

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 link¹. 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

¹Please 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.

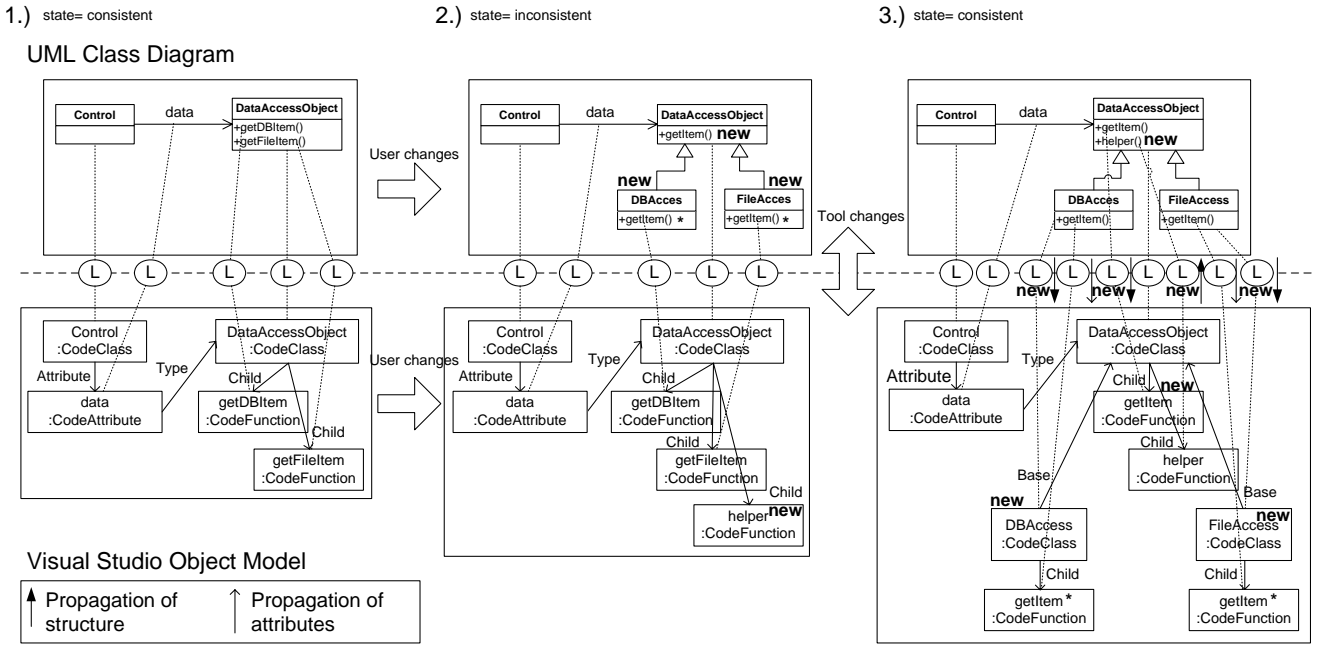


Figure 1: Recovering consistency between design and implementation

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

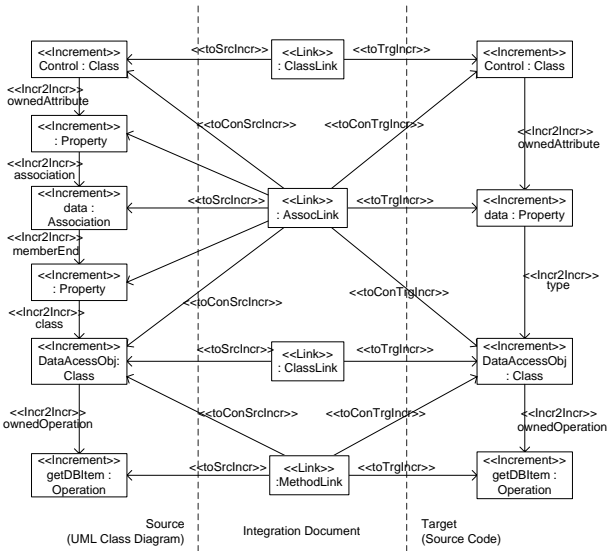


Figure 2: Triple graph as UML object diagram

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

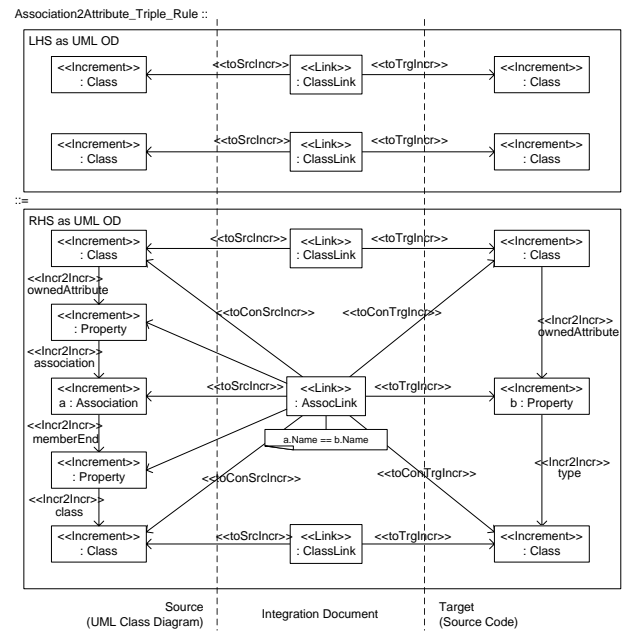


Figure 3: Triple rule (synchronous) creating an association *a* in a UML class diagram (left) and a corresponding attribute *b* with the same name in a source code (right)

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.

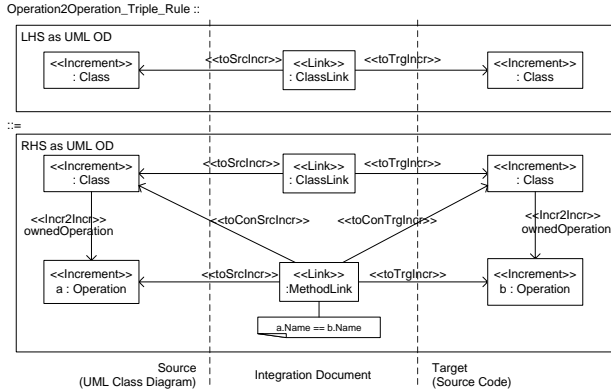


Figure 4: Triple rule (synchronous) creating an operation a in a UML class diagram (left) and a corresponding operation b with the same name in a source code (right)

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.

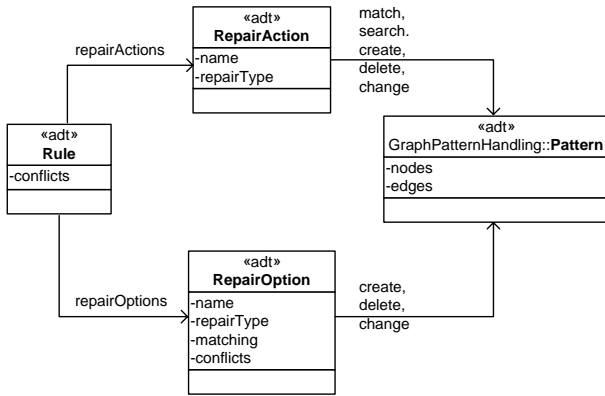


Figure 5: Data structure to store repair actions and options in the integration document

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 list² of already created repair actions `ras`, the pattern

²Please 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#)

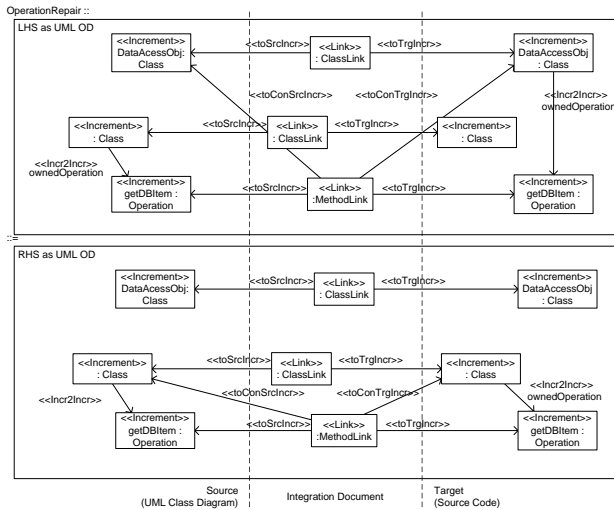


Figure 8: Repair action which reassigns the method to another class

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. *Integriatoren 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 29th 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 2nd 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 2nd 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 5th 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 Konsistenzsicherungswerkzeugen 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 11th 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 25th 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 20th 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 6th 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 18th 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].