Graph-Transformation Based Support for Model Evolution

Tom Mens[1]

Software Engineering Lab, Université de Mons-Hainaut
Av. du champ de Mars 6, 7000 Mons, Belgium
tom.mens@umh.ac.be

Abstract. During model-driven software development, we are inevitably confronted with design models that contain a wide variety of design defects. Interactive tool support for improving the model quality by resolving these defects in an automated way is therefore indispensable. In this paper, we report on the development of such a tool, based on the underlying formalism of graph transformation. Due to the fact that the tool is developed as a front-end of the *AGG Engine*, a general purpose graph transformation engine, it can exploit some of its interesting built-in mechanisms such as critical pair analyis and the ability to reason about sequential dependencies. We explore how this can help to improve the process of quality improvement, and we compare our work with related research.

About this document *At the end of each section, optional exercises are added to enable the reader to gain a deeper understanding of the underlying mechanisms. For some of the exercises, the AGG GUI will need to be used. It can be downloaded from the website http://tfs.cs.tu-berlin.de/agg/. Other exercices may require the use of the SIRP tool that is reported on in this paper. It can be downloaded from the SegraVis school website. Additional reading material related to this paper can be found in [1–3].*

1 Introduction

During development and evolution of design models it is often desirable to tolerate inconsistencies in design models. Indeed, such inconsistencies are inevitable for many reasons: (i) in a distributed and collaborative development setting, different models may be developed in parallel by different persons; (ii) the interdependencies between models may be poorly understood; (iii) the requirements may be unclear or ambiguous at an early design stage; (iv) the models may be incomplete because some essential information may be deliberately left out, in order to avoid premature design decisions; (v) the models are continuously subject to evolution; (vi) the semantics of the modeling language itself may be poorly specified.

All of these reasons hold in the case of UML, the de-facto general-purpose modelling language [4]. Therefore, current UML modeling tools should provide better support for resolving these inconsistencies in an automated way. Other types of design defects may also affect the quality of a model. Therefore, we suggest an automated approach to detect and resolve, among others, the following types of defects:

- nonconformance to standards (both industry-wide and company-specific standards);
- breaches of conventions (e.g., naming conventions);
- incomplete models, that are only partially specified and still have missing items [5];
- syntactic inconsistencies, i.e., models that do not respect the syntax of the modeling language;
- semantic inconsistencies, i.e., models that are not well-formed with respect to the semantics of the modeling language[1];
- design smells (in analogy with "bad smells") that indicate opportunities for performing a model refactoring;
- redundancies (e.g., double occurrences of a model element with the same name);
- visual problems (e.g., overlapping model elements in a diagram);
- bad practices;
- antipatterns, i.e., misuses or violations of design patterns

In addition, these problems may either be localised in a single UML diagram, or may be caused by mismatches between different UML diagrams.

The goal is therefore to provide a general framework and associated tool support to detect and resolve such design defects. In this paper, we suggest a transformation-based approach to do this. More in particular, we propose to use graph transformation technology. We report on an experiment that we have carried out and a tool that we have developed to achieve this goal, and we discuss how our approach may be integrated into comtemporary modeling tools.

Exercise *For each type of defect mentioned above, try to find concrete examples in the context of UML. Do this for various types of UML diagrams. Discuss your results with your fellow students.*

Exercise *Do you think there are other types of defects that do not fit into the list above? If yes, explain why and give a concrete example.*

2 Suggested approach

In Figure 1 we explain the iterative process of gradually improving the quality of a design model in an iterative way. First, defects in the model are identified. As explained above, these defects can be of diverse nature. Next, resolutions are proposed, selected and applied. The user may also wish to ignore or disable certain types of defects or resolutions. This process continues until all problems are resolved or until the user is satisfied.

[1] In UML this is a common problem due to the lack of a formal semantics combined with the fact that some parts of the semantics are deliberately left open, which makes the models subject to interpretation.

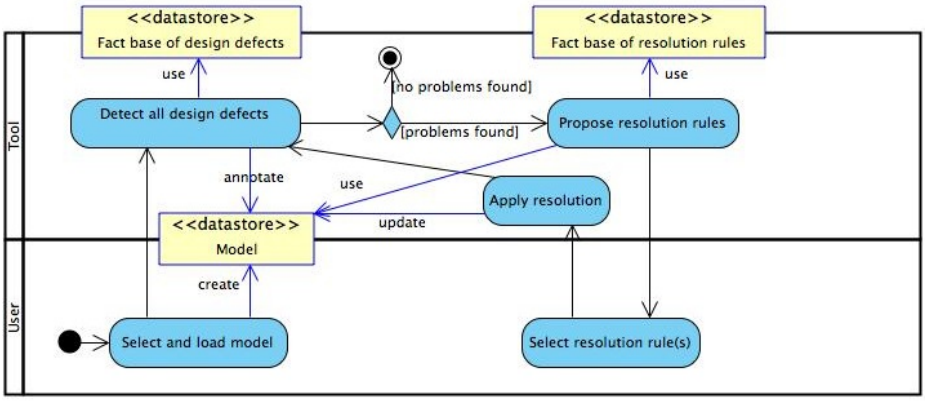

Fig. 1. UML activity diagram showing the interactive process for detecting and resolving design defects in a model.

When trying to develop tool support for this process, it is important to decide how the design defects and their resolutions should be specified. We opted for a formal specification, because this gives us an important added value: it allows us to analyse and detect mutual conflicts and sequential dependencies between resolution rules, which can be exploited to optimise the resolution process.

The particular formalism that we have chosen is the theory of graph transformation [6, 7]. The main idea that will enable us to perform conflict and dependency analysis is the application of theoretical results about critical pairs [8], which allow us to reason about parallel and sequential dependencies between rules.

Exercise *Extend the activity diagram of Figure 1 to take into account the following activities:*

- *Undoing a resolution after it has been applied;*
- *Ignoring certain detected design defects, such that no resolution will be proposed for them;*
- *Disabling certain proposals for resolution that the user considers to be inappropriate;*
- *Optionally modify the model manually after a resolution step (in order to perform manual changes that are required but that have not been taken into account by the resolution rule.*

3 Graph transformation

To perform detection and resolution of model defects, the tool relies entirely on the underlying formalism of graph transformation.

The UML metamodel is represented by a so-called *type graph*. A simplified version of the metamodel, showing a subset of UML 2.0 class diagrams, statemachine diagrams and sequence diagrams, is given in Figure 2. The notion of design defect is incorporated

explicitly in this type graph by the node type `Conflict`, which is used to identify model defects.

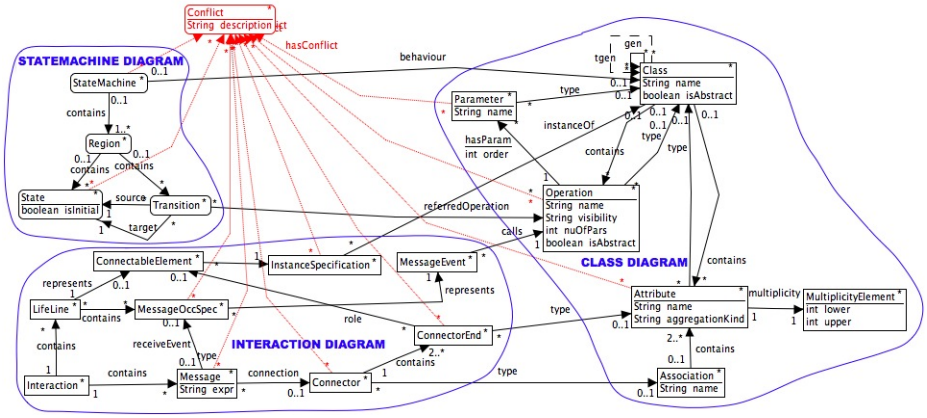

Fig. 2. Simplified metamodel for UML class diagrams, state machine diagrams, expressed as a type graph with edge multiplicities in AGG. In addition, a node type `Conflict` is introduced to represent model defects.

Every UML model will be represented internally as a *graph* that satisfies the constraints imposed by the aforementioned *type graph*. Figure 3 shows a simple example of a UML class diagram, represented as a graph model. More precisely, it corresponds to a directed, typed, attributed graph. These graph representations can be generated automatically from the corresponding UML model without any loss of information. [2]

Detection of design defects will be achieved by means of graph transformation rules. For each particular defect, a graph transformation rule will be specified that detects the defect. This is realised by searching for the occurrence of certain graph structures in the model, as well as the *absence* of certain forbidden structures (so-called *negative* application conditions or NACs).

A simple example of a detection rule is given in Figure 4. It detects the so-called *Dangling Type Reference* defect. This occurs when an `Operation` contains `Parameters` whose `type` has not (yet) been specified. The specification of this rule as a graph transformation is composed of three parts. The middle pane represents the *left-hand side (LHS)* of the rule, which is basically the occurrence of some `Operation` having a `Parameter`. The leftmost pane represents a *negative application condition (NAC)*, expressing the fact that the `Parameter` of interest does not have an associated `type`. Finally, the rightmost pane represents the *right-hand side (RHS)* of the rule, showing the result after the transformation. In this case, the only modification is the introduction of a `Conflict` node that is linked to the `Parameter` to show that there is a potential design defect.

[2] An experiment along these lines has been carried out by Laurent Scolas as a student project.

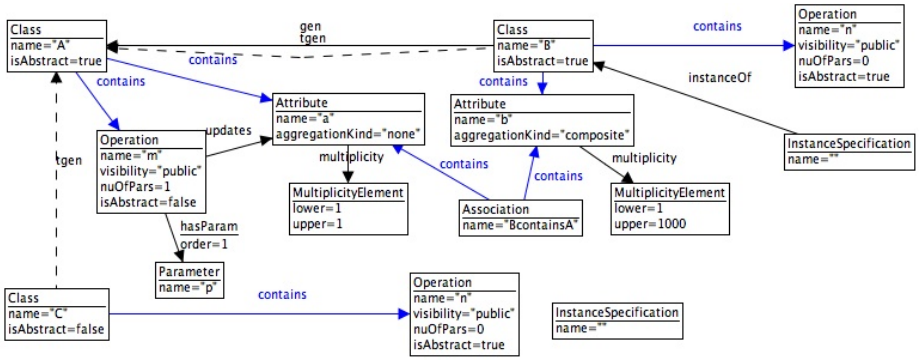

Fig. 3. Simplified UML class diagram model represented as a directed, typed, attributed graph in AGG.

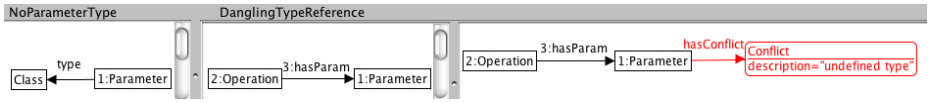

Fig. 4. Graph transformation representing the detection of a design defect of type *Dangling Type Reference*.

Given a source model, we can apply all detection rules in sequence to detect all possible design defects. By construction, the detection rules are parallel independent, i.e., the application of a detection rule has no unexpected side effects on other detection rules. This is because the only thing a detection rule does is introducing in the RHS a new node of type `Conflict` and a new edge of type `hasConflict` pointing to this node. Morever, the LHS and NAC of a detection rule never contain any `Conflict` nodes and `hasConflict` edges.

If we apply all detection rules (only one of these has been shown in Figure 4) to the graph of Figure 3, this graph will be annotated with nodes of type `Conflict`, as shown in Figure 5, to represent all detected design defects. The type of defect is indicated in the `description` attribute of each `Conflict` node. Observe that the same type of conflict may occur more than once at different locations, and that the same model element may be annotated by different types of conflicts.

Graph transformations will also be used to resolve previously detected design defects. For each type of design defect, several resolution rules can be specified. Each resolution rule has the same general form. On the left-hand side, we always find a `Conflict` node that indicates the particular inconsistency that needs to be resolved. On the right-hand side, this `Conflict` node will no longer be present because the rule removes the design defect.

Figure 6 proposes three resolution rules for the *Dangling Type Reference* defect mentioned previously. The first one removes the problematic parameter, the second one

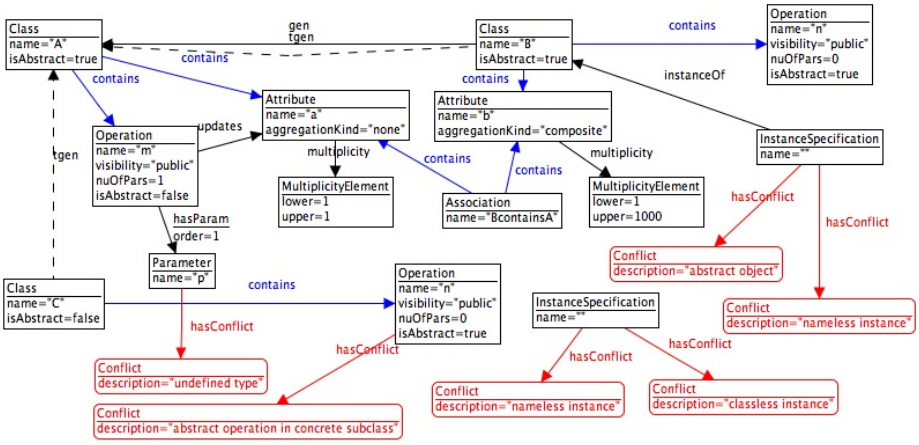

Fig. 5. Same UML class diagram model as in Figure 3, but annotated with all detected design defects.

uses an existing class as type of the parameter, and the third one introduces a new class as type of the parameter.

Exercise *Compare the UML 2.0 metamodel for class diagrams, statemachines and sequence diagrams with the type graph of Figure 2. Discuss the differences. Try to extend the type graph to make it more complete and more precise. Are there certain things in the UML metamodel that cannot be expressed in the type graph? Why?*

Exercise *Take a given UML class diagram, statemachine and sequence diagram, and represent it as a graph (such as the one in Figure 3) conforming to the type graph of Figure 2.*

Exercise *Specify graph transformation rules in AGG to detect different types of design defects (cf. Figure 4 for the* Dangling Type Reference *defect). Verify in AGG whether the rule does what it is supposed to do.*

Exercise *Propose one or more resolution rules for the deign defects for which you have specified a rule in the previous exercise (cf. Figure 6 for the* Dangling Type Reference *resolution rules). Verify with AGG whether these resolution rules do what they are supposed to do.*

4 Tool support

The tool that we have selected to perform our experiments is *AGG*[3] (version 1.4), a state-of-the-art general purpose graph transformation tool [9]. We rely on the AGG engine as a back-end, and we have developed a dedicated user interface on top of it to

[3] See `http://tfs.cs.tu-berlin.de/agg/`

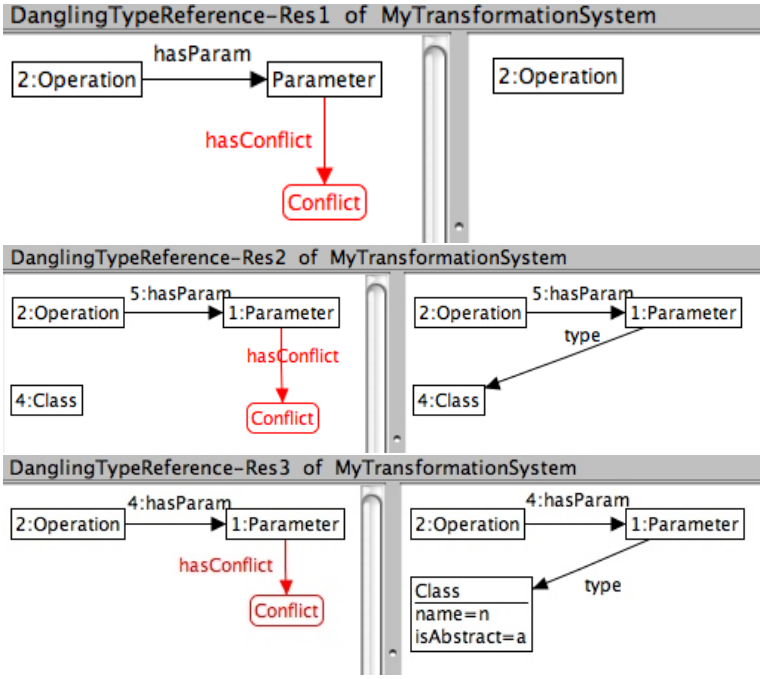

Fig. 6. Three graph transformations specifying alternative resolution rules for the *Dangling Type Reference* defect.

enable the user to interactively detect and resolve design defects [10]. This tool is called SIRP, for Simple Interactive Resolution Process. As will explained in more detail in the discussion section, SIRP has not (yet) been integrated into a UML modeling tool for various reasons.

Figure 7 shows a screenshot of the SIRP tool in action. It displays the detected design defects of Figure 3 as well as the resolution rules proposed to resolve these defects. In the screenshot, we see three resolution rules that can be selected to resolve the occurrence of the *Dangling Type Reference* defect. After selecting one of these rules, we can apply the chosen resolution, after which the model will be updated and the list of remaining design defects will be recomputed.

According to the resolution process of Figure 1, each resolution step is followed by a redetection phase. Currently, during redetection, we follow a brute-force approach, and detect all design defects again from scratch. A more optimal approach would be to come to an *incremental* redetection algorithm, thereby remembering those design defects that have already been identified in a previous phase. However, when doing this, we need to deal with a number of situations that may occur due to side effects that may impact existing model defects:

– **Orphan defects** arise when certain model elements have been removed as a result of resolving a certain design defect. In that case, some `Conflict` nodes may

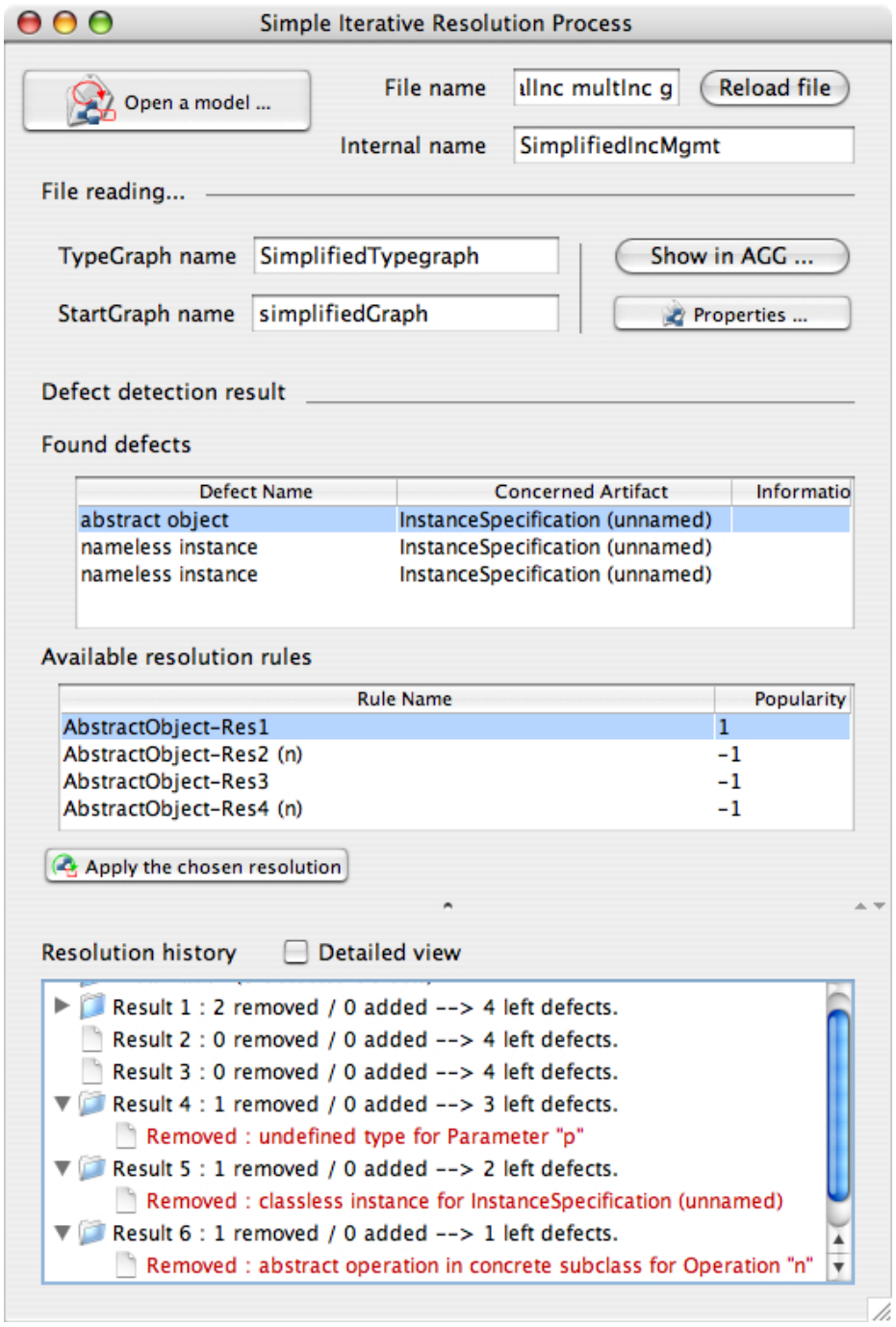

Fig. 7. Screenshot of the SIRP tool in action. Several defects have been resolved already, as shown in the resolution history. Resolution rules are proposed for each remaining defect with a certain popularity (based on whether the rule has already been applied before by the user). Selected rules can be applied to resolve the selected defect.

remain in the graph without any model element to which they refer (because the model element has been removed).
- **Expired defects** arise if the resolution of a certain design defect also resolves other design defects as a side-effect. If this is the case, there will be a `Conflict` node that points to some model element, even though the defect has already been resolved.

To address these two problems, we need to provide so-called *cleanup rules*, that remove all `Conflict` nodes that are no longer necessary. Such *cleanup rules* can be generated automatically from the detection and resolution rules.

Exercise *Give an example of a concrete situation of a resolution rule that gives rise to an orphan defect.*

Exercise *Give an example of a concrete situation of a resolution rule that gives rise to an expired defect.*

Exercise *What would a cleanup rule look like? Try to specify it as a graph transformation rule in AGG.*

Exercise *Use the SIRP tool to verify whether the detection and resolution rules that you have specified in previous exercises actually behave as they should.*

Exercise *Suggest ways in which the SIRP tool could be improved.*

5 Graph transformation dependency analysis

There are also other types of problems that may inevitably occur during the detection and resolution process, due to the inherently incremental and iterative nature of the conflict resolution process.

Induced defects may appear when the resolution of a certain design defect introduces other design defects as a side effect. An example is given in Figure 8. Suppose that we have a model that contains a defect of type *Abstract Object*, i.e., an instance specification (an object) that refers to an abstract class (labelled 1 in Figure 8). The resolution rule *AbstractObject-Res1* resolves the defect by setting the attribute `isAbstract` of class 1 to `false`. As a result of this resolution, the design defect called *Abstract Operation* suddenly becomes applicable. This is the case if class 1, which now has become concrete, contains one or more abstract operations.

Conflicting resolutions may appear when there are multiple design defects in a model, each having their own set of applicable resolution rules. It may be the case that applying a resolution rule for one design defect, may invalidate another resolution rule for another design defect. As an example, consider Figure 9. The left pane depicts a situation where two defects occur, of type *Abstract Object* and *Abstract Operation* respectively, but attached to different model elements. The resolution rules *AbstractObject-Res1* and *AbstractOperation-Res4* for these defects (shown on the right of Figure 9) are conflicting, since the first resolution rule sets the attribute `isAbstract` of class 1 to `false`, whereas the second resolution rule requires as a precondition that its value should be `true`.

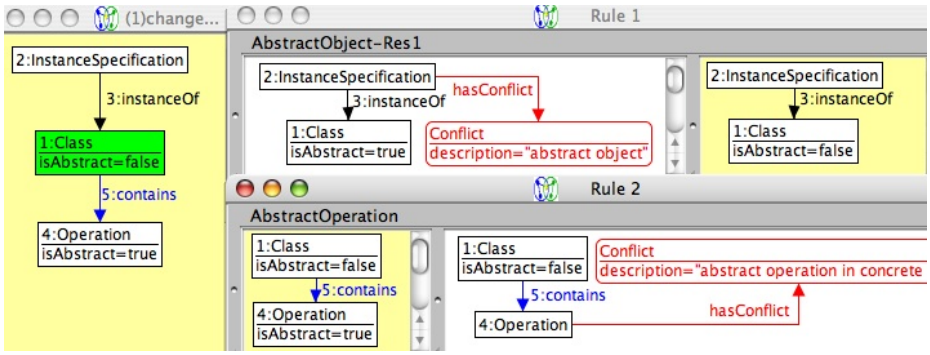

Fig. 8. *Induced defects*: Example of a sequential (causal) dependency of detection rule **AbstractOperation** on resolution rule **AbstractObject-Res1**.

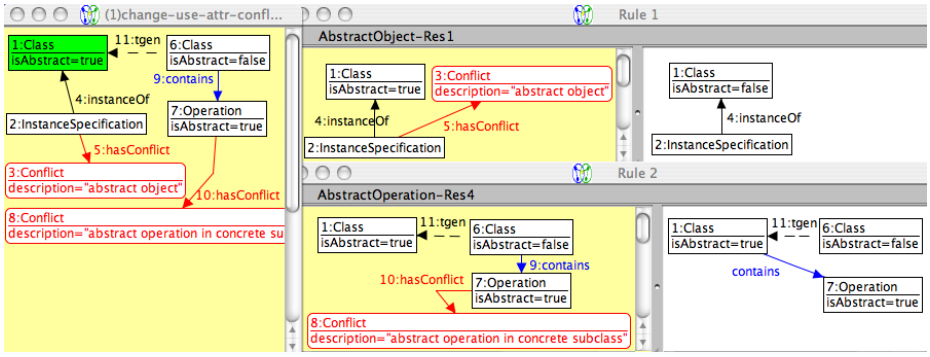

Fig. 9. *Conflicting resolutions*: Example of a critical pair illustrating a mutual exclusion between resolution rules **AbstractObject-Res1** and **AbstractOperation-Res4**.

To identify and analyse the two situations explained above in an automated way, we need to make use of the mechanism of *critical pair analysis* of graph transformation rules [8, 11]. The goal of critical pair analysis is to compute all potential mutual exclusions and sequential dependencies for a given set of transformation rules by pairwise comparison. Such analysis is directly supported by the *AGG* engine, so it can readily be used in our approach.

The problem of **induced defects** is a typical situation of a *sequential dependency*: a detection rule causally depends on a previously applied resolution rule. Figure 10 shows an example of a dependency graph that has been generated by AGG. Given a selection of design defects, it shows all **induced defects**, i.e., all detection rules that sequentially depend on a resolution rule. This information is quite important in an incremental resolution process, as it informs us, for a given resolution rule, which types of defects will need to be redetected afterwards.

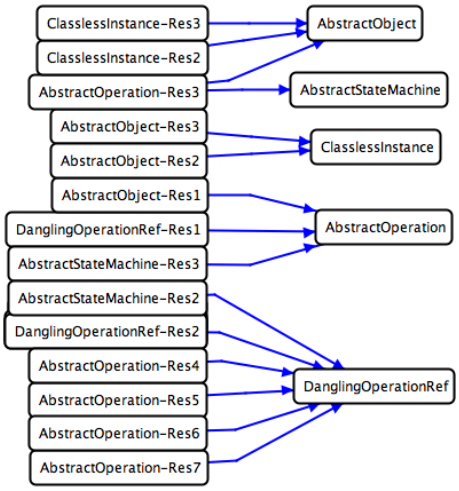

Fig. 10. Dependency graph generated by AGG showing all **induced defects**, i.e., all defect detection rules that sequentially depend on a resolution rule.

The problem of **conflicting resolutions** is a typical situation of a *parallel conflict*: two rules that can be applied in parallel cannot be applied one after the other (i.e., they are mutually exclusive) because application of the first rule prevents subsequent application of the second one. Figure 11 shows an example of a conflict graph that shows all possible **conflicting resolutions** (for a given selection of design defects). Except for some layout issues, this graph has been automatically generated by AGG's critical pair analysis algorithm. Again, the information reported in the graph is quite important during an internative resolution process, as it informs the user about which resolution rules are mutually exclusive and, hence, cannot be applied together.

Fig. 11. Conflict graph generated by AGG showing all **conflicting resolutions**, i.e., mutual exclusions between resolution rules for distinct design defects. In order not to clutter the figure, mutual exclusions between different resolution rules of the same defect have been omitted.

Exercise *Find other examples of induced defects, and check whether AGG's critical pair analysis algorithm (which can be found via AGG's "Analyzer" menu) actually detects this situation as a sequential dependency.*

Exercise *Find other examples of conflicting situations, and check whether AGG's critical pair analysis algorithm actually detects this situation as a parallel conflict.*

Exercise *Use AGG's critical pair analysis to compute a graph similar to the one of Figure 10 for the defect detection and resolution rules that you have specified in previous exercices. To reduce computing time, do this in two steps. First, disable all detection rules except for the ones that you would like to analyse, and then compute all sequential dependencies of these detection rules on all resolution rules. Second, disable all resolution rules, except for the ones you would like to analyse, and then compute all sequential dependencies of any detection rule on these resolution rules.*

6 Cycle detection and analysis

As illustrated in Figure 12, starting from the dependency graph, we can also compute possible *cycles* in the conflict resolution process. This may give important information to the user (or to an automated tool) to avoid repeatedly applying a certain combination of resolution rules over and over again. Clearly, such cycles should be avoided, in order to optimise the resolution process (e.g., by preventing cycles to occur).

As an example of such cycle, consider Figure 12, which represents a carefully selected subset of sequential dependencies that have been computed by AGG.[4] In this figure, we observe the presence of multiple cycles of various lengths, all of them involving the *Abstract Operation* defect.

Let us start by analysing the cycles of length 4, that correspond to two successive detection and resolution steps that give rise to a cycle. The cycle corresponding to re-

[4] To interpret the dependency graph, the blue lines could be read as "enables" or "triggers".

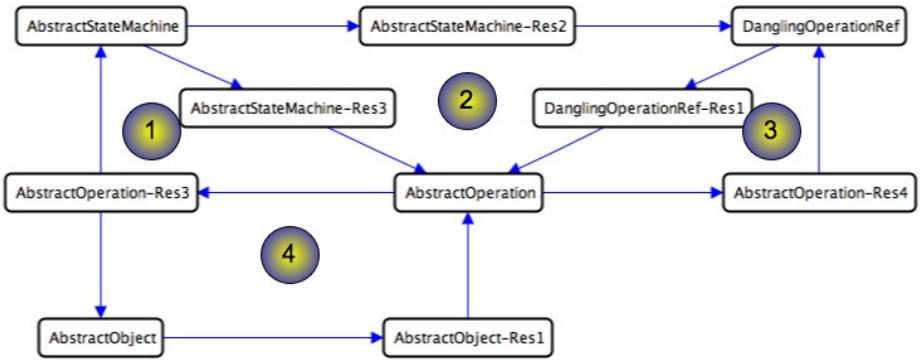

Fig. 12. Some examples of detected cycles in the sequential dependency graph.

gion 1 shows that we can repeatedly apply resolution rules *AbstractStateMachine-Res3* and *AbstractOperation-Res3 ad infinitum*. This is the case because the two resolution rules are each others inverse. Therefore, after applying one of both rules, the interactive resolution tool should not propose the other rule because it would undo the effect of the first one.The cycle corresponding to region 4 is similar to the previous one, except that it occurs between resolution rules *AbstractObject-Res1* and *AbstractOperation-Res3*.

There is also a cycle of length 6, corresponding to an succession of three detection and resolution rules. The cycle is described by the boundaries of the region composed by 1 and 2, and occurs when we apply resolution rules *AbstractStateMachine-Res2*, *DanglingOperationRef-Res1* and *AbstractOperation-Res3* in sequence.

A cycle of length 8 is described by the boundaries of the region composed of 1, 2 and 4, and corresponds to a succession of 4 resolution rules, namely *AbstractStateMachine-Res2*, *DanglingOperationRef-Res1*, *AbstractObject-Res1* and *AbstractOperation-Res3*.

Because the sequential dependency graph can be very large, manual detection of cycles is unfeasible in practice. Therefore, we have used a small yet intuitive user interface for detecting all possible cycles in a flexible and interactive way, based on the output generated by AGG's critical pair analysis algorithm. This program has been developed by Stéphane Goffinet in the course of a student project.

Exercise *Find out whether the detection and resolution rules that you have added give rise to any cycles, and analyse these cycles.*

7 Discussion and Future Research

Currently, our approach has not yet been integrated into a modeling tool. The reason is that there are many mechanisms for doing this, and we haven't decided yet on which alternative is the most appropriate. The most obvious solution would be to directly integrate the proposed process into an existing UML modeling tool. *ArgoUML*[5] seems the

[5] http://argouml.tigris.org/

most obvious candidate for doing this because it is open source and already provides support for design critics. It is not clear, however, how this can be combined easily with critical pair analysis since this requires an underlying representation based on graph transformation. Therefore, another more feasible approach could be to develop a modeling tool directly based on graph transformation as an underlying representation. Several such tools have already been proposed (e.g. VIATRA, GReAT, Fujaba), but none of those currently provides support for critical pair analysis. Another alternative could therefore be to build a modeling environment on top of the AGG engine. To achieve this, one may rely on the Tiger project, an initiative to generate editors of visual models using the underlying graph transformation engine [12].

The fact that the resolution of one model defect may introduce other defects is a clear sign of the fact that defect resolution is a truly iterative and interactive process. One of the challenges is to find out whether the resolution process will ever *terminate*. It is easy to find situations that never terminate (cf. the presence of cycles in the dependency graph). Therefore, the challenge is to find out under which criteria a given set of resolution rules (for a given set of design defects and a given start graph) will terminate. Recent work that explores such termination criteria for model transformation based on the graph transformation formalism has been presented in [13].

Another challenge is to try and come up with an *optimal order* of resolution rules. For example, one strategy could be to follow a so-called "opportunistic resolution process", by always following the choice that corresponds to the least cognitive effort (i.e., the cognitive distance between the model before and after resolution should be as small as possible). How to translate this into more formal terms remains an open question. A second heuristic could be to avoid as much as possible resolution rules that give rise to *induced defects* (i.e., resolutions that inadvertently introduce other defects). Yet another strategy could be to prefer resolution rules that give rise to *expired defects* (since these are rules that resolve more than one inconsistency at once).

Another important question pertains to the *completeness* of results. How can we ensure that the tool detects all possible defects, that it proposes all possible resolution rules, and that all conflicting resolutions and sequential dependencies are correctly reported? How can we avoid false positives reported by the tool?

A related question concerns *minimality*. Is it possible to detect and avoid redundancy between detection rules and between resolution rules? Is it possible to come up with a minimal set of resolution rules that still cover all cases for a given set of detection rules?

A limitation of the current approach that we are well aware of, is the fact that not all kinds of model inconsistencies and resolution rules can be expressed easily as graph transformation rules. For example, behavioural inconsistencies are also difficult to express in a graph-based way. Because of this, our tool has been developed in an extensible way, to make it easier to plug-in alternative mechanisms for detecting defects, such as those based on the formalism of description logics [14, 15]. Of course, it remains to be seen how this formalism can be combined with the formalism of graph transformation, so that we can still benefit from the technique of critical pair analysis.

Exercise *If you have experience with a modeling tool or a graph transformation tool, discuss how the approach presented in this paper may be integrated into (or reimplemented in) that particular tool.*

Exercise *Do you see any other suggestions for improvement or future research? If yes, discuss them.*

8 Related Work

Critiquing systems originate in research on artificial intelligence, and more in particular knowledge-based systems and expert systems. Rather than giving a detailed account of such systems, let us take a look at one particular attempt to incorporate these ideas into a modeling tool, with the explicit aim to critic and improve design models [16, 17]. In this view, "a *design critic* is an intelligent user interface mechanism embedded in a design tool that analyses a design in the context of decision-making and provides feedback to help the designer improve the design. Support for design critics has been integrated into the ArgoUML modeling tool. It is an automated and unobtrusive user interface feature that checks in the background for potential design anomalies. The user can chose to ignore or correct these anomalies at any time. Most critiquing systems follow the so-called ADAIR process which is sequentially composed of five phases: Activate, Detect, Advice, Improve and Record. Without going into details, our approach roughly follows the same process.

Another approach that is very related to ours is reported in [18]. A rule-based approach is proposed to detect and resolve inconsistencies in UML models, using the Java Rule Engine JESS. In contrast to our approach, where the rules are graph-based, the specification of their rules is logic-based. However, because the architecture of their tool provides a Rule Engine Abstraction Layer, it should in principle be possible to replace their rule engine by a graph-based one. Other logic-based approaches to inconsistency management and resolution have been proposed in [15, 19].

The main novelty of our approach compared to the previously mentioned ones, is the use of the mechanism of critical pair analysis to detect mutual inconsistencies between rules that can be applied in parallel, as well as sequential dependency analysis between resolution rules.

There have been several attempts to use graph transformation in the context of inconsistency management. In [20], distributed graph transformation is used to deal with inconsistencies in requirements engineering. In [21], graph transformations are used to specify inconsistency detection rules. In [22] repair actions are also specified as graph transformation rules. Again, the added value of our approach is the ability to analyse conflicts and dependencies between detection and resolution rules.

The technique of critical pair analysis of graph transformations has also been used in other, related, domains. [23] used it to detect conflicting functional requirements in UML models composed of use case diagrams, activity diagrams and collaboration diagrams. [3] used it to detect conflicts and dependencies between software refactorings. [24] used it to improve parsing of visual languages.

An important aspect of research on model quality that is still underrepresented in literature is empirical research and case studies on the types of defects that commonly occur in industrial practice and how these can be resolved [5, 25, 26].

Exercise *Is there other relevant related work that has not been explained above? If yes, discuss it.*

9 Conclusion

In this article we addressed the problem of model quality improvement. The quality of a model can be improved in an iterative way by looking for design defects, and by proposing resolution rules to remove these defects. Interactive tool support for this process can benefit from a formal foundation. This article proposed a tool based on the underlying formalism of graph transformation. Given a formal specification of the UML model as a graph (and the metamodel as a type graph), design defects and their resolutions were specified as graph transformation rules. Furthermore, critical pair analysis was used to identify and analyse unexpected interactions between resolution rules, new defects that are introduced after resolving existing defects, and cycles in the resolution process. Futher work is needed to integrate this tool ino a modeling environment.

Exercise *Try to reformulate the conclusion based on what you think is the main contribution and interest of this work.*

References

1. Mens, T., Van Der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proceedings MoDELS/UML 2006. Volume 4199 of Lecture Notes in Computer Science., Springer-Verlag (2006) 200–0214
2. Mens, T.: On the use of graph transformations for model refactoring. In Ralf Lämmel, Joao Saraiva, J.V., ed.: Generative and transformational techniques in software engineering. Volume 4143 of Lecture Notes in Computer Science., Springer (2006) 215–254
3. Mens, T., Taentzer, G., Runge, O.: Analyzing refactoring dependencies using graph transformation. Software and Systems Modeling (2006) To appear.
4. Object Management Group: Unified Modeling Language 2.0 Superstructure Specification. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf (2005)
5. Lange, C.F., Chaudron, M.R.: An empirical assessment of completeness in uml designs. In: Proc. Int'l Conf. Empirical Assessment in Software Engineering. (2004) 111–121
6. Rozenberg, G., ed.: Handbook of graph grammars and computing by graph transformation: Foundations. Volume 1. World Scientific (1997)
7. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools. Volume 2. World Scientific (1999)
8. Plump, D.: Hypergraph rewriting: Critical pairs and undecidability of confluence. In: Term Graph Rewriting. Wiley (1993) 201–214
9. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Proc. AGTIVE 2003. Volume 3062 of Lecture Notes in Computer Science., Springer-Verlag (2004) 446–453
10. Warny, J.F.: Détection et résolution des incohérences des modèles uml avec un outil de transformation de graphes. Master's thesis, Université de Mons-Hainaut, Belgium (2006)
11. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Proc. Int'l Conf. Graph Transformation. Volume 3256 of Lecture Notes in Computer Science., Springer-Verlag (2004) 161–177
12. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plugins. In: Proc. Int'l Conf. Automated Software Engineering, ACM Press (2005) 134–143

13. Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. In: Proc. Fundamental Aspects of Software Enginering (FASE). Volume 3442 of Lecture Notes in Computer Science., Springer-Verlag (2005) 49–63
14. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) 326–340
15. Van Der Straeten, R.: Inconsistency Management in Model-driven Engineering. An Approach using Description Logics. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium (2005)
16. Robbins, J.E., Redmiles, D.F.: Software architecture critics in the argo design environment. Knowledge-Based Systems **11** (1998) 47–60
17. Robbins, J.E.: Design Critiquing Systems. PhD thesis, University of California, Irvine (1999) Technical Report UCI-98-41.
18. Liu, W., Easterbrook, S., Mylopoulos, J.: Rule-based detection of inconsistency in UML models. In: Proc. UML 2002 Workshop on Consistency Problems in UML-based Software Development, Blekinge Insitute of Technology (2002) 106–123
19. Straeten, R.V.D., D'Hondt, M.: Model refactorings through rule-based inconsistency resolution. In: Proc. Symposium on Applied computing (SAC), ACM Press (2006) 1210–1217
20. Goedicke, M., Meyer, T., , Taentzer, G.: Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In: Proc. Requirements Engineering 1999, IEEE Computer Society (1999) 92–99
21. Ehrig, H., Tsioalikis, A.: Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In: ETAPS 2000 workshop on graph transformation systems. (2000) 77–86
22. Hausmann, J.H., Heckel, R., Sauer, S.: Extended model relations with graphical consistency conditions. In: Proc. UML 2002 Workshop on Consistency Problems in UML-Based Software Development. (2002) 61–74
23. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach. In: Proc. Int'l Conf. Software Engineering, ACM Press (2002)
24. Bottoni, P., Taentzer, G., Schürr, A.: Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In: Proc. IEEE Symp. Visual Languages. (2000)
25. Lange, C., Chaudron, M., Muskens, J.: In practice: Uml software architecture and design description. IEEE Software **23** (2006) 40–46
26. Lange, C.F., Chaudron, M.R.: Effects of defects in uml models – an experimental investigation. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM Press (2006) 401–410