# Inconsistency Detection in Distributed Model Driven Software Engineering Environments

Alix Mougenot*
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.

*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.

$$Visibility \iff \{\exists\ Me,\ N1 \cdot \\ lastAddProperty(Me, visibility, \text{“}public''\text{)} \wedge \\ lastAddReference(Me, namespace, N1) \wedge \\ lastAddProperty(N1, visibility, V)\ \wedge not(V = \text{“}public''\text{)}.$$

**Figure 3: Visibility rule for Praxis**

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

**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 | **true** |
| $SR_{OwnedElement}$ | **true** | false |
| $SR_{Class}$ | false | false |
| $SR_{Attribute}$ | false | false |
| $SR_{Namespace}$ | false | **true** |
| $Delete$ | 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 $SR_{Namesapce}$. 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 $SR_{OwnedElement}$. 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 $SR_{OwnedElement}$ 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 $SR_{OwnedElement}$ 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 to 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 [1] 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 assed. 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 relevent 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].

[1] http://meta.lip6.fr

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

# 7. REFERENCES


[1] X. Blanc, A. Mougenot, 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. Mougenot, 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. Peiin. 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. Mougenot, 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.