On the Use of Graph Transformations for Model Refactoring

Tom Mens[1]

Service de Génie Logiciel
Université de Mons-Hainaut, Belgium
tom.mens@umh.ac.be
http://w3.umh.ac.be/genlog

Abstract. Model-driven software engineering promotes the use of models and transformations as primary artifacts. Several formalisms can be used for the specification of model transformations. In this tutorial, we introduce and discuss such a formalism that is based on graphs and graph transformations. In particular, we focus on the activity of model refactoring, and show how graph transformation can provide formal support for this activity. We also show how such support can be implemented in state-of-the-art graph transformation tools such as *AGG* and *Fujaba*.

1 Introduction

Model-driven engineering is a software engineering approach that promotes the usage of models and transformations as primary artifacts. Its goal is to tackle the problem of developing, maintaining and evolving complex software systems by raising the level of abstraction from source code to models. As such, model-driven engineering promises reuse at the domain level, increasing the overall software quality. *Model transformation* is the heart and soul of this approach [1].

Graph transformation seems to be a suitable technology and associated formalism to specify and apply model transformations for the following reasons:

- Graphs are a natural representation of models that are intrinsically graph-based in nature (e.g., statecharts, activity diagrams, collaboration diagrams, class diagrams, Petri nets), as opposed to source code for which a tree-based approach is likely to be more appropriate. In Bézivin's tutorial on model-driven engineering [2], this link between models and graphs is explained as follows: "... we will give a more limited definition of a model, in the context of MDE only, as a graph-based structure representing some aspects of a given system and conforming to the definition of another graph called a metamodel."
- Graph transformation theory provides a formal foundation for the automatic application of model transformations. As such, one can reason about many interesting formal properties such as confluence, sequential and parallel dependence, and so on.
- Tool support for model-driven development based on graph transformation engines is starting to emerge (e.g., *GReAT* [3], *MOLA* [4] and *VIATRA* [5]).

An important activity within the domain of model transformation is *model refactoring*. The term refactoring was originally introduced by Opdyke in his seminal PhD dissertation [6] in the context of object-oriented programming. Martin Fowler [7] defines this activity as "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure". Recently, research interest has shifted from program refactoring to model refactoring [8,9,10,11,12], which aims to apply refactoring techniques at model level as opposed to source code.

The objectives of this tutorial are threefold:

- To introduce the notion of model refactoring as a special kind of model transformation activity, and to motivate the importance of this activity in the MDE process;
- To introduce graph transformation as a promising technique (covering both theoretical foundations and tool support) for software transformation;
- To show how graph transformation can provide formal support to automate the activity of model refactoring, and to compare graph transformation to related approaches.

The rest of this tutorial is structured as follows. Section 2 provides a high-level overview of model transformation and model refactoring, and introduces the necessary terminology. Section 3 introduces some theory on graph transformation. Section 4 discusses and compares *AGG* [13] and *Fujaba* [14,15], two general-purpose graph transformation tools, and shows how these tools can be used to implement support for model refactoring. Finally, section 7 concludes.

2 Model transformation

The aim of this section is to give a general high-level overview of model transformation, and to show where model refactoring fits in. In order to do this, it is important to be aware of the fact that model refactoring represents only a very specific kind of model transformation. To illustrate this, we briefly discuss a taxonomy of model transformation in the first subsection.

2.1 Taxonomy

[16] presented a detailed taxonomy of model transformation and showed how it could be applied to graph transformation. We will summarise some important ideas of the model transformation taxonomy here. Applying graph transformations to model transformation in general, however, is outside the scope of this paper.

In order to transform models, these models need to be expressed in some modeling language, the syntax of which is expressed by a *metamodel*. Based on the metamodels that are used for expressing the source and target models of a transformation, a distinction can be made between *endogenous* and *exogenous* transformations. Endogenous transformations are transformations between models expressed in the same metamodel. Exogenous transformations are transformations between models expressed in different metamodels. A typical example of an exogenous transformation is *migration* of a model

a program written in one particular language to another one. A typical example of an endogenous transformation is *refactoring*, where the internal structure of a model is improved (with respect to a certain software quality characteristic) without changing its observable behaviour [7].

Besides this distinction between endogenous and exogenous model transformations, we can also distinguish horizontal and vertical model transformations. A *horizontal transformation* is a transformation where the source and target models reside at the same abstraction level. A typical examples is again *refactoring* (an endogenous transformation). A *vertical transformation* is a transformation where the source and target models reside at different abstraction levels. A typical example of a vertical transformation is *synthesis* of a higher-level, more abstract, specification (e.g., a UML design model) into a lower-level, more concrete, one (e.g, a Java program).

Table 1 illustrates that the dimensions *horizontal versus vertical* and *endogenous versus exogenous* are truly orthogonal, by giving a concrete example of all possible combinations. As a clarification for the *Formal refinement* mentioned in the table, a specification in first-order predicate logic or set theory can be gradually refined such that the end result uses exactly the same language as the original specification (e.g., by adding more axioms).

Table 1. Orthogonal dimensions of model transformations

	horizontal	vertical
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language migration*	*Synthesis (e.g., code generation)*

Exercise 1. Identify some other types of model transformation (besides the four mentioned in Table 1) and classify them according to the taxonomy presented above.

2.2 Model refactoring

An emerging research trend is to port the idea of refactoring to the modeling level, for example by applying refactoring techniques to UML models [8,9,10].

Boger *et al.* developed a refactoring browser integrated with a UML modeling tool [9]. It supports refactoring of class diagrams, statechart diagrams, and activity diagrams. For each of these diagrams, the user can apply refactorings that cannot easily or naturally be expressed in other diagrams or in the source code.

Sunyé *et al.* formally defined some statechart refactorings using OCL pre- and postconditions [8]. Similarly, Van Gorp *et al.* propose a UML extension to express the pre- and postconditions of source code refactorings using OCL [10]. The proposed extension allows an OCL empowered CASE tool to verify nontrivial pre and postconditions, to compose sequences of refactorings, and to use the OCL query engine to detect bad code smells. Such an approach is desirable as a way to refactor designs independent of

the underlying programming language. Correa and Werner built further on these ideas, and implemented the refactorings in OCL-script, an extension of OCL [17].

An alternative approach is followed by Porres [18], who implements model refactorings as rule-based update transformations in SMW, a scripting language based on Python.

Zhang *et al.* developed a model transformation engine that integrates a model refactoring browser that automates and customises various refactoring methods for either generic models or domain-specific models [12].

Exercise 2. For each type of UML diagram you are familar with, try to come up with one or more examples of a model refactoring, i.e., a transformation that improves the structure of a UML model yet preserves its behaviour.

In the remainder of this tutorial, we will mainly restrict ourselves to model refactoring of UML class diagrams for various reasons. However, most of the ideas explained in this tutorial are directly applicable to refactorings of other kinds of models as well. Even domain-specific models and non-UML-compliant models (e.g. database schemas in ER notation) can be targeted. The primary restriction is that the models have to be expressible in a diagrammatic, graph-like notation.

2.3 Model consistency

Another crucial aspect of model transformation, and model refactoring in particular, is *model consistency*. It will not be treated in this tutorial, but we briefly mention some relevant related work here.

Spanoudakis and Zisman [19] provided an excellent survey on inconsistency management in software engineering. According to them, an inconsistency is “a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not jointly satisfiable". They claim that the following activities are essential for inconsistency management: detection of overlaps, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking, specification and application of an inconsistency management policy.

Since a UML model is typically composed of many different diagrams, the consistency between all these diagrams needs to maintained when any of them evolves. Sunyé *et al.* explored how the integrity of class diagrams and statecharts could be maintained after refactorings [8]. Van Der Straeten *et al.* explored the use of description logics as a way to specify and implement consistency rules and behaviour preservation rules [20,11].

Exercise 3. Give some examples of consistency problems that can occur in a UML model consisting of different kinds of UML diagrams. Consider only class diagrams, state transition diagrams (describing the internal behaviour of a class) and sequence diagrams (describing the message interations between class instances). Inconsistencies between these diagrams can arise when any of these diagrams is modified.

The problem of consistency maintenance becomes even more problematic when we acknowledge the obvious fact that models are only an intermediate step in the software development life-cycle, where the actual executable program is the most important deliverable. In this context, consistency should be maintained between the modeling level and the implementation level. Modifications to the models should be automatically reflected in the source code and vice versa. Again, this is a far from trivial problem that is outside the scope of this paper.

Exercise 4. Give an example of a consistency problem that can arise between UML design models and their generated source code when a small change is made to either the design models or the source code.

If you have a CASE tool available that allows you to generate programs from UML models, or to extract UML models from source code, verify whether this consistency problem is addressed by the CASE tool.

3 Graph transformation theory

Since the aim of this tutorial is to study the use of graph transformation for the purpose of model refactoring, in this section we will introduce the necessary theory and terminology about graph transformation. In Section 4 we will try to put this theory into practice by using two concrete graph transformation tools, *AGG* and *Fujaba*.

3.1 Introduction

Graph transformation theory has been developed over the last three decades as a suite of techniques and tools for formal modeling and very high-level visual programming. Graph transformations can typically be found in two flavours: *graph grammars* and *graph rewriting*.

Graph grammars are the natural extension of Chomsky's generative *string grammars* into the domain of graphs. Production rules for (string-) grammars are generalized into production rules on graphs, which generatively enumerate all the sentences (i.e., the "graphs") of a graph grammar. Similarly, *string rewriting* can be generalized into graph rewriting. A string rewriting consists of a pattern and a replacement string. The pattern is matched against an input string, and the matched substring is replaced with the replacement string of the rule. In analogy, a graph rewriting consists of a pattern graph and a replacement graph. The application of a graph rewriting rule matches the pattern graph in an input graph, and replaces the matched subgraph with the replacement graph.

Many tools, even full-fledged programming environments (e.g., *AGG* [13] and *Fujaba* [15]), have been developed that illustrate the practical applicability of the graph transformation approach. These environments have demonstrated that (1) complex transformations can be expressed in the form of rewriting rules, and (2) graph rewriting rules can be compiled into efficient code. In recent years, a number of model transformation tools have emerged that use graph transformation as an underlying transformation engine. Concrete examples are *GReAT* [3], *MOLA* [4] and *VIATRA* [5]).

3.2 Graphs

According to Bézivin and many others, a model can naturally be represented as a graph-based structure. Figure 1 clarifies this idea, and shows the correspondence between models and their graph representation. In this subsection, we will formally define the notions of graph and type graph, as well as how they are related.

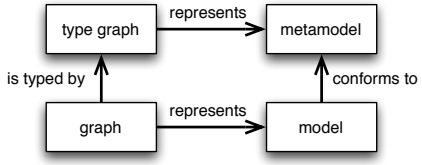

Fig. 1. Relationship between models and their graph representation.

Definition 1. *Directed labelled graphs.*

A (directed, labelled) graph $G = (V_G, E_G, s_G, t_G, l_G)$ *has a set of vertices* V_G *and a set of edges* E_G *such that* $V_G \cap E_G = \emptyset$*, functions* $s_G : E_G \rightarrow V_G$ *and* $t_G : E_G \rightarrow V_G$ *to associate to each edge a source and target vertex, and a labelling function* $l_G : V_G \cup E_G \rightarrow \mathcal{L}$ *to assign a label to each vertex and edge.*

Example 1. A concrete example of a graph is shown in Figure 4. This graph can be seen as the flattened variant of the UML class diagram of Figure 3, representing the design model of a local area network simulation (LAN). The behaviour of the LAN is visually represented in Figure 2.

It is worthwhile to note that this example has been used at different universities to teach object-oriented design and programming concepts, as well as to teach refactoring principles [21].

Definition 2. *Graph morphism.*

Let G *and* H *be two graphs. A graph morphism* $m : G \rightarrow H$ *consists of a pair of partial functions* $m_V : V_G \rightarrow V_H$ *and* $m_E : E_G \rightarrow E_H$ *that preserve sources and targets of edges, i.e.,* $s_H \circ m_E = m_V \circ s_G$ *and* $t_H \circ m_E = m_V \circ t_G$*. It also preserves vertex labels and edge labels, i.e.,* $l_H \circ m_V = l_G$ *and* $l_H \circ m_E = l_G$*.*

A graph morphism $m : G \rightarrow H$ *is injective (surjective) if both* m_V *and* m_E *are injective (surjective). It is isomorphic if* m *is injective and surjective. In that case, we write* $G \cong H$*.*

Note that the functions m_V and m_E are partial to allow for vertex deletions and edge deletions. All vertices in $V_G \setminus dom(m_V)$ and all edges in $E_G \setminus dom(m_E)$ are considered to be deleted by m.

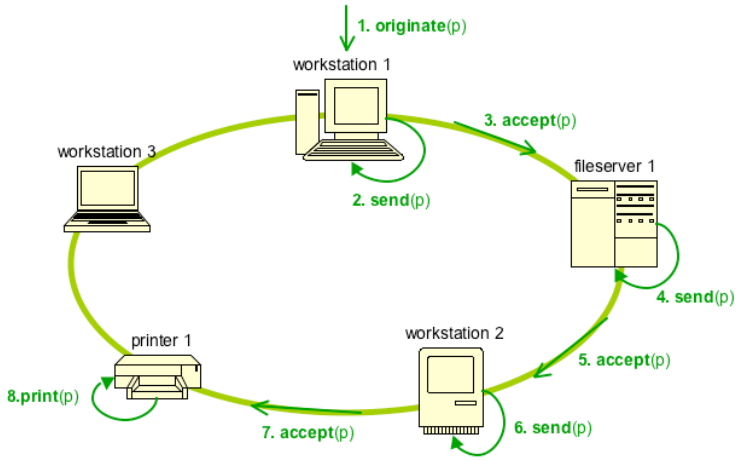

Fig. 2. Visualisation of the behaviour of a Local Area Network (LAN).

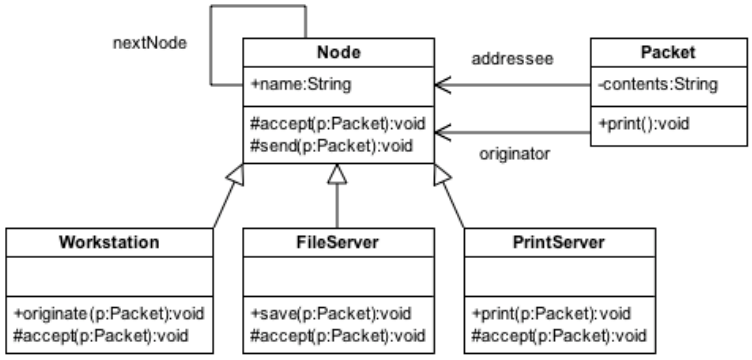

Fig. 3. UML class diagram of the LAN simulation.

Given a graph, it is often useful to determine its well-formedness, by checking whether it conforms to a so-called *type graph*. The formal definition of typed graphs is taken from [22]. Basically this boils down to the same idea as the one that is taken in model-driven engineering, where each model (e.g. a UML design model) needs to conform to a metamodel (e.g., the UML metamodel) [2]. The correspondence between both ideas is depicted in Figure 1.

Definition 3. *Typed graph.*

Let TG be a graph (called the type graph*). A* typed graph *(over TG) is a pair (G, t) such that G is a graph and $t : G \to TG$ is a graph morphism. A* typed graph morphism *$(G, t_G) \to (H, t_H)$ is a graph morphism $m : G \to H$ that also preserves typing, i.e., $t_H \circ m = t_G$.*

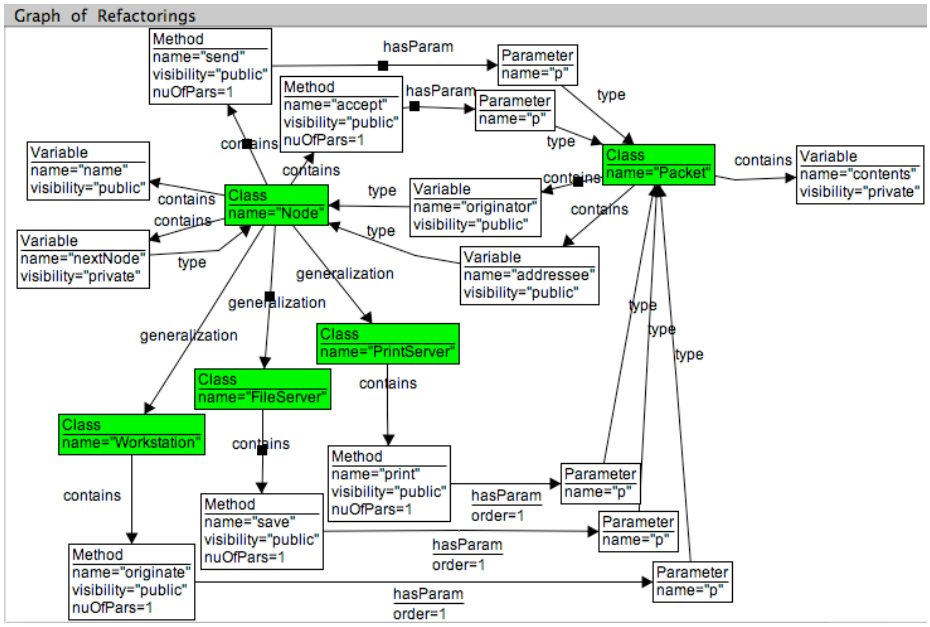

Fig. 4. Example of a graph representing the UML class diagram of the LAN simulation modeled in Figure 3.

Given a graph, it is also possible and very useful in practice to attach additional information to vertices and edges by attributing them. In that case, we talk about *attributed graphs*. Each vertex or edge can contain zero or more attributes. Such an attribute is typically a name-value pair, that allows to attach a specific value to each attribute name. These values can be very simple (e.g., a number or a string) or more complex (e.g., a Java expression). Examples of both will be shown later.

Example 2. As an example of an attributed graph, reconsider Figure 4. It is effectively attributed in the sense that some or all of its vertices contain attributes with names `name`, `visibility` and so on. Even some of the edges are attributed with name `order`.

For attributed graphs, the notion of type graph can be extended to constrain the names and types of attributes that are allowed for certain types of vertices and edges.

Example 3. An example of an attributed *type graph*, representing a simplified object-oriented metamodel, is shown in Figure 5. It expresses the following constraints on concrete graphs:

Constraints between nodes and edges: Classes can be related by generalization (*gen*-edges, or their transitive variant *tgen*). Classes contain Methods and Variables. Methods send Messages to each other and have a number of Parameters. Methods access or update Variables. Variables and Parameters are typed by Classes.

Multiplicity constraints on edges: For example, each Variable or Method is contained in exactly one Class. A Class contains zero or more Variables and Methods.

Attribute contraints: For example, the number of Parameters of a Method, as well as the name and visibility of Methods and Variables, is represented by vertex attributes. The order of a Parameter in a Method declaration is represented by an edge attribute.

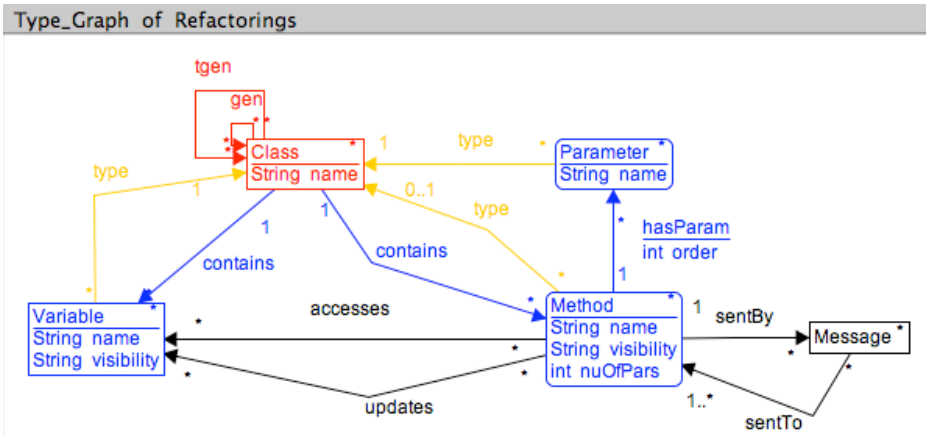

Fig. 5. Example of a type graph representing a simplified object-oriented metamodel. Observe that the vertices and edges of this type graph are attributed. The graph of Figure 4 is a concrete instance graph of this type graph.

In the remainder of the paper, when we use the term *graph*, we will always refer to an attributed, typed, directed labelled graph, unless indicated otherwise. Obviously, other variants of graphs exist but they will not be treated here, as they are not used by the graph refactoring tools we will employ for our experiments in Section 4

Exercise 5. Give several examples of concrete instance graphs that conform to the type graph of Figure 5. Give some examples of instance graphs that do not conform to the type graph and explain why.

Exercise 6. Check the multiplicities of each edge in the type graph and explain why they are there. (E.g., analyze the multiplicities of the *generalization* edge to determine whether or not the type graph allows for multiple inheritance.)

Exercise 7. Explain how method overriding, method overloading and late binding (or dynamic binding) can be modeled in this type graph. Give a concrete example.

Remark. The *signature* of a method is determined by the method name, the number of method arguments, the order of these arguments, the type of each argument, and the return type of the method. With *method overloading*, the same class can contain or understand more than one method with the same name, but a different signature. With

method overriding, a method with a given signature can be redefined with another implementation in a descendant class. With *late binding* (or dynamic binding), a message can be sent to a method whose signature occurs more than once in the inheritance hierarchy (due to method overriding). The actual receiver method of the message will be bound at execution time, depending on the dynamic type of the object that receives the message.

Exercise 8. Extend the type graph of Figure 5 so that it is possible to model exception handling in the Java way. An exception is modelled as a class, and a method can throw exceptions or catch exceptions. (For more details, please consult a Java reference manual.)

Exercise 9. Compare the type graph of Figure 5 with the part of the UML metamodel that is relevant to express class diagrams. Discuss the differences and determine if and how it is possible to integrate these differences into a more complete type graph.

Exercise 10. Do you know any other object-oriented metamodels? If yes, discuss and compare them with the one of Figure 5.

3.3 Graph transformations

Definition 4. *Graph production (rule).*

Let L and R be labelled graphs. A graph production is a graph morphism $p : L \rightarrow R$.

Example 4. An example of a graph production is shown in Figure 6. It moves a method one level up in the class hierarchy. (We use the edge *tgen* to denote transitive generalization of a class, i.e., any direct or indirect superclass or the class itself.) The left-hand side L is shown on the left, the right-hand side R is shown on the right of the figure. Vertices and edges that are preserved have the same number in L and R. In this example, only one edge (of type *contains*) is removed in L, and another edge (of the same type) is added in R. All other vertices and edges are preserved.

A graph transformation is the result of applying a graph production rule $p : L \rightarrow R$ in the context of a concrete graph G. Informally, this is performed by (i) finding a match of the left-hand side L in the graph G; (ii) creating a context graph by removing the part of the concrete graph that is mapped to L but not to R; (iii) gluing the context graph with those vertices and edges of R that do not have a counterpart in L. Formally, we follow the single pushout approach with injective graph morphisms [23].

Definition 5. *Graph transformation.*

A graph transformation $G \Rightarrow_t H$ is a pair $t = (p, m)$ consisting of a graph production $p : L \rightarrow R$ and an injective graph morphism (called match*) $m : L \rightarrow G$.*

Using a category-theoretical construct called pushout*, one can automatically compute the morphisms $m' : R \rightarrow H$ and $p' : G \rightarrow H$ that make the diagram (p, m) commute. The graph H obtained through this process is the result of applying the graph transformation t to G.*

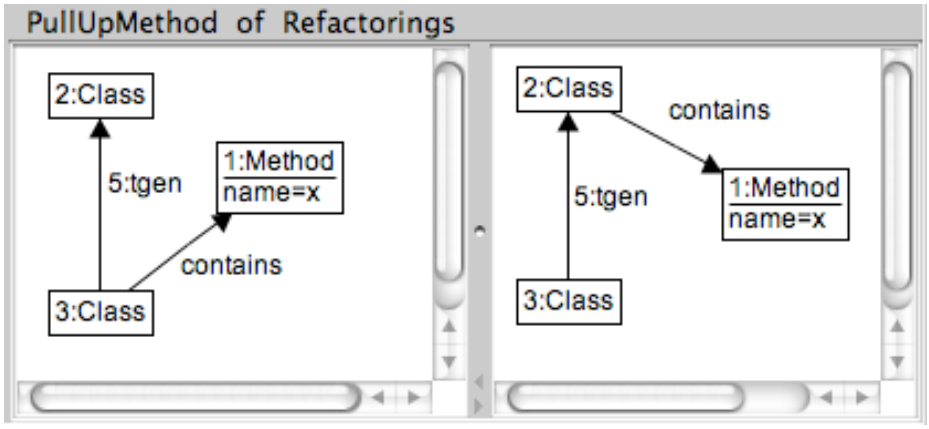

Fig. 6. A *Pull Up Method* graph production, represented in *AGG* notation. The left pane represents the production's left-hand side, the right pane the production's right-hand side. Numbers are used to identify the vertices and edges that are preserved by the graph transformation.

In graph transformation systems with a large number of graph productions it is often necessary to restrict the application of productions. Therefore, in [24,25] the notion of *negative application conditions* (NAC) is introduced. Intuitively, a NAC is a graph that defines a forbidden graph structure (e.g., the absence of some vertices or edges).

The mechanism of graph transformation can be extended easily to deal with application conditions, by checking all NACs associated to the graph production in the context of the concrete input graph G. Needless to say, the introduction of this notion of application conditions makes graph transformation considerably more expressive.

Definition 6. *Negative application condition.*

Let $p : L \rightarrow R$ be a graph production. A negative application condition *for p is a graph morphism $nac : L \rightarrow \hat{L}$. A graph transformation $G \Rightarrow_{(p,m)} H$ satisfies a* negative *application condition nac if no graph morphism $\hat{m} : \hat{L} \rightarrow G$ exists such that $\hat{m} \circ nac = m$.*

In practice, several NACs can be attached to a single graph production, i.e., each production p has an associated set N of NACs.

Definition 7. *Applicability of a graph transformation.*

Let $\hat{p} = (p, N)$ be a graph production $p : L \rightarrow R$ together with a set N of negative application conditions. A graph transformation $G \Rightarrow_{(\hat{p},m)} H$ is applicable if $G \Rightarrow_{(p,m)} H$ satisfies each negative application condition in N.

Example 5. An example of a NAC for the PullUpMethod graph production of Figure 6 is shown in Figure 7. The NAC is shown on the left. It specifies that the method cannot be pulled up if a method with the same name already exists in the class to which the method needs to be pulled up.

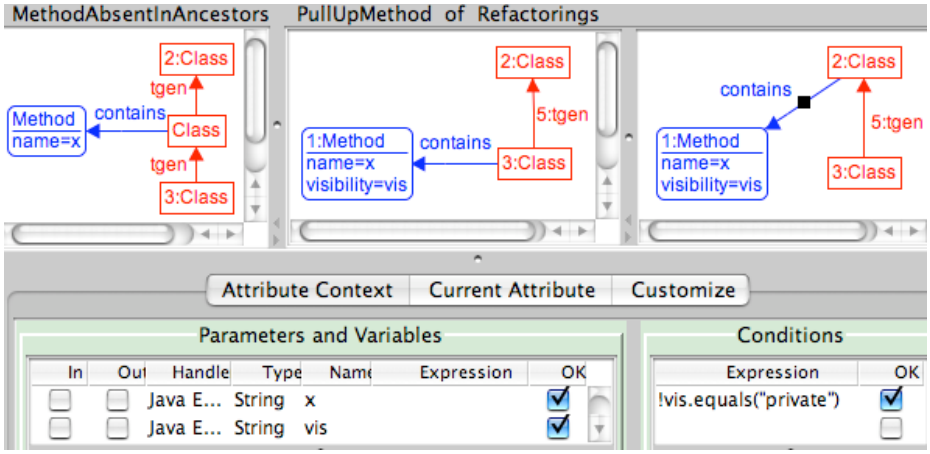

Fig. 7. A *Pull Up Method* graph production with negative application condition (NAC). The middle pane specifies the production's left-hand side, the right pane the production's right-hand side, and the left pane the NAC. It specifies that the method with name `x` and visibility `vis` in class number 3 cannot be pulled up if there is already an ancestor class containing a method with name `x`. Note that the figure also shows another NAC that cannot be expressed graphically because it has to do with the value of one of the vertex attributes. More specifically, the Java condition `!vis.equals("private")` expresses that only methods with a non-private visibility can be pulled up.

Exercise 11. Given the type graph of Figure 5 representing an object-oriented metamodel, try to specify a graph transformation to move a method from one class to another one. Do not forget to specify the negative application conditions that are needed for such a graph transformation.

3.4 Programmed graph transformation and graph grammars

To cope with a graph transformation system that contains a large number of graph productions, we need additional mechanisms to reduce the complexity. In this section we briefly introduce two alternative approaches: programmed graph transformation and graph grammars.

With *programmed graph transformation*, it is possible to specify a control structure that controls the order in which graph productions can be applied. Basically, this implies that we can use sequencing, branching and loop constructs to control the order of application of graph productions. Programmed graph transformation is supported by tools such as *Fujaba* [15] and *PROGRES* [26]. In particular, Fujaba uses an intuitive and compact notation, called *story diagrams*, to represent both the graph productions as well as their order of application. We will come back to this specific notation in Section 4.2.

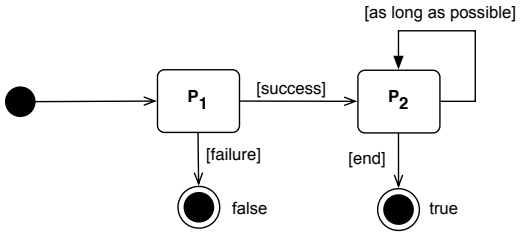

Fig. 8. An example of a programmed graph production composed of two graph productions P_1 and P_2.

Example 6. An example of a programmed graph production is shown in Figure 8. It shows the application order of two graph productions P_1 and P_2. First P_1 is applied, and if it succeeds, P_2 is aplied as long as possible.

This programmed graph production can be used to express a more sophisticated variant of *Pull Up Method*, that is composed of two graph productions. The first production, P_1 (shown in Figure 7), takes a method in a class C and moves it to its parent class. The second production, P_2, looks for the same method signature in a sibling of C (i.e., a class with the same parent as C) and deletes this method. This production is repeated for each sibling of C where the same method signature can be found.

Exercise 12. Formally specify the graph production P_2 explained above in the context of *Pull Up Method*.

With graph grammars, no control structure is imposed, and all available graph productions are applied non-deterministically or at random. As such, a given initial graph G can give rise to a whole range of possible result graphs, which is referred to as $L(G)$, the language generated by the graph grammar. Each word in this language corresponds to a possible sequence of graph transformations that can be applied to G. Graph grammars are used in the *AGG* tool.

3.5 Confluence and critical pairs

Definition 8 (Confluence). *A relation $R \subseteq A \times A$ is called* confluent *if* $\forall a, b, c \in A$*: if aRb and aRc then $\exists d \in A$: bRd and cRd*

Confluence is well-known in term rewriting, and is used to check whether a term rewriting systems (i.e., a term grammar) has a functional behaviour. Irrespective of the order in which the term rewritings are applied the end result should always remain the same. These confluence results can also be shown for the more general graph grammars [27].

Given a term grammar (or a graph grammar), it is crucial to know whether this grammar has the confluence property. To determine this, the notion of *critical pair analysis* has been introduced for term rewriting, and has been generalised later for graph

rewriting [27]. Critical pairs formalize the idea of a minimal example of a conflicting situation. From the set of all critical pairs we can extract the vertices and edges which cause conflicts or dependencies.

To find all conflicting productions in a graph grammar, minimal critical graphs are computed to which productions can be applied in a conflicting way. Basically, one has to consider all overlapping graphs of the left-hand sides of two productions with the obvious matches and analyze these rule applications. All conflicting rule applications are called critical pairs. If one of the rules contains NACs, the overlapping graphs of one left-hand side with a part of the NAC have to be considered in addition.

Definition 9 (Conflict). *Two graph transformations* $G_1 \Rightarrow_{(p_1,m_1)} H_1$ *and* $G_2 \Rightarrow_{(p_2,m_2)} H_2$ *are in conflict if* p_1 *may disable* p_2*, or, vice versa,* p_2 *may disable* p_1*.*

There is a conflict if at least one of the following three conditions are fulfilled. The first two are related to the graph structure while the last one concerns the graph attributes.

1. One rule application deletes a graph object which is in the match of another rule application. 2. One rule application generates graph objects in a way that a graph structure would occur which is prohibited by a NAC of another rule application. 3. One rule application changes attributes being in the match of another rule application.

Definition 10 (Glue graph). *Given a pair of graph productions* $p_1 : L_1 \rightarrow R_1$ *and* $p_2 : L_2 \rightarrow R_2$*. Their* glue graph G *can be computed by overlapping* L_1 *and* L_2 *in all possible ways, such that the intersection of* L_1 *and* L_2 *contains at least one item that is deleted or changed by one of the productions and both productions are applicable to* G *at their respective occurences.*

Definition 11. *[Critical pair] A* critical *pair is a pair of graph transformations* $G \Rightarrow_{(p_1,m_1)} H_1$ *and* $G \Rightarrow_{(p_2,m_2)} H_2$ *which are in conflict, and* G *is the minimal glue graph of* p_1 *and* p_2*.*

The set of critical pairs represents precisely all potential conflicts between a given pair of graph productions (p_1, p_2). Therefore, we can apply critical pair analysis to a graph grammar (i.e., a set of graph productions), by performing a pairwise comparison of all graph productions in the grammar. After computation of all critical pairs for every pair of graph productions, the production set will be divided into conflict-free pairs and conflicting pairs.

Example 7. The *Pull Up Method* graph production of Figure 7 and the *Move Method* graph production of Figure 9 give rise to a critical pair situation, as depicted in Figure 10. The same method m is pulled up and moved by different graph productions. This clearly leads to a conflict, since both productions cannot be applied in sequence. Once the method m is pulled up, it can no longer be moved from its original location c, since it is no longer present there. The glue graph that identifies this critical pair is shown as a gray ellipse in the figure.

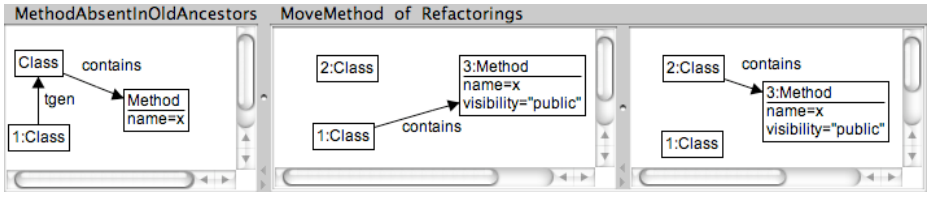

Fig. 9. A *Move Method* graph production with negative application condition. The three panes indicate, from left to right: a negative application condition, the left-hand side of the graph production, the right-hand side of the graph production.

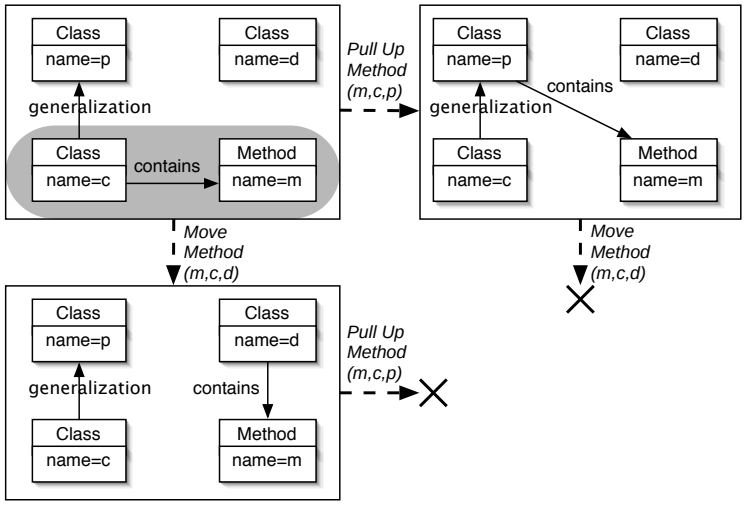

Fig. 10. Concrete example of a critical pair situation between the graph productions *Move Method* and *PullUpMethod*.

4 Specifying model refactorings as graph transformations

In the previous section we introduced the basic notions of graph transformation. In this section, we will show how these can be used to specify model refactorings. A comparison is given in Table 2.

To concretise the results mentioned in Table 2, we will start by introducing two general-purpose graph transformation tools, *AGG* and *Fujaba*. Then, we continue showing some experiments we have performed with these tools in the context of model refactoring.

Graph transformation	*Refactoring*
type graph and global graph constraints	well-formedness constraints
negative application conditions	refactoring preconditions
parameterised graph productions with NACs and context conditions	refactoring transformation
programmed graph transformations, story diagrams	composite refactorings
critical pair analysis	detecting refactoring conflicts
confluence analysis	detecting sequential dependencies

Table 2. Comparison of graph transformation and refactoring concepts

4.1 The AGG graph transformation tool

AGG [13] is a rule-based visual programming environment supporting a graph transformation language based on the so-called algebraic single-pushout approach [23]. *AGG* aims at the specification and prototypical implementation of applications with complex graph-structured data. It contains a general-purpose graph transformation engine that is implemented in Java.

AGG supports the specification of type graphs with multiplicities and attributes, such as the one shown in Figure 5. In *AGG*, vertex and edge attributes act like ordinary Java variables to which a value can be assigned. By specifying Java expressions, the graph production rules can specify how attribute values need to be updated by the transformation. The graph productions can also contain NACs and extra constraints (context conditions) that need to be satisfied when the production rule is applied in the context of an input graph. This is quite useful in practice, since the type graph and NACs are not always sufficiently expressive. An example of this was shown in Figure 7, where an extra context condition was needed to express a constraint on the visibility of the method to be pulled up.

In *AGG*, graph productions are stored as part of an attributed graph grammar. Given a start graph, the graph grammar can be applied by selecting rules that are applicable. A very useful feature of *AGG*, that is absent in other graph transformation tools, is that graph grammars may be validated using the techniques of *critical pair analysis* and *consistency checking* that were introduced in section 3.5.

Example 8. As a concrete example of a graph transformation in *AGG*, consider the specification of the refactoring *Encapsulate Variable* in Figure 11. The goal of this refactoring is to change the visibility of the name of a variable (from "public" to "private"). In addition, we need to introduce a getter and setter method for this method. The name of these methods depends on the name of the variable. This constraint can be expressed by means of the Java expressions `s.equals("set"+v)` and `g.equals("get"+v)` where `v` is the name of the variable, `s` the name of the setter method, and `g` the name of the getter method. The graph production also contains NACs, stating that the setter and getter methods introduced by the refactoring do not exist yet in the inheritance chain. One such NAC, called “noSetterInAncestors” is shown in the upper left pane of Figure 11.

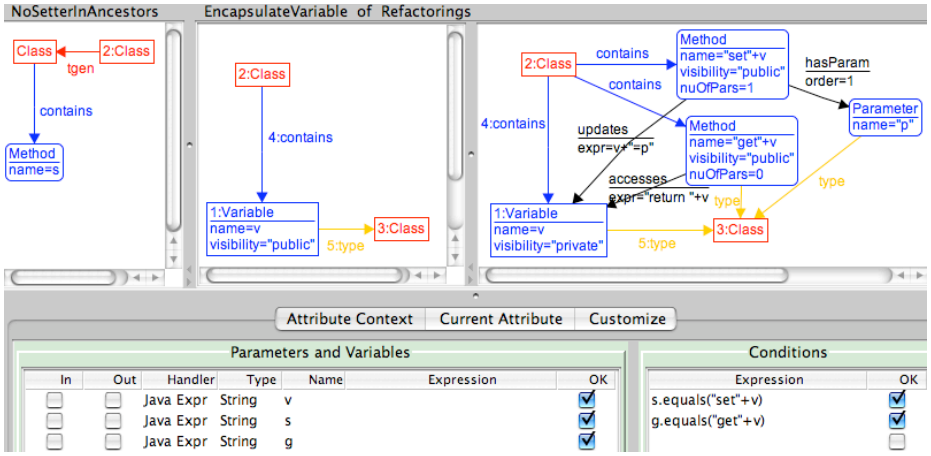

Fig. 11. *Encapsulate Variable* refactoring expressed as a graph production. The NAC in the left pane specifies that the name of the newly created setter method for the encapsulated variable should not exist in one of the ancestor classes. Note that this NAC also requires an extra context condition stating a relation between the value `v` of the name attribute of the variable and the value `s` of the name attribute of the method: `s.equals("set"+v)`

Exercise 13. Go to the AGG website `http://tfs.cs.tu-berlin.de/agg/`, download the latest version (in May 2005 this was version 1.2.6), and install it on your machine. Run AGG, and open the file `transrefactorings10.ggx` that can be found in the subdirectory `ExamplesV126/Refactorings` of the directory where you installed AGG. After loading this file, you should get something like the screenshot shown in Figure 12.

Exercise 14. The *Encapsulate Variable* refactoring explained in Figure 11 and Example 8 is not complete. It does not express the fact that we still need to redirect all accesses to the public variable by a call to its new setter method, and all updates to the public variable by a call to its new getter method. Specify a graph production in *AGG* that does exactly this.

Exercise 14 shows one of the shortcomings of *AGG*. It is not possible to specify a complex graph transformation that is composed as a sequence of more primitive transformations. This would require another mechanism, called programmed graph transformation, that has been explained in Section 3.4. Unfortunately, such a mechanism is not available in *AGG*. It is, however, available in the *Fujaba* tool that will be explained in the next subsection.

4.2 The Fujaba graph transformation tool

Fujaba is a graph transformation tool in Java that uses the UML notation for design and realisation of software projects. It uses a combination of activity diagrams and a spe-

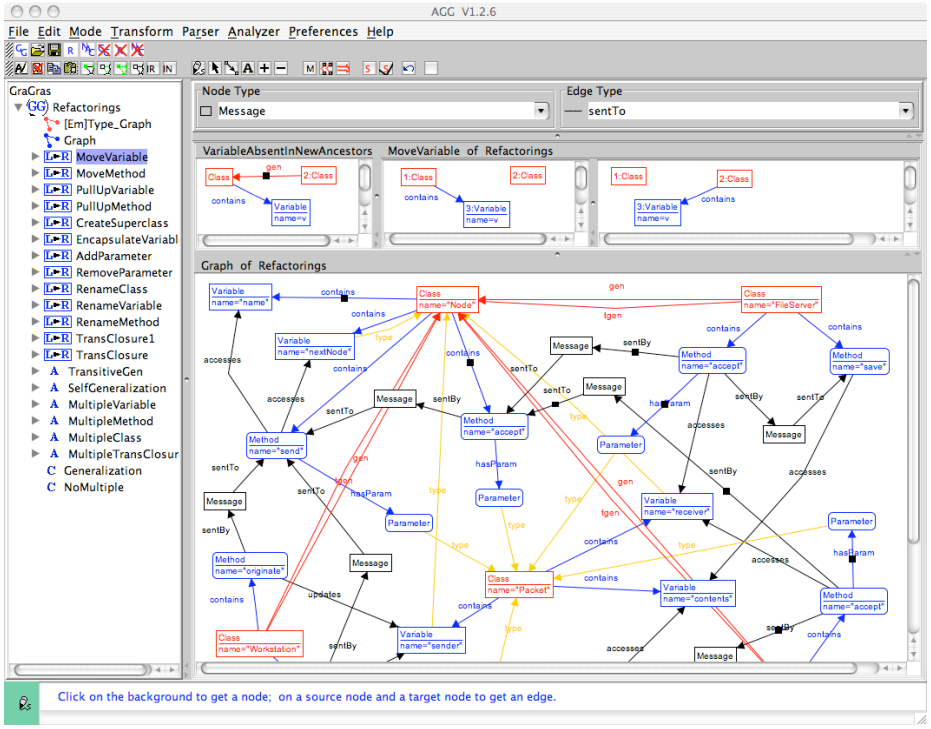

Fig. 12. Screenshot of the AGG version 1.2.6, after opening the Refactorings graph grammar.

cific variant of collaboration diagrams (called story diagrams) for the specification of operational behaviour. The semantics of these story diagrams are based on programmed graph transformations such as the ones shown in Figure 8. Story diagrams offer many powerful constructs of graph transformation such as multiobjects, non-injective matching, negative application conditions, and many more. This makes it a powerful language that allows to model even complex problems in an elegant way. The operational behaviour modeled with such story diagrams can then be tested using the graph-based object browser DOBS.

Fujaba generates standard Java code that is easily integrated with other Java programs and that runs in a common Java runtime environment. This enables the use of graph transformation concepts in all kinds of Java applications.

The way to specify refactorings in Fujaba is very similar to the way it is done in AGG. The main difference is the notation used by Fujaba, which is closer to UML, and hence more intuitive to users already acquainted with UML.

Example 9. As a concrete example to illustrate Fujaba's story diagram notation, we implemented the *EncapsulateField* refactoring in Fujaba as well. It is implemented by means of two methods `checkPreconditions` and `execute`. This separation allows us to check the precondition of a refactoring separately from its actual execution.

The implementation of both methods is specified in Fujaba by means of a story diagram, in Figure 13 and Figure 14, respectively.

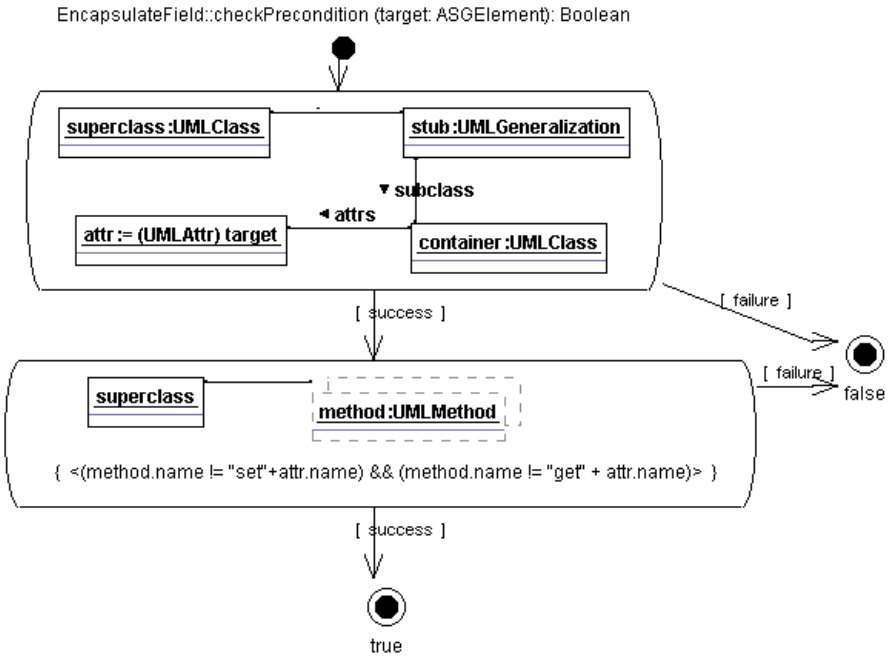

Fig. 13. Encapsulate Variable refactoring in Fujaba - Preconditions check

Exercise 15. Go to the *Fujaba* website `http://www.fujaba.de/`, download the latest version of the *Fujaba Tool Suite* (in March 2005 this was version 4.3.1), and install it on your machine. Run Fujaba, and open the file `Refactoring.fpr.gz` After loading this file, you should get something like the screenshot shown in Figure 15.

Exercise 16. The specification of the *Encapsulate Field* refactoring execution in Figure 14 is not complete. The story diagram does not specify that, after creating the `setter` and `getter` methods, all direct accesses or updates to the (previously public) attribute `attr` should be replaced by a method call to the `getter` and `setter` method, respectively. Modify the story diagram to include this part of the refactoring.
Hint. For the sake of simplicity, you may assume in this exercise that attribute accesses, attribute updates and message sends, are modelled in the same way as was the case in the AGG type graph. (In fact, this is an oversimplification.)

Exercise 17. Try to specify the *Pull Up Method* refactoring as a graph transformation in Fujaba, using the story diagram notation.
Hint. Take a look at the *Encapsulate Field* refactoring, whose specification is provided

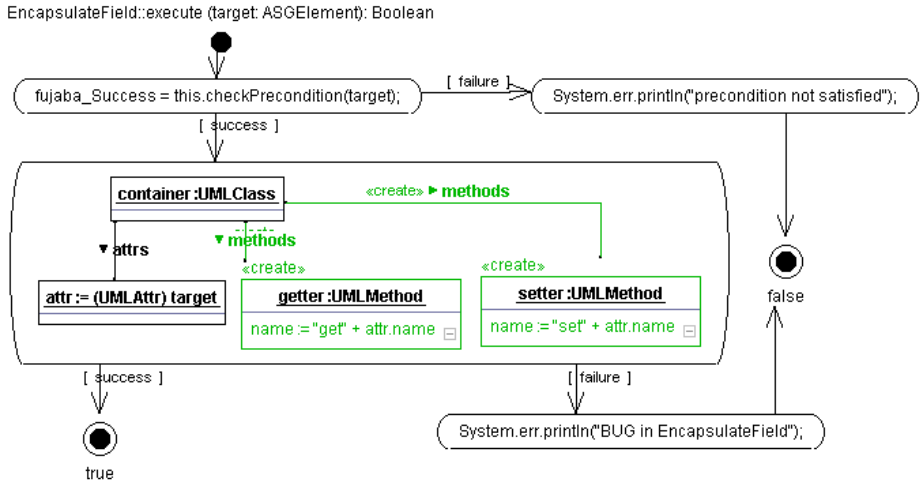

Fig. 14. Encapsulate Variable refactoring in Fujaba - Execution

in the project as two activity diagrams for the class `EncapsulateField`. Also inspect the metamodel, which is provided as a class diagram called `ASG`.

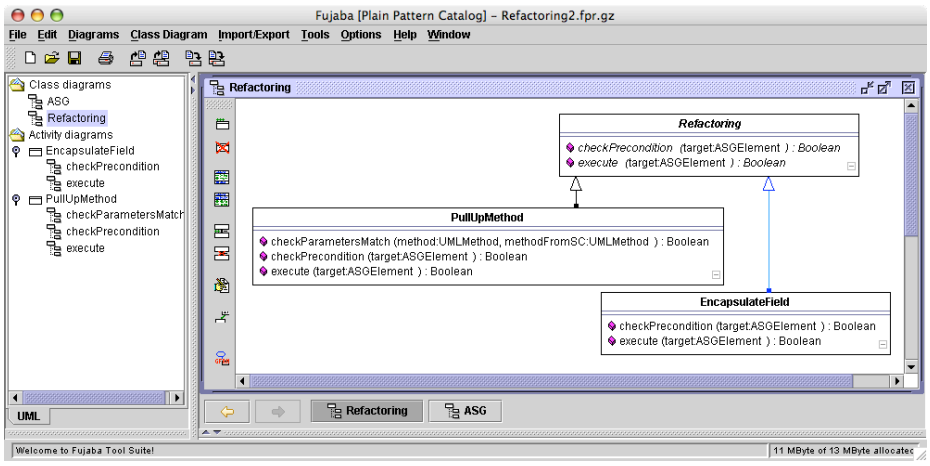

Fig. 15. Screenshot of the Fujaba Tool Suite version 4.3.1, after opening the Refactoring project.

An important feature of Fujaba is its very flexible plug-in mechanism. One such plug-in has been developed by the University of Kassel to provide support for some simple refactorings. When a refactoring is selected via the refactoring plug-in, the cor-

responding refactoring will be executed. Currently, the following list of refactorings has been implemented: *Extract Method*, *Override Method*, *Implement Method*, and *Change Method Signature*. However, the implementation of these refactorings was hard-coded in Java. Hence, this cannot be used as a proof of concept that graph transformation can effectively be used to implement model refactorings in the way suggested above.

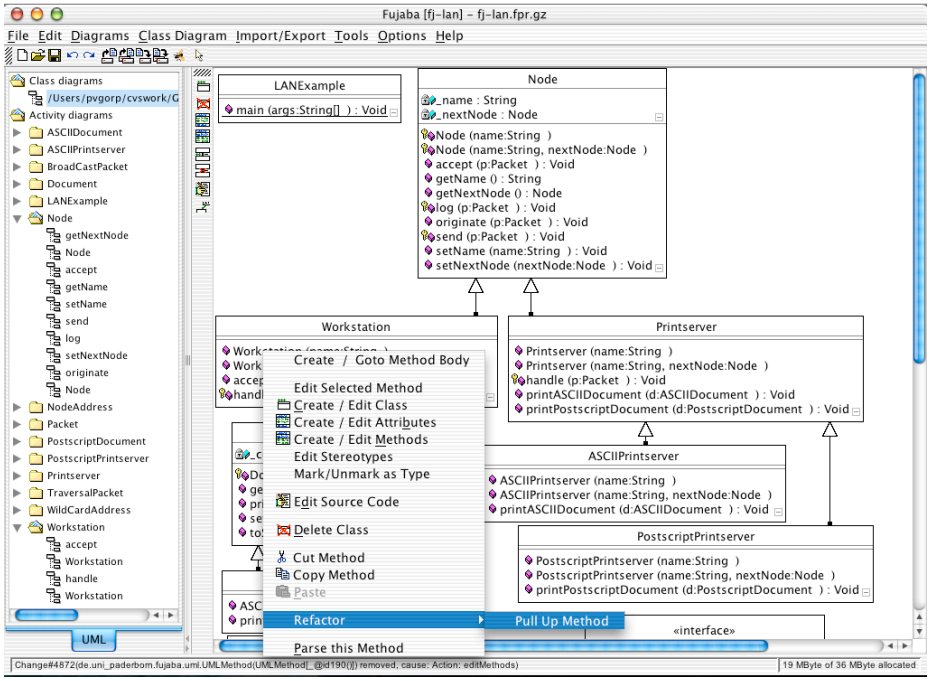

Fig. 16. Refactoring plugin for Fujaba

Therefore, in another experiment that has been documented in [28], Pieter Van Gorp from the University of Antwerp (Belgium) tried to implement some class diagram refactorings, such as the *Pull Up Method* refactoring, in Fujaba using the story diagram notation. Figure 16 illustrates how the *Pull Up Method* refactoring is selected in the context of the Local Area Network simulation. When this refactoring is applied, the Java code, that was automatically generated by Fujaba from the graph production specification, is executed. As such, the refactoring designer did not need to implement any Java code himself.

The above case study had two important limitations that were overcome in later experiments [29,30]:

– The refactoring specifications (transformation models) could not be exchanged with other tools. This problem was overcome by designing a UML profile for Fujaba's graph transformation language.

– The code generated by Fujaba could only be executed on repositories conforming to Fujaba's API conventions. This limitation was resolved by building a tool that generates MOF compliant code from transformation models conforming to Fujaba's UML profile. The class diagram refactoring case study could consequently be redone with general purpose UML tools such as MagicDraw and Poseidon.

4.3 Comparison of AGG and Fujaba

When we compare the functionality of the graph transformation tools *AGG* and *Fujaba*, we see that there are a lot of similarities. Both are implemented in Java, both make use of attributed, labelled, directed graphs, both use a type graph, and both support negative application conditions.

On the other hand, there are also a number of important differences. Probably the most striking one is the way in which the application of graph productions is controlled. *AGG* relies on a graph grammar approach. This means that the control structure is implicit and non-deterministic: whenever a graph production is found that is applicable in the context of the host graph, it is applied, and this process continues with the result graph as the new host graph. *Fujaba* takes the opposite approach. Since it relies on programmed graph transformation, the control structure is explicit and deterministic: the programmer needs to provide explicit sequencing, branching and loop constructs to control the order of application of graph productions.

Another important difference is the kind of tool support and consistency checking that can be provided in both tools. *AGG* provides support for critical pair analysis, in the way explained in section 3.5. *Fujaba* does not provide such support, but is on the other hand much better when it comes to round-trip engineering. In *Fujaba*, there is a seamless integration between UML modeling and Java programming.[1] As a result, the user of the tool can simply express his design as a UML class diagram, or directly write Java code and perform a reverse engineering step to automatically generate the class diagrams. The only place where graph transformation comes in is at the level of method implementations. In *Fujaba*, a method can be implemented as a graph production using a story diagram. (An example of this was presented in Figure 13 and Figure 14.)

4.4 Refactorings experiments in AGG

In order to specify model refactorings by means of graph transformations, one first needs to agree upon a *metamodel* that can be used to specify the models and model refactorings while abstracting away from implementation-specific details. This metamodel is specified as a *type graph* in *AGG*, and has already been introduced in Figure 5 of Section 3.2.

Note that not all possible well-formedness constraints can be expressed in the type graph. In *AGG*, this problem can be resolved by adding additional *global graph constraints*. We used this mechanism to express the following well-formedness constraints:

– no two classes should have the same name

[1] Remember that FUJABA is an acronym for "From Uml to Java And Back Again".

- no two methods contained in the same class should have the same name
- no two variables contained in the same class should have the same name
- If there are multiple methods with the same name in the same class hierarchy, any message sent to one of these methods should also be sent to all other methods with the same name in the hierarchy (since it is impossible to determine the actual receiver method statically due to the mechanism of dynamic method binding)

Refactoring transformations can be implemented in a straightforward way as *AGG* graph productions. Obviously, these productions have to respect the constraints imposed by the type graph of Figure 5, as well as the additional constraints mentioned above.

We already saw *Pull Up Method*, *Move Method* and *Encapsulate Variable* as concrete examples in Figures 7, 9 and 11. In a similar way, other refactorings can be implemented. For all these refactoring specifications, the preconditions were specified as NACs. For some refactorings, such as *Pull Up Method* and *Encapsulate Variable*, additional context conditions were needed for those constraints that could not be expressed in terms of the type graph. These context conditions were specified as ordinary Java expressions.

In Definition 11 of Section 3.5, we explained the notion of critical pairs and how it can be used to detect conflicts between graph productions. A concrete example of such a conflict was shown in Figure 10. *AGG* supports critical pair analysis for typed attributed graph transformations. Given a graph grammar, *AGG* can compute a table showing the number of conflicting situations for each critical pair of productions.

Example 10. We applied *AGG*'s critical pair analysis algorithm to a representative selection of refactorings. The results are shown in the table of Figure 17. Among others, we can see in this table that four critical pairs are reported between *Pull Up Method* and *Move Method*. Two of the critical graphs computed by AGG for this situation are shown in Figure 18. Both critical graphs report similar conflict situations that correspond to the conflict illustrated in Figure 10. The additional two conflicts not depicted are less interesting, since they report possible conflicts that cannot occur in our setting. This is due to the fact that AGG's critical pair algorithm abstracts away from concrete attribute interrelations. Since arbitrary Java expressions can be used for attribute conditions and computations, it just reports general conflicts on attribute usage, i.e., one rule application changes an attribute that another rule application uses. Acting in this way, it happens that some of the possible conflicts reported can never become real conflicts.

Exercise 18. Give some concrete examples of critical pairs, and visualise them in the same way as done in Figure 10.

In *AGG* it is also possible to check which of the refactorings are applicable to a concrete input graph: A refactoring is applicable if there exists at least one match of its left-hand side (taking into account the NACs). Figure 19 gives an example that shows that certain refactorings are not applicable in a particular situation. It is obtained by using AGGs menu item "Check Rule Applicability". *Pull Up Variable* and *Remove Parameter* are reported as non-applicable because, in the considered input graph, none of the subclasses had variables, and because all methods having parameters are called by others, thus prohibiting their removal.

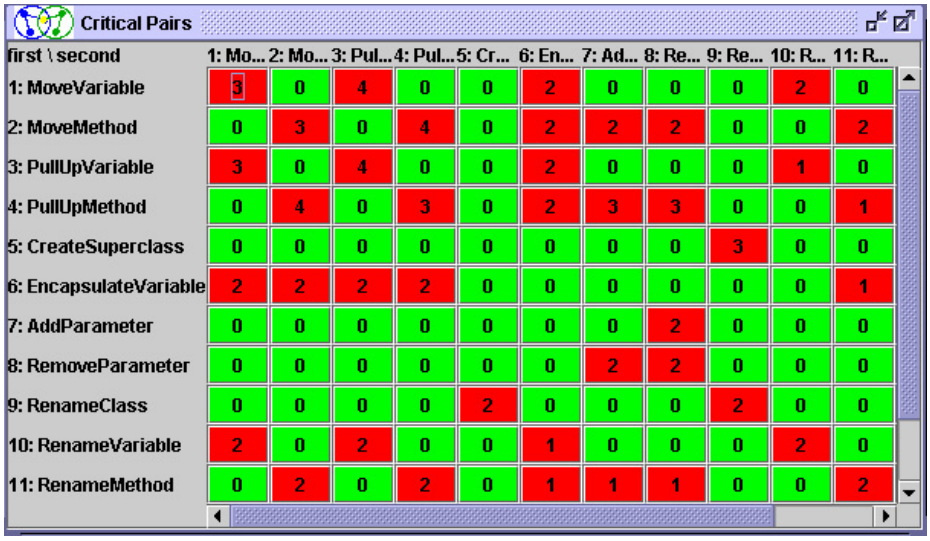

Fig. 17. Critical pair analysis

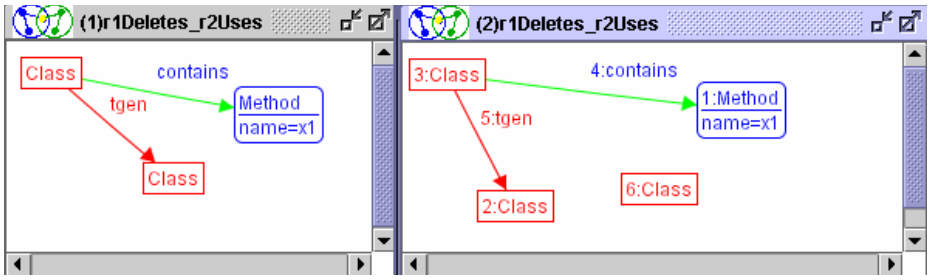

Fig. 18. Posible conflicts of *Move Method* and *Pull Up Method*

While the critical pair table of Figure 17 shows all potential conflicting situations that can occur between any pair of refactoring productions, the number of actual conflicts in the context of concrete input graph is of course much lower, since not all refactorings may be applicable to this concrete input graph. Therefore, *AGG* can also show the conflicts in concrete input graphs by selecting only the relevant critical pairs and showing how the corresponding conflict graphs are matched to the instance graph. An example is given in Figure 20.

For a more detailed discussion of the analysis we performed on the critical pairs of refactoring productions, we refer to [31].

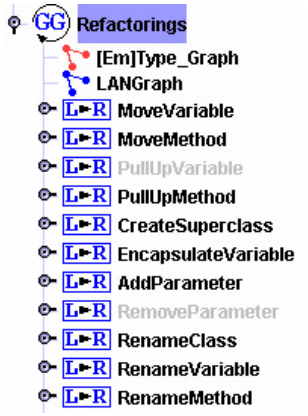

Fig. 19. Applicability of refactoring productions. Those that are applicable in the context of the given input graph are shown in black, the others are shown in gray.

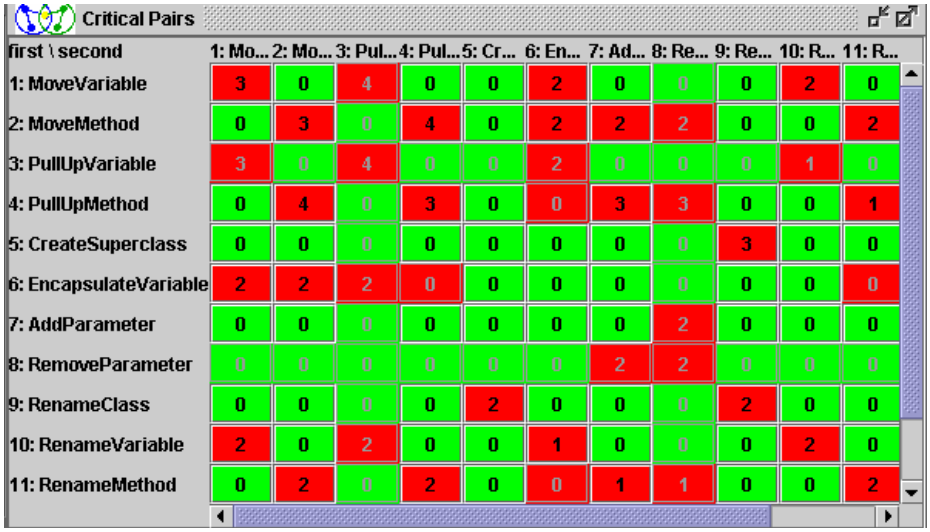

Fig. 20. Critical pairs reported in the context of a concrete input graph.

5 Benefits and drawbacks of graph transformation

In this section we briefly sketch some other reasons why graph transformation is a good underlying foundation for refactoring technology, and point out some references to relevant literature for the interested reader.

5.1 Guaranteeing behaviour preservation

In order to determine whether a refactoring preserves the correctness or the behaviour, we can also rely on graph transformation theory. Mens *et al.* [32] showed that graph transformation is a promising formalism to do this, but also indicated a number of limitations with respect to the expressiveness of existing graph transformation formalisms. Therefore, Van Eetvelde and Janssens [33] proposed to extend graph transformations with new mechanisms to enhance their expressive power.

5.2 Composition of refactorings

Another very important question in refactoring research is how to compose primitive refactorings into more complex, composite refactorings [34,35,36]. An essential question in this context is how, given a sequence of refactorings, one can compute the preconditions of the composite refactoring without needing to apply each refactoring in the sequence. As a partial answer to this question, in his masters thesis, Reiko Heckel theoretically showed how a sequence of graph transformations with application pre- and postconditions could be transformed into an equivalent composite graph transformation with pre- and postconditions [37]. Kniesel [36] explored this idea in the context of refactoring, and showed how it can be used to build tools that facilitate the static composition of refactoring transformations.

5.3 Co-evolution and consistency maintenance

To be able to focus on different aspects of a program, software engineers usually employ different views on the software. For example, a UML model usually consists of a class diagram, a set of use case diagrams, a set of interaction diagrams, and a set of state transition diagrams. All these diagrams represent a different view that is represented using a different modeling notation. Hence techniques are required to maintain the consistency between all these different views when one of them evolves (e.g., by means of a refactoring).

When we also take the source code into account, we even need to maintain the consistency between the models and the corresponding program. To ensure that both views remain consistent when applying refactorings, Bottoni *et al.* [38] proposed a framework based on distributed graphs to maintain consistency between the code (represented as a flow graph) and the model (given by several UML diagrams of different kinds). Each refactoring is specified as a set of distributed graph transformations, structured and organized into transformation units.

Other formal approaches based on graph transformations that seem promising to address the consistency problem are pair grammars and triple graph grammars.

5.4 Graph rewriting versus tree rewriting

During our experiments in *AGG* as well as *Fujaba* we encountered an issue that has to do with the expressiveness of graphs and graph transformations. When trying to

specify certain class diagram refactorings, in particular those that have to perform non-trivial manipulations of method bodies (e.g., *Extract Method*, *Move Method* and *Push Down Method*), the graph transformations quickly become very complex. This issue has already been acknowledged in [32,10,33].

Because a method body is essentially an abstract syntax tree, probably a better approach would be to make use of tree rewriting techniques [39,40], as they are much better suited for these kinds of manipulations. In practice, this implies that the ideal refactoring specification language would probably need to incorporate the best of both worlds: graph transformation for those parts of the model that are essentially graphical in nature (e.g., the class structure of a class diagram, including all inheritance, typing and association relationships), and tree transformation for those parts of the model that are essentially tree-based in nature (e.g., method parse trees).

6 Relation to the other tutorials

6.1 Model-Driven Engineering (Bézivin)

The relation between *model-driven engineering* (MDE), as introduced by Jean Bézivin in his tutorial, and *model refactoring*, as treated in this tutorial, is very straightforward and has already been explained in the introduction of this tutorial. Indeed, model transformation is an essential activity of model-driven engineering, and model refactoring is a particular kind of model transformation.

6.2 Database tranformations (Hainaut)

The notions of program transformation, program refactoring and model refactoring are clearly related to the transformational approach to database reengineering proposed by Jean-Luc Hainaut in his tutorial. In fact, one can even say that the research domain of software refactoring is a natural and logical successor of the research on *database schema restructuring* that has been going on (and still is, as a matter of fact) in the 1980s [41].

If we consider a database to be a model, the database schema is its metamodel that imposes certain well-formedness and other constraints on the database. Given such a database schema, we can define a schema restructuring as a transformation that provably *preserves* certain important properties that were imposed on the databases by the original schema. The properties to be preserved can be very diverse, ranging over data- and constraint preserving [42], correctness preserving, semantics preserving. The latter notion of semantics preservation, also known as reversibility, indicates the extent to which the target schema can be substituted by the source one without loosing information. For more details, we refer to Hainaut's tutorial.

It is based on the idea of schema restructurings being semantics-preserving schema transformations, that the notion of *software refactoring* was first introduced by John Opdyke in his seminal dissertation [6] as being a semantics-preserving object-oriented program transformation. In hindsight, the step from schema restructuring to program

restructuring was a logical one, given the close analogy between an object-oriented program and a (object-oriented) database schema. Indeed, a class diagram essentially represents the structure of an object-oriented program, much in the same way as a schema represents the structure of a database. Seen in this light, it makes perfect sense to apply restructuring techniques to programs. Even the techniques that have been used to specify schema transformations (typically in terms of pre- and postconditions) can be reused to a certain extent for program refactorings.

Unfortunately, if one wants to reason about the behaviour-preserving properties of a program transformation, this is extremely hard, and even undecidable in most practical situations. The main reason is that an object-oriented program, and even an object-oriented design model, not only contains data, but behaviour as well! Therefore, current results on schema restructuring do not suffice to prove the preservation of behaviour of program refactorings. For more details, we refer to a survey on software refactoring [43].

To conclude this discussion, it is interesting to note that the recent shift to model refactoring (as opposed to program refactoring) seems to bring the database research community and the software engineering community closer together again. Especially in the context of model-driven engineering, it makes perfect sense to consider a database schema as a special kind of model, and to use the same set of tools to manipulate and restructure these models.

6.3 Feature Oriented Programming (Batory)

As indicated by Don Batory in the introduction of his tutorial on *feature-oriented programming*, one of the crucial challenges of software engineering is to manage and control the ever-growing complexity of software systems. The problem is that the implementation level is too low-level to address this challenge, since it exposes too much detail to make it easy o design, construct and modify programs. This is the reason why we need to abstract away from this implementation detail, and start designing and modifying systems at the modeling level.

In addition, according to Batory, advancement on at least three fronts is needed: generative programming, domain-specific languages, and automated programming. While we tend to agree with this view, the current tutorial addresses an orthogonal and complementary issue. Even when a software system is designed or modeled at a high level of abstraction (possibly using a domain-specific language), we still require formalisms, techniques and tools that allow us to restructure or refactor the designs or models without affecting their functional behaviour. Therefore, our tutorial focusses on this issue of model refactoring, and suggests graph transformation as a possible implementation technology.

6.4 Reflective and Aspect-Oriented Program Transformation (Chiba)

In his tutorial, Shigeru Chiba presents the techniques of *reflection* and *aspect-oriented programming* (AOP) to support program transformation. More specifically, he shows how these techniques can be used to write program translators that facilitate the way in

which software developers can use complicated application frameworks or component libraries.

Seen from a refactoring point of view, such a program translation can effectively be seen as a dedicated, domain-specific, program refactoring. The main difference with the traditional refactorings proposed in literature (e.g. Martin Fowler's refactoring catalog [7]) is that such program translators are not applicable outside the application framework for which they are built.

Seen from a technological point of view, it is true that reflective capabilities of programming languages are crucial to build tools that support program refactoring (such as, for example, the Smalltalk refactoring browser [44]). The kind of refactoring support that can be provided depends of course on specific properties of the programming language of interest, such as which kind of reflection is supported by the language. Even within the same programming language, different variants of refactoring techniques can be envisioned. For example, Smalltalk's refactoring browser can be considered as a static tool in the sense that it only modifies the static class structure. However, with the same reflective techniques, and considerably more effort, it is also possible to build tools that support dynamic (in the sense of runtime) evolution and refactoring. As an illustration of this, we refer to [45,46,47].

Concerning the technology of AOP, it remains to be seen if and how it can provide support for program refactoring. In addition, the introduction of AOP languages such as *AspectJ* [48] also raises new important questions, such as:

aspect identification or aspect mining: how can one detect croscutting concerns in the program that are good candidates for turning them into aspects?

aspect introduction: how can one transfrom (read: refactor) a traditional program into a program containing aspects

aspect refactoring: how can one apply refactoring techniques to aspect-oriented programs?

For more details on these issues, we refer to [49].

Let us conclude this discussion with a final important question: can reflective techniques also be used to support *model refactoring* (as opposed to *program refactoring*). In principle, the answer to this question would be positive, provided that the modeling language is reflective. A prerequisite for this seems to be that models should be first-class entities in the language, and model transformations should be models too [2].

7 Conclusion

In this tutorial we explored the idea of model refactoring (using a simplified version of UML class diagrams as our metamodel), and we explained how the formalism of graph transformation can be used as an underlying foundation. More in particular, we provided some concrete experiments to show how graph transformation technology can be used to support model refactoring.

Using the graph transformation tool *Fujaba*, we explained how a refactoring plug-in can be developed to refactor UML class diagrams, where each refactoring is expressed as a graph production. Using the graph transformation tool *AGG*, we explained how to

use the built-in technique or critical pair analysis to detect potential conflicts between refactorings, and to help the developer decide which refactoring should be selected when different choices are applicable.

References

1. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, September-October 2003. Special Issue on Model-Driven Software Development.
2. Jean Bézivin. Model driven engineering: Principles, scope, deployment and applicability. In *Proceedings of 2005 Summer School on Generative and Transformation Techniques in Software Engineering*, 2005.
3. Jonathan Sprinkle, Aditya Agrawal, Tíhamer Levendovszky, Feng Shi, and Gabor Karsai. Domain model translation using graph transformations. In *Proc. Int'l Conf. Engineering of Computer-Based Systems*, pages 159–168. IEEE Computer Society, 2003.
4. A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In *Proc. Model-Driven Architecture: Foundations and Applications*, pages 14–28, 2004.
5. György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - visual automated transformations for formal verification and validation of UML models. In *Proc. 17th Int'l Conf. Automated Software Engineering*, pages 267–270. IEEE Computer Society, 2002.
6. William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
7. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
8. G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquel. Refactoring UML models. In *Proc. UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–138. Springer-Verlag, 2001.
9. Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In *Proc. 3rd Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering*, pages 77–81, 2002. Alghero, Sardinia, Italy.
10. Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158. Springer-Verlag, 2003.
11. Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. Supporting model refactorings through behaviour inheritance consistencies. In Ana Moreira Thomas Baar, Alfred Strohmeier, editor, *UML 2004 - The Unified Modeling Language*, volume 3273 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, October 2004.
12. Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development - Research and Practice in Software Engineering*. Springer Verlag, 2005.
13. Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *Proc. AGTIVE 99*, volume 1779 of *Lecture Notes in Computer Science*, pages 481–488. Springer-Verlag, 1999.
14. Jörg Niere and Albert Zündorf. Testing and simulating production control systems using the Fujaba environment. In *Proc. AGTIVE 99*, volume 1779 of *Lecture Notes in Computer Science*, pages 449–456. Springer-Verlag, 1999.

15. Jörg Niere and Albert Zündorf. Using Fujaba for the development of production control systems. In M. Nagl, A. Schürr, and M. Münch, editors, *Proc. Int. Workshop Agtive 99*, volume 1779 of *Lecture Notes in Computer Science*, pages 181–191. Springer-Verlag, 2000.
16. Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. Technical report, Université de Mons-Hainaut, 2005.
17. Claudia Werner Alexandre Correa. Applying refactoring techniques to uml/ocl models. In Ana Moreira Thomas Baar, Alfred Strohmeier, editor, *UML 2004 - The Unified Modeling Language*, volume 3273 of *Lecture Notes in Computer Science*, pages 173–187. Springer-Verlag, October 2004.
18. Ivan Porres. Model refactorings as rule-based update transformations. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 159–174. Springer-Verlag, 2003.
19. G. Spanoudakis and A. Zisman. *Handbook of Software Engineering and Knowledge Engineering*, chapter Inconsistency management in software engineering: Survey and open research issues, pages 329–380. World scientific, 2001.
20. Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logics to maintain consistency between UML models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer-Verlag, 2003.
21. Serge Demeyer, Dirk Janssens, and Tom Mens. Simulation of a LAN. *Electronic Notes in Theoretical Computer Science*, 72(4), 2002.
22. Andrea Corradini, Ugo Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3 and 4):241–265, 1996.
23. Hartmut Ehrig and Michael Löwe. Parallel and distributed derivations in the single-pushout approach. *Theoretical Computer Science*, 109:123–143, 1993.
24. Hartmut Ehrig and Annegret Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer-Verlag, 1986.
25. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, June 1996.
26. A. Schürr, A.J. Winter, and A. Zündorf. *Handbook of Graph Grammars and Graph Transformation*, chapter PROGRES: Language and Environment, pages 487–550. World scientific, 1999.
27. Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. 1st Int'l Conf. Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2002.
28. Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing refactorings as graph rewrite rules on a platform independent metamodel. In *Proc. Fujaba Days*, 2003.
29. Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. In *Proc. Int'l Workshop Software Evolution through Transformations (SETra)*, 2004. To appear in ENTCS.
30. Hans Schippers and Pieter Van Gorp. Standardizing sdm for model transformations. In *Proc. 2nd Int'l Fujaba Days*, September 2004.
31. Tom Mens, Gabriele Taentzer, and Olga Runge. Analyzing refactoring dependencies using graph transformation. *Software and System Modeling*, February 2005. Submitted.
32. Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proc. 1st Int'l Conf. Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.
33. Niels Van Eetvelde and Dirk Janssens. Extending graph rewriting for refactoring. In *Proc. 2nd Int'l Conf. Graph Transformation*, volume 3526 of *Lecture Notes in Computer Science*, pages 399–415. Springer-Verlag, 2004.

34. Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
35. Mel Ó Cinnéide and Paddy Nixon. Composite refactorings for java programs. Technical report, Department of Computer Science, University College Dublin, 2000.
36. Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004.
37. Reiko Heckel. Algebraic graph transformations with application conditions. Master's thesis, Technische Universität Berlin, 1995.
38. Paolo Bottoni, Francesco Parisi Presicce, and Gabriele Taentzer. Specifying integrated refactoring with distributed graph transformations. *Lecture Notes in Computer Science*, 3062:220–235, 2003.
39. Eelco Visser. A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–. Springer-Verlag, 2001.
40. Mark van den Brand, Paul Klint, and Jurgen Vinju. Term rewriting with traversal functions. *Transactions on Software Engineering and Methodology*, 12:152–190, 2003.
41. J. Banerjee and W. Kim. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. ACM SIGMOD Conference*, 1987.
42. R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems*, 2(3), 1977.
43. Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–162, February 2004.
44. Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
45. Peter Ebraert, Theo D'Hondt, and Tom Mens. Enabling dynamic software evolution through automatic refactoring. In Ying Zhou and James R. Cordy, editors, *Proc. 1st Int'l Workshop on Software Evolution Transformations*, pages 3–6, November 2004.
46. R. Hirschfeld, K. Kawamura, and H. Berndt. Dynamic service adaptation for runtime system extensions. In M. Conti R. Battiti, R. lo Cigno, editor, *Wireless On-Demand Network Systems*, volume 2928 of *Lecture Notes in Computer Science*, pages 225–238. Springer-Verlag, February 2004.
47. Robert Hirschfeld and Ralf LŁmmel. Reflective designs. *IEE Journal on Software, Special Issue on Reusable Software Libraries*, 152(1), February 2005.
48. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. European Conf. Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.
49. Tom Mens, Kim Mens, and Tom Tourwé. Aspect-oriented software evolution. *ERCIM News*, (58):36–37, July 2004.