH

We propose to realize our vision stepwise. First of all we will use the choice generation technique described by Egyed et. al. [5] to better characterize fixes for individual inconsistencies (i. e., to compute concrete fixes for given abstract

fixes). As a second step, we will investigate overlaps among inconsistencies. Initially, we will require the software engineer to identify relationships among inconsistencies; however, we will also develop heuristics to help the engineer. Questions we plan to answer are:

- How often occur interrelated inconsistencies in real world examples?
- How many choices for fixing an inconsistency can be excluded considering these interrelations? How strong is this reduction?
- Can abstract fixes become constrained fixes (constrained fixes being abstract fixes with some restrictions, for example the location is known and some possibilities of how to fix the inconsistency at that location have been excluded) or constrained fixes concrete fixes respectively? If yes, how often does this happen?

If the above mentioned qualitative aspects prove to be useful, as a next step we plan to investigate how the choice generation can be improved and sped up by considering the interplay among related inconsistencies already during the choice generation. For that we will rely on concepts and algorithms from CSPs (Constrained Satisfaction Problems). We also think some sort of grouping of actions the user can take will be necessary to reduce the amount of information we have to deal with. This grouping for example could be according to the effect on other inconsistencies or consistency rules in general.

As a final step we are planning to automatically determine which inconsistencies are related. On the one hand for certain consistency rules it could easily be defined on the meta-level, on the other hand for situational and not so obvious relations an online determination could be necessary. Online meaning in this case that the determination would occur while engineers are using the modeling tool. Our basic idea for this task until now is:

1. Get all model elements of one inconsistency that are involved during the evaluation of the consistency rule.
2. Look for other inconsistencies that share model elements with the one under investigation and repeat this step for those inconsistencies.
3. Do a pairwise search for fixes with a cardinality of one. If there are none handle those inconsistencies as none-related. If there are fixes search for overlaps with other pairwise search results to form inconsistency clusters.

Additionally we want to incorporate the concept of trust into our reasoning, this concept was already described in [9]. The concept basically states that some pieces of information observed through user behavior can be trusted. Such pieces of information are not evident from the model itself. An example would be that design decisions that introduce an inconsistency and are not undone must be important to the user and therefore can be assumed to be correct. Of course this assumption could only hold if it would be apparent to the user that an inconsistency was just created through tool support like instant consistency checking as described in [3]. This sort of trust would for example benefit engineers in including new requirements into consistent models. As already mentioned the first change of a series of model

changes could cause several inconsistencies. Even with considering all those inconsistencies while searching for fixes the result will probably be an undo, since it is the simplest fix. However combined with trusting this first change the search would continue and hopefully come up with changes that are required by the requirements change anyway.

## 5. RELATED WORK

The problem of resolving inconsistencies has received considerable attention in the last two centuries. In this section we give a brief overview of work that has been done in this research area. On the one hand, in order to resolve inconsistencies, they have to be detected and tolerated. Almost 20 years ago, Balzer argued that inconsistencies should be detected and communicated to the developers; however, developers should not be hindered in continuing their work despite the presence of inconsistencies [1]. However at some point those inconsistencies have to be resolved, preferably with the support of automated techniques.

First off all, to resolve inconsistencies they have to be detected. However the knowledge if the whole model or a single constraint is consistent, is not enough to produce fixes. As Nentwich et. al. for example stated in [7], it is important that trace links from the inconsistency to the model element(s) in question exist. In their work they propose to use first-order logic to express consistency rules and are able to provide trace links between inconsistent elements. Performance also is an issue when checking for consistency and approaches like the incremental consistency checking approach by A. Egyed [3] addresses this issue.

For generating fixing or repair actions several approaches exist. On the one hand, Xiong et. al. propose writing additional “fixing procedures” for each constraint, in order to produce fixes when needed [11]. On the other hand Nentwich et. al. describe in their work [8] a method for generating interactive repairs from first order logic formulae - the same formulae that they already used to detect inconsistencies [7]. Another approach described by Egyed et. al. in their paper [5] shows how to generate choices for fixing an inconsistency without having to understand such formulae which can be complex in case consistency rules are written in programming languages. These approaches look at other model elements already defined in the model and use them as choices. This generated choices are then reduced by looking at the impact of each choice [4, 2] and removing those that would cause additional inconsistencies.

Despite the considerable progress on research for fixing inconsistencies, to the best of our knowledge no approach looks at more than one inconsistency at a time. However the need for a more “global” approach during consistency checking itself is demonstrated by Sabetzadeh et. al. in [10] but not used for fixing yet. Additionally Nentwich et. al. already stated in their work [8], that one of the biggest challenges is not to look at one single inconsistency but to look at inconsistencies from a more “global” point of view. This notion is also in accordance with our vision that a more “global” view should be beneficial for fixing inconsistencies.

## 6. CONCLUSIONS AND FUTURE WORK

In this work, we presented our vision of the next steps in inconsistency resolving, namely to not look at them individually but in clusters of related inconsistencies. We described

the potential we think this approach can have for resolving inconsistencies and hope that we can substantiate it in the near future. As a result we hope we can further improve the user guidance during modeling. Once we have evaluated and validated the qualitative properties of searching for fixes for several inconsistencies at once, we are planning to explore the scalability properties.

Open questions that also would be interesting to investigate are: Are there different relationships between inconsistencies? Can more conceptual parallels be found to the compiler community and used? To what degree can clustering algorithms be applied? From a user guidance point of view it would be interesting to investigate how important qualitative aspects can be utilized. For example if the user tells the system that certain inconsistencies are related, but no common fix can be found. As stated before this would imply that those inconsistencies are not related. How is this piece of information useful to the user and can it be used to guide the user to a satisfying solution?

## Acknowledgments

This research was funded by the Austrian FWF under agreement P21321-N15.

## 7. REFERENCES

- [1] R. Balzer. Tolerating Inconsistency. In *ICSE*, pages 158–165, 1991.
- [2] L. C. Briand, Y. Labiche, and L. O’Sullivan. Impact Analysis and Change Management of UML Models. In *ICSM*, pages 256–265. IEEE Computer Society, 2003.
- [3] A. Egyed. Instant consistency checking for the UML. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 381–390. ACM, 2006.
- [4] A. Egyed. Fixing Inconsistencies in UML Design Models. In *ICSE*, pages 292–301. IEEE Computer Society, 2007.
- [5] A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *ASE*, pages 99–108. IEEE, 2008.
- [6] C. W. Johnson and C. Runciman. Semantic Errors - Diagnosis and Repair. In *SIGPLAN Symposium on Compiler Construction*, pages 88–97, 1982.
- [7] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Trans. Internet Techn.*, 2(2):151–185, 2002.
- [8] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *ICSE*, pages 455–464. IEEE Computer Society, 2003.
- [9] A. Nöhler and A. Egyed. Conflict Resolution Strategies during Product Configuration. In D. Benavides, D. Batory, and P. Grünbacher, editors, *VaMoS*, volume 37 of *ICB Research Report*, pages 107–114. Universität Duisburg-Essen, 2010.
- [10] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik. Global consistency checking of distributed models with TReMer+. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 815–818. ACM, 2008.
- [11] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting Automatic Model Inconsistency Fixing. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 315–324. ACM, 2009.