Electronic Communications of the EASST
Volume 8 (2008)

Proceedings of the
Third International ERCIM Symposium on
Software Evolution
(Software Evolution 2007)

Refactoring of UML models using AGG

Alessandro Folli[1], Tom Mens

15 pages

Guest Editors: Tom Mens, Ellen Van Paesschen, Kim Mens, Maja D'Hondt
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

[1] This paper reports on work that has been carried out by the first author in the context of a masters thesis supervised by the second author in the context of an ERASMUS exchange programme.

Refactoring of UML models using AGG

Alessandro Folli[2], **Tom Mens**

Service de Génie Logiciel, Université de Mons-Hainaut, Belgique

Abstract: Model refactoring is an emerging research topic that is heavily inspired by refactoring of object-oriented programs. Current-day UML modeling environments provide poor support for evolving UML models and applying refactoring techniques at model level. As UML models are intrinsically graph-based in nature we propose to use graph transformations to specify and apply model refactoring. More in particular, we use a specific graph transformation tool, AGG, and provide recommendations of how AGG may be improved to better support model refactoring. These recommendations are based on a small experiment that we have carried out with refactoring of UML class diagrams and state machines.

Keywords: UML, model refactoring, AGG, graph transformation

1 Introduction

Model-driven engineering (MDE) is a software engineering approach that promises to accelerate development, to improve system quality, and also to enable reuse. Its goal is to tackle the complexity of developing, maintaining and evolving complex software systems by raising the level of abstraction from source code to models. The mechanism of *model transformation* is at the heart of this approach, and represents the ability to transform and manipulate models [SK03].

Model transformation definition, implementation and execution are critical aspects of this process. The challenge goes beyond having languages to represent model transformations. The transformations also need to be reused and to be integrated into software development methodologies and development environments that make full use of them.

The term *refactoring* was originally introduced by Opdyke in his seminal PhD dissertation [Opd92] in the context of object-oriented programming. Martin Fowler [Fow99] defines this activity as “the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure”.

Current-day refactoring techniques focus primarily on the source code level and do not take into account the earlier stages of design. A need exists for refactoring tools that enable designers to better manipulate their model, not just their source code. Furthermore, there is a need to synchronise and maintain consistency between models and their corresponding code; source code refactorings may need to be supplemented with model-level refactoring to ensure their consistency. This article will focus on the problem of *model refactoring*, which is a particular kind of model transformation.

The *Unified Modeling Language* (UML) [Obj05] is the industry standard used to specify, visualize, and document models of software systems, including their structure and design. Therefore,

[2] This paper reports on work that has been carried out by the first author in the context of a masters thesis supervised by the second author in the context of an ERASMUS exchange programme.

the goal of this article is to explore the refactoring of UML models. We use graphs to represent UML models, and graph transformations to specify and apply model transformations. This choice is motivated by the fact that graphs are a natural representation of models that are intrinsically graph-based in nature (e.g., class diagrams, state machine diagrams, activity diagrams, sequence diagrams).

Graph transformation theory has been developed over the last three decades as a suite of techniques and tools for formal modelling and very high-level visual programming [Roz97, EEKR99, EKMR99]. It allows to represent complex transformations in a compact visual way. Moreover, graph transformation theory provides a formal foundation for the analysis and the automatic and interactive application of model transformations. Among others, it provides the ability to formally reason about such graph transformations, for example to analyse parallel and sequential dependencies between rules.

In this article, we argue that the use of graph transformations for the purpose of model refactoring is both possible and useful. As a proof of concept, we implement a number of complex model refactorings in AGG[3] [Tae04]. It is a rule-based visual programming environment supporting an algebraic approach to graph transformation [EM93]. AGG may be used as a general-purpose graph transformation engine in high-level Java applications employing graph transformation methods. AGG is also one of the rare tools that incorporates mechanisms such as critical pair analysis for formally analysing graph transformations, which can be very useful for analysing refactoring rules [MTR07, Men06].

Based on our experience, we provide recommendations on how the AGG tool, and graph transformation tools in general, may be improved.

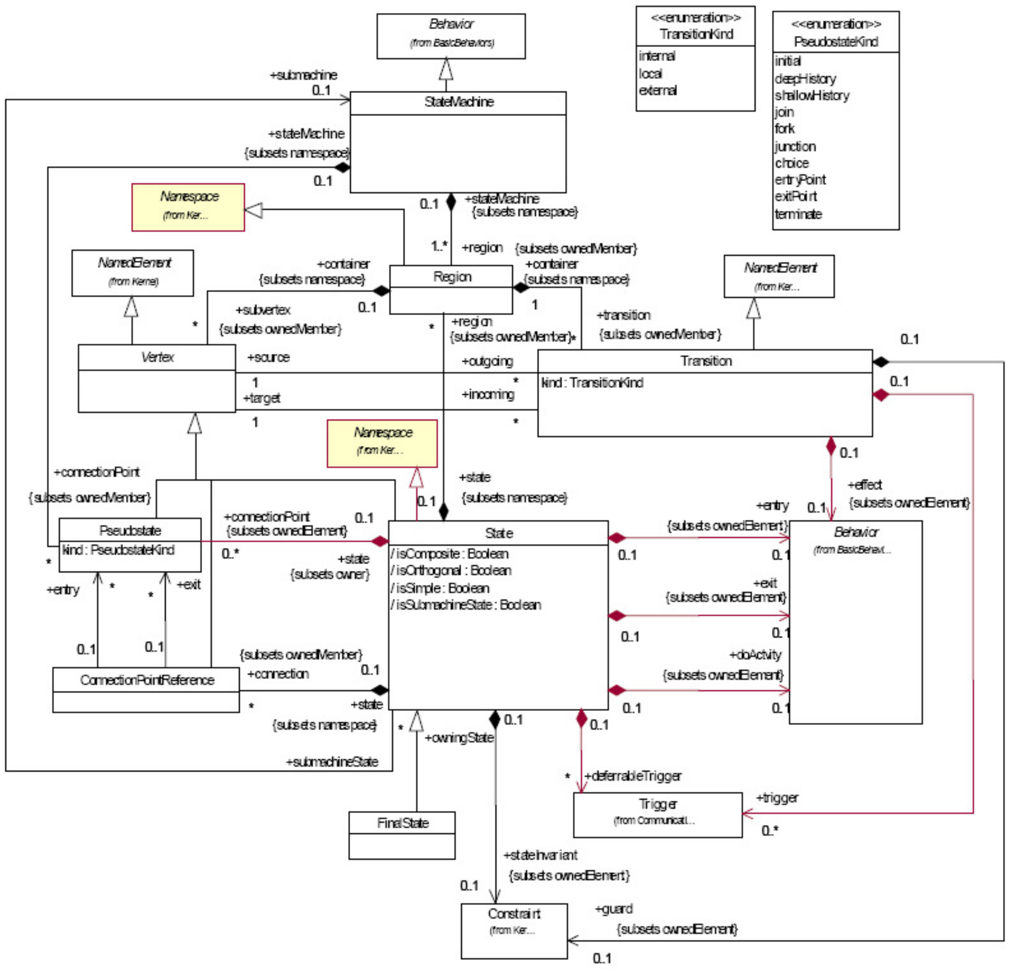
2 Motivating Example

In the field of software engineering, the Unified Modeling Language (UML) defined by the Object Management Group (OMG) [Obj05] is the *de facto* industry standard specification language for modelling, specifying, visualising, constructing, and documenting software-intensive systems. UML provides a standardized graphical notation to create abstract models of a system, referred to as UML models. These models must be conform to the UML metamodel which describes the syntax and semantics. As an example, Figure 1 shows the metamodel of UML state machine diagrams.

Model refactoring is a special kind of model transformation that aims to improve the structure of the model, while preserving (certain aspects of) its behaviour. Like the process of source code refactoring, the process of model refactoring is a complex activity. A definition of refactoring has been introduced by *Opdyke* in his PhD dissertation [Opd92]. He defines refactorings as program transformations containing particular preconditions that must be verified before the transformation can be applied.

In [Fol07] we have discussed eight primitive model refactorings for UML Class diagrams and UML State Machine diagrams. This clearly shows that it is possible to formalise the specification and execution of model refactoring using graph transformation rules. Table 1 shows the list of model refactorings that we have discussed and implemented. Each model refactoring was

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



Behavior (from BasicBehaviors)
<<enumeration>> TransitionKind
internal
local
external
<<enumeration>> PseudostateKind
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
StateMachine
+submachine 0..1
+stateMachine {subsets namespace} 0..1
Namespace (from Ker...
NamedElement (from Kernel)
Region
+region {subsets ownedMember} 1..*
+container {subsets namespace}
+subvertex {subsets ownedMember}
+transition {subsets ownedMember}
NamedElement (from Ker...
Vertex
+source
+target
+outgoing
+incoming
Transition
kind : TransitionKind
+connectionPoint {subsets ownedMember}
Pseudostate
kind : PseudostateKind
+connectionPoint {subsets ownedElement}
State
/ isComposite : Boolean
/ isOrthogonal : Boolean
/ isSimple : Boolean
/ isSubmachineState : Boolean
+state {subsets owner}
+entry
+exit
+doActivity
{subsets ownedElement}
+effect {subsets ownedElement}
Behavior (from BasicBehavi...
ConnectionPointReference
+connection {subsets ownedMember}
+state {subsets namespace}
+submachineState
+owningState
+deferrableTrigger
Trigger (from Communicati...
+trigger
FinalState
+stateInvariant {subsets ownedElement}
Constraint (from Ker...
+guard {subsets ownedElement}

Figure 1: UML metamodel of state machines

formalised, explained and motivated using a concrete example. A detailed explanation of when it should be used and how it can be realised precedes a discussion of the list of mechanisms to accomplish the refactoring itself.

In this article, we explore the *Introduce Initial Pseudostate* model refactoring in more detail in order to explain and illustrate the main concepts.[4] As suggested by the name, it adds an initial pseudostate to a composite state, or region. The *Introduce Initial Pseudostate* refactoring is used to improve the structure of a State Machine diagram. In general, it is a good convention not to cross boundaries of a composite state.

Figures 2 and 3 show a simple example of using this kind of refactoring. An initial pseudostate

[4] For reasons of simplicity, the representation of UML state machines used for this article does not consider the actions attached to states, such as *do*, *entry* and *exit* actions.

UML Class diagram	**UML State Machine diagram**
Pull Up Operation	***Introduce Initial Pseudostate***
Push Down Operation	*Introduce Region*
Extract Class	*Remove Region*
Generate Subclass	*Flatten State Transitions*

Table 1: List of model refactorings

has been added to the *ACTIVE* composite state. The target of the transition that initially referred to the *Ready* state has been redirected to its enclosing region. An automatic transition has been defined between the initial pseudostate and the *Ready* state. The *Ready* state has thus become the default initial state of the *ACTIVE* region; a transition whose target is the *ACTIVE* state will lead the State Machine to the *Ready* state. When the refactoring is used to introduce a final state as well, similar changes to the transitions involved in the composite state will need to take place.

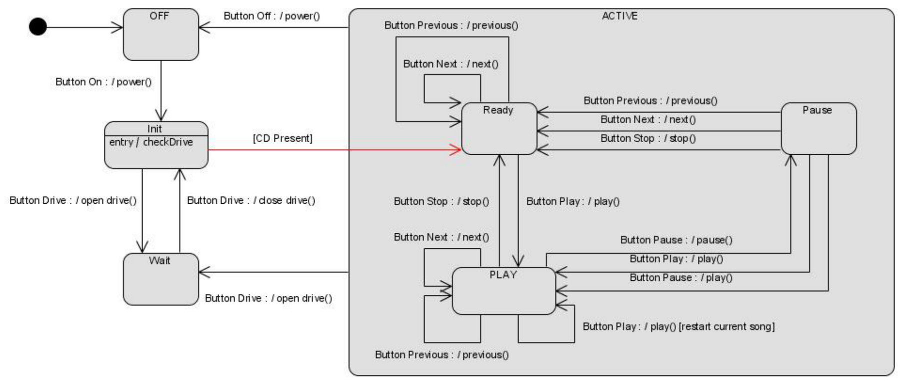


Figure 2: UML state machine diagram - Before refactoring

Figure 4 shows the control flow of this model refactoring that is composed of more primitive refactoring actions. The notation of *UML Interaction Overview diagrams* has been used to formally depict the control flow and to specify in which order the action must be executed. The *interaction occurrence frames* that compose the diagram indicate activities or operations to be invoked. For the purpose of defining model refactoring, they have been associated to graph transformation rules. Some custom notations have been added to enrich the diagram with all necessary information. In particular, input and output parameters for each atomic step have been specified.

In order to apply the refactoring, it is necessary to provide two input parameters *r* and *s*. The parameter *r* specifies which composite state, or region, will be modified by the refactoring. The parameter *s* specifies which will be the default state of the region. Before applying the refactoring it is necessary to verify that the composite state does not contain an initial pseudostate; this check

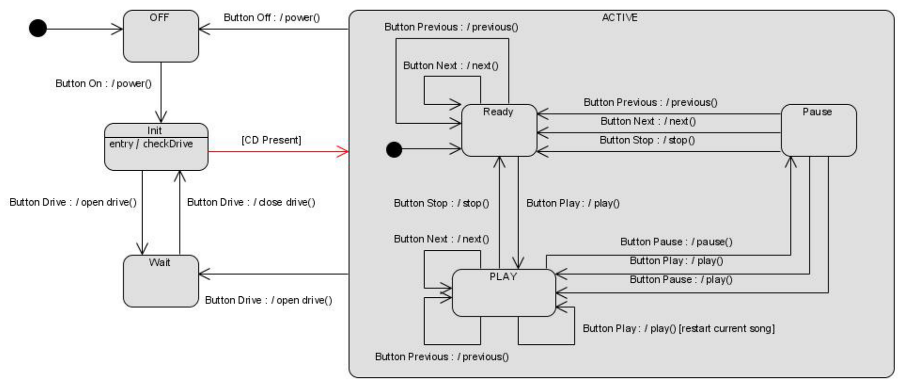


Figure 3: UML state machine diagram - After refactoring

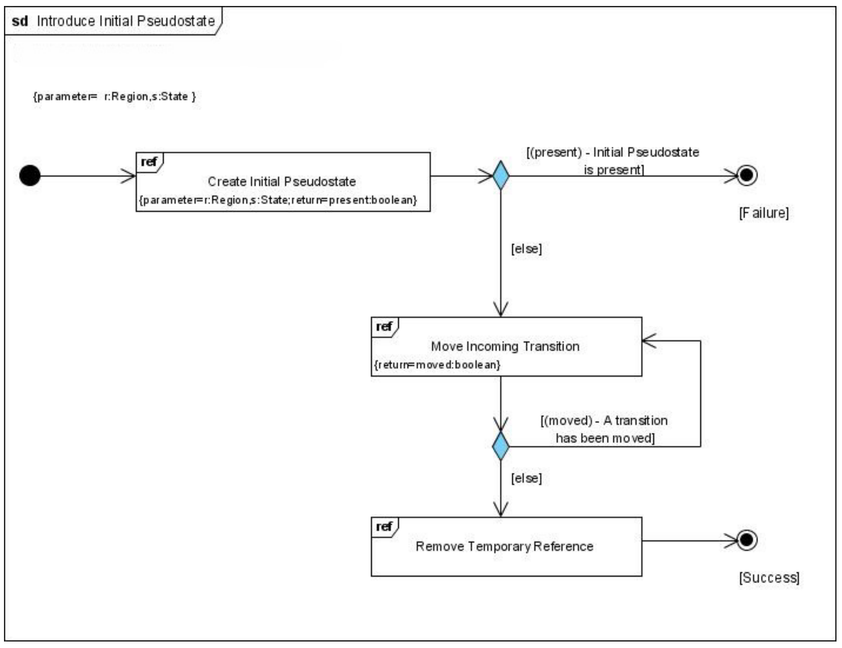

sd Introduce Initial Pseudostate
{parameter= r:Region,s:State }
ref
Create Initial Pseudostate
{parameter=r:Region,s:State;return=present:boolean}
[(present) - Initial Pseudostate is present]
[Failure]
[else]
ref
Move Incoming Transition
{return=moved:boolean}
[(moved) - A transition has been moved]
[else]
ref
Remove Temporary Reference
[Success]

Figure 4: Introduce Initial Pseudostate model refactoring, specified as UML Interaction Overview diagram

will be implemented as a precondition.

If the precondition is respected, the refactoring proceeds by creating the initial pseudostate inside the composite state. Subsequently, the refactoring changes the target of all transitions pointing to the default state. The new target state of those transitions will become the composite state that contains the region *r*.

For technical reasons, a final cleanup phase is needed in order to remove auxiliary elements that have been added during the transformation process.

3 Formal representation as graph transformation

UML models can be represented as a graph-based structure, and graphs must conform to the corresponding *type graph*, much in the same way as models must conform to their *metamodel*. A *type graph* corresponding to the *UML metamodel* is required to formally represent the UML models as graphs and to formally define the UML model refactoring. [5]

For the purpose of this article, we have chosen to take into account a subset of the concepts defined by the UML metamodel. In particular, we focus on *UML State Machine diagrams* only. Figure 5 shows the *type graph* corresponding to the UML metamodel of figure 1. This type graph has been created using the AGG graph transformation tool.

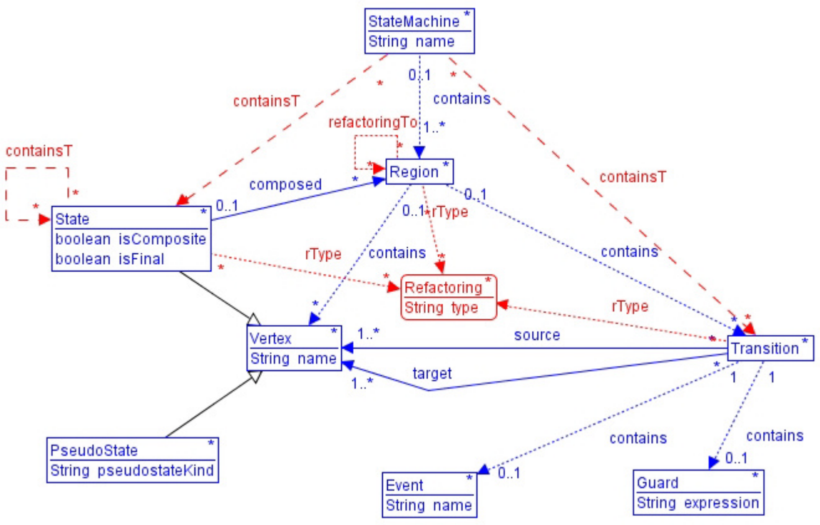


Figure 5: UML State Machine diagram – Type Graph

AGG offers many concepts that are useful to define a *type graph* very similar to the corresponding UML metamodel. AGG allows enrichment of the *type graph* with a generalisation relation between nodes, and each node type can have one or more direct ancestors (parents) from which it inherits the attributes and edges. Moreover, it is also possible to define a node type as an abstract type, thereby prohibiting creation of instance nodes of this abstract type.

The primitive refactoring actions shown in figure 4 can be implemented by means of graph transformation rules. Transformation rules are expressed mainly by two object structures: the left-hand side (LHS) of the rule specifies a subgraph to search for, while the right-hand side (RHS) describes modifications generated by the transformation. The LHS and the RHS of a rule are related by a partial graph morphism. The applicability of a rule can be further restricted by

[5] For a detailed account on the relation between refactoring and graph transformation, we refer to [Men06].

additional negative application conditions (NACs). The LHS or a NAC may contain constants or variables as attribute values, but no Java expressions, in contrast to an RHS.

In order to link all these primitive refactoring actions together (as specified in figure 4), we need to ressort to some kind of controlled (or programmed) graph transformation mechanism. Tools like Fujaba offer this possibility by relying on the notion of so-called story diagrams [GZ04]. AGG, unfortunately, does not support controlled application of graph transformation rules, so we were forced to implement such a mechanism ourselves, as will be explained in section 4.

The first primitive refactoring, named *Create Initial Pseudostate* is shown in figure 6. It contains a NAC to ensure that the region does not contain an initial pseudostate. The state *s* provided as input parameter (node number 3 in the figure) will become the default state of the region. The *strName* variable used in the rule is not an input parameter but is needed in order to define the name of the initial pseudostate. The state *s* provided as input parameter must be part of a composite state otherwise application of this kind of refactoring is no longer possible. If the precondition is respected, the transformation rule marks the default state with an auxiliary "Refactoring" node in order to recognize it during the execution of the subsequent steps.

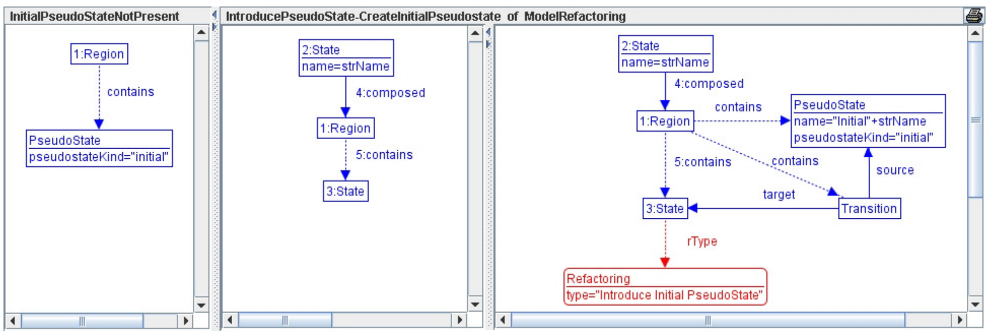


Figure 6: Introduce Initial Pseudostate - Create Initial Pseudostate

Input Parameters $r : Region \Rightarrow node1; \quad s : State \Rightarrow node3$

The second primitive refactoring step, named *Move Incoming Transition*, is shown in figure 7. It takes into account the transitions which have the default state defined as target (the auxiliary "Refactoring" node is used to identify the default state). The transformation rule replaces the target edge of a transition by one pointing to the composite state. For this transformation rule a NAC has been added in order to ensure that only the transitions that are defined outside the region will be modified. The rule must be repeated as long as possible (i.e., until no further match can be found).

The last step, named *Remove Temporary Reference* is shown in figure 8. It removes the auxiliary "Refactoring" node that has been attached to the default state during the execution of the first rule.

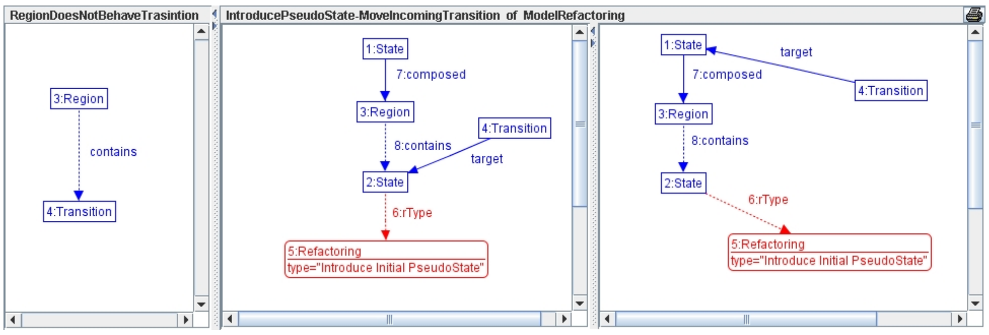
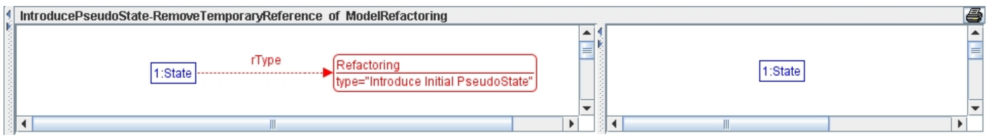


Figure 7: Introduce Initial Pseudostate - Move Incoming Transition



IntroducePseudoState-RemoveTemporaryReference of ModelRefactoring
1:State
rType
Refactoring
type="Introduce Initial PseudoState"
1:State

Figure 8: Introduce Initial Pseudostate - Remove Temporary Reference

4 Tool support

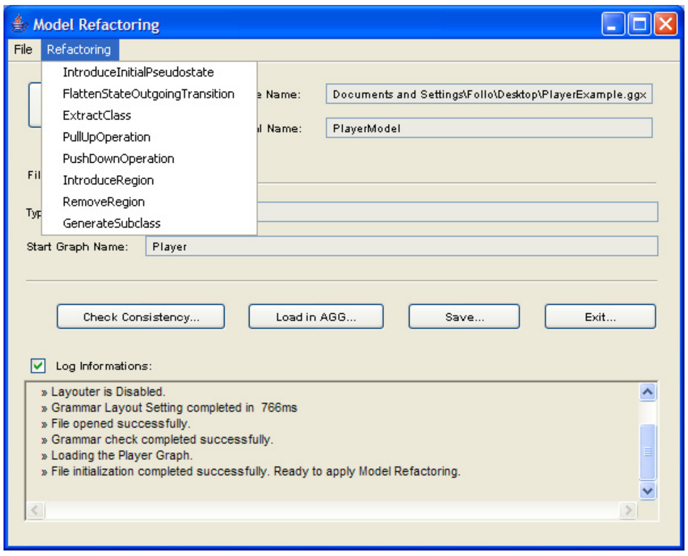
In this section, we illustrate the feasibility of developing model refactoring tools using graph transformations. For this purpose, we have developed a prototype application in AGG. The choice of AGG was motivated by its good support for formal analysis techniques.

Based on our experience with implementing this prototype, we will discuss the current limitations of AGG and graph transformation in general in section 5.

The AGG graph transformation engine is delivered together with an API (Application Programming Interface) that allows to integrate the internal graph transformation engine into other environments. We used this API to develop our prototype application. This allowed us to specify the graph transformation rules of section 3, as well as the control flow specified in figure 4.

Figure 9 shows the graphical user interface of the model refactoring application that we developed in Java by making use of the AGG API. The application internally loads a file containing the model refactoring specifications and the necessary graph transformation rules. It then allows to open files containing the UML models to be refactored that respect the *type graph*. Using the "Refactoring" context menu, the user can apply the different model refactorings. When necessary the user will be prompted to enter the input parameter values and possibly to supply a match if the model refactoring can be applied to different parts of the UML model.

The representation of the control flow explained in figure 4 has been a crucial point for the implementation of the prototype application. The control flow describes the order in which the individual graph transformation rules of each model refactoring have to be executed. At the moment, the AGG tool does not provide a satisfactory solution for organizing and combining rules, and the supplied mechanisms were not sufficient for describing model refactorings. The



e Name: Documents and Settings\Follo\Desktop\PlayerExample.ggx

l Name: PlayerModel

Start Graph Name: Player

```
» Layouter is Disabled.
» Grammar Layout Setting completed in  766ms
» File opened successfully.
» Grammar check completed successfully.
» Loading the Player Graph.
» File initialization completed successfully. Ready to apply Model Refactoring.
```

Figure 9: Model Refactoring Application

prototype application avoids the underlying problem by using a custom control structure that represents the control flow of model refactorings. Based on the UML Interaction Overview diagram syntax, we have represented the control flow as a graph structure, which is used to drive the application of graph transformation rules. Figure 10 presents the *type graph* that we have defined in AGG in order to represent this control flow.

When the prototype application needs to apply a model refactoring, it first loads the corresponding graph representing the control flow. It searches the starting point and walks through the graph to determine which graph transformation rules have to be applied. It continues exploring the graph until it reaches a *final point*, and reports the result to the user.

Figure 11 shows the control flow we have implemented for the *Introduce Initial Pseudostate* refactoring. It corresponds to the UML Interaction Overview diagram reported in figure 4. The prototype application has been enriched with an interpreter in order to evaluate the expression of decision points. That way, implementation of complex transformations is made possible.

5 Discussion

The main goal of this article was to present a practical study of the current limitations of graph transformation technology. Seen in this light, we can use the insights gained during our experiments to discuss some of the limitations of AGG, and of graph transformations in general. These suggestions may be used to advance the state-of-the-art in graph transformation tool support in the future.

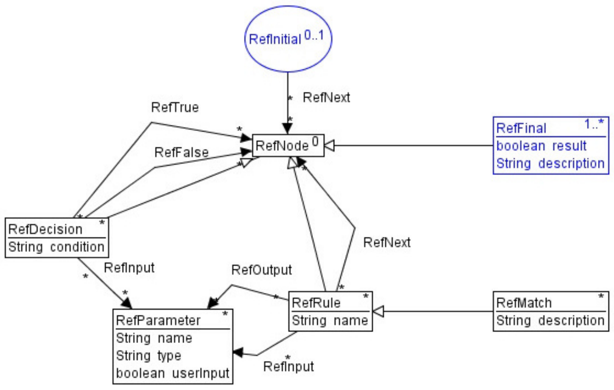


Figure 10: Interaction Overview Diagram – Type Graph

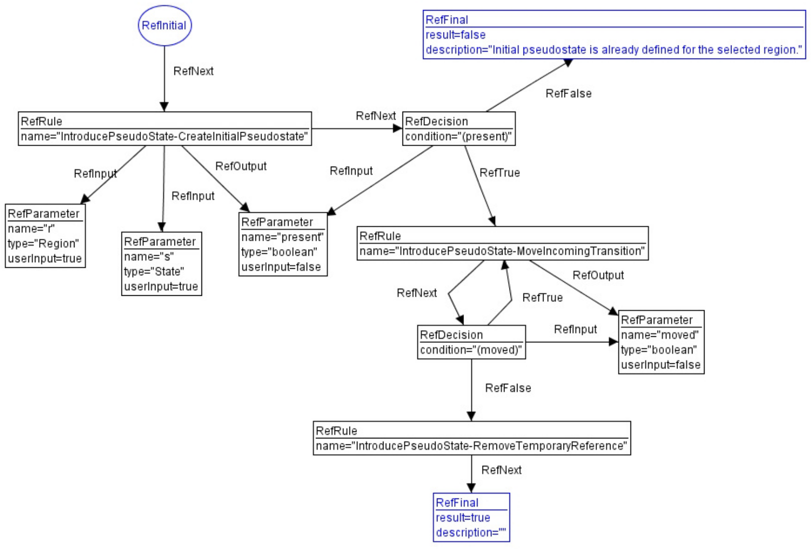


Figure 11: Control Flow – Introduce Initial Pseudostate

5.1 Limitations of AGG

Concerning the AGG tool in particular, we encountered a number of limitations. AGG does not allow to represent concepts like *Aggregation* and *Composition* used by the UML metamodel. Therefore, the *type graph* needed to be simplified by using the more generic concept of association. Moreover, AGG does not have the notion of *Enumeration* type. The property *kind* of the *Pseudostate* element has been represented using a String value. The type graph also did not take into account OCL constraints imposed on the UML metamodel. These constraints typically specify invariants that must hold for the system being modeled. OCL expressions need to be taken into account when specifying model refactorings in AGG. This can be achieved by expressing them using so-called “graph constraints” in AGG. Trying to formalise OCL constraints as graph constraints, however, is a far from trivial task.

AGG does not provide a satisfactory control structure for organizing and combining rules, and the supplied mechanisms for composing rules were not sufficient to describe model refactorings. In order to reach our goal we needed to implement a notion of *“controlled” graph transformation* on top of AGG. This is nothing new, in the sense that other graph transformation tools (e.g. PROGRES, Fujaba, GReAT, VIATRA2) already support such mechanisms.

AGG also does not allow to specify optional patterns inside graph transformation rules. Therefore, it is necessary to create similar graph transformation rules that take into account the different optional patterns. For example, the *“Guard”* nodes may or may not be associated to a transition. In order to match transitions with an associated *“Guard”* and transitions without *“Guard”*, creation of two different graph transformation rules is necessary. Again this limitation is not present in some of the other graph transformation tools around.

Another limitation of AGG is that it does not support concrete domain-specific visual notation. This would be a nice feature, in order to be able to specify model refactorings using a notation that the designer is familiar with. As a solution to this problem, an Eclipse plug-in called *Tiger* is being developed for specifying domain-specific visual editors on top of AGG [EETW06].

Finally, in order to make model refactoring support acceptable to the community, it needs to be integrated in standard UML modeling environments. This could be realised, for example, by relying on the *Tiger EMF transformation framework*. It is a tool environment that allows to generate an Eclipse editor plugin based on the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF) using AGG as underlying transformation engine [BEK$^+$06].

5.2 Limitations of graph transformation tools in general

For the purpose of model refactoring, an important advantage of graph transformation is that rules may yield a concise visual representation of complex transformations. Unfortunately, as identified by [DHJ$^+$06], the current state-of-the-art in graph transformation does not suffice to easily define model refactorings, so their expressive power needs to be increased. Two mechanisms have been proposed by these authors: one for cloning, and one for expanding nodes by graphs. These are mechanisms that allow one to write more generic graph transformation rules.

During our own experiments, we encountered the need for a *reusability mechanism* at the level of transformation rules. In our large set of transformation rules, there were many similarities among the rules, and we would like to have some mechanisms to capture these similarities, in order to write some of the rules in a more compact way, and also in order to be able to reuse (parts of) existing rules when specifying new ones. This reusability can take on many forms: the ability to share application conditions between different rules, the ability to call some rule from within another one (similar to subroutines and procedure calls), the ability to express multiple rules with the same LHS (resp. RHS) without needing to duplicate the LHS (resp. RHS) each time, to avoid redundancy, the ability to define a notion of specialisation between rules, and so on. In VIATRA2, for example, graph patterns can be decoupled and manipulated separately from the graph transformation rules themselves [BV06]. Recently, an even more sophisticated mechanism of recursive graph pattern matching has been introduced [VV07].

Better *modularisation* mechanisms are also needed. Whenever a large set of rules needs to be specified, we need to modularise them in some way. The layering or prioritisation mechanism offered by AGG is too rudimentary for this purpose. Other techniques for structuring transformations are therefore needed. One potentially useful approach could be the use of so-called transformation units [KK99]. Other mechanism are available as well, and a detailed comparison of them has been made by [HEET99]. Integration of such mechanisms in graph transformation tools is starting to emerge. To give but one example, a new grouping operator has been introduced in the model transformation language *GReAT* to enable the concise specification of complex transformations [BNN$^+$07].

An important limitation of current graph transformation tools is that graphs and graph transformations are represented in a different way. Ideally, graph transformations need to be treated as first class values. If graph transformations could be expressed as graphs, we would gain a lot of expressive power. For example, we could specify higher-order transformations, i.e., graph transformations that transform graph transformations. We could also use graph transformations to generate graph transformations from a graph specification (or vice versa). This could for example be very useful in the approach suggested in [EKTW06] to generate an equivalent graph grammar specification for a given input metamodel.

5.3 Alternative graph transformation tools

Many different tools for graph transformation exist. They all have their own specific reason of existence, and can be considered to be complementary in some sense. We already mentioned the PROGRES [SAZ99] and Fujaba tool[6] [NZ00] that offer built-in support for controlled graph transformation. VIATRA2 [BV06] and GREAT [AKN$^+$06] are two other tools that perform sophisticated support for dealing with complex graph transformations. The MOFLON meta modeling framework[7] [AKRS06] is an extension of Fujaba that additionally supports triple graph grammars [KS06], a technique that can be quite useful for synchronising different model views (such as class diagrams and state machines, for example). The MOFLON tool also offers a more standardised way to represent UML models, due to the similarities between the UML metamodel and the MOFLON concepts. ATOM^3 is a domain-specific modeling tool based on graph transformation[8]. As such, it combines the virtues of visual concrete syntax with abstract graph transformations. We could continue our list by discussing other graph transformation tools such as MoTMoT, GROOVE, GrGen, and so on.

6 Conclusion

[MT04] provided a detailed survey of research on software refactoring, and suggested model refactoring as one of the future challenges. In this article, we have shown how the formalism of graph transformation can be used as an underlying foundation for the specification of model

6 http://www.fujaba.de
7 http://www.moflon.org
8 http://atom3.cs.mcgill.ca

refactoring.

We have developed a prototype application in order to verify the feasibility of graph transformations for the purpose of model refactorings. The prototype application shows that it is possible to develop model refactoring tools this way. However, it is necessary to improve current graph transformation tool support in order to better support the specification of model refactorings.

Future work should formally explore the characteristics of model refactoring paying more attention on the preservation of the behaviour. Model refactoring is a rather recent research issue and such definitions of behaviour preservation properties have not yet been completely given. There are some proposals about behaviour preservation but, in the context of the UML, such definitions do not exist because there is no consensus on a formal definition of behaviour.

A UML model is composed of different diagrams that address different aspects of a software system. The application of model refactorings may generate inconsistencies between these UML diagrams. Future work should explore the possibility to preserve the consistency among different kind of UML models after the application of model refactoring expressing inconsistency detections and their resolutions as graph transformation rules. Mens, Van Der Straeten and D'Hondt [MVD06] propose to express inconsistency detection and resolutions as graph transformation rules, and to apply the theory of critical pair analysis to analyse potential dependencies between the detection and resolution of model inconsistencies.

References

[AKN+06] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo. The design of a language for model transformations. *Journal on Software and System Modeling* 5(3):261–288, September 2006.

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Rensink and Warmer (eds.), *Model Driven Architecture - Foundations and Applications: Second European Conference*. Lecture Notes in Computer Science (LNCS) 4066, pp. 361–375. Springer Verlag, Heidelberg, 2006.

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*. Lecture Notes in Computer Science. Springer-Verlag, 2006.

[BNN+07] D. Balasubramanian, A. Narayanan, S. Neema, B. Ness, F. Shi, R. Thibodeaux, G. Karsai. Applying a Grouping Operator in Model Transformations. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Pp. 406–421. Wilhelmshöhe, Kassel, Germany, 2007.

[BV06] A. Balogh, D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proc. 21st ACM Symposium on Applied Computing*. Pp. 1280–1287. ACM Press, April 2006.

[DHJ+06] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, N. V. Eetvelde. Adaptive Star Grammars. In *Proc. Int'l Conf. Graph Transformation (ICGT)*. Lecture Notes in Computer Science 4178, pp. 77–91. Springer Verlag, 2006.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 2: Applications, Languages and Tools. World Scientific, October 1999.

[EETW06] C. Ermel, K. Ehrig, G. Taentzer, E. Weiss. Object-Oriented and Rule-based Design of Visual Languages using TIGER. In *Proc. workshop on Graph-Based Tools (GraBaTs)*. Electronic Communications of the EASST 1. 2006.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 3: Concurrency, Parallelism and Distribution. World Scientific, September 1999.

[EKTW06] K. Ehrig, J. Küster, G. Taentzer, J. Winkelmann. Generating Instance Models from Meta Models. In *8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*. Lecture Notes In Computer Science 4037, pp. 156–170. Springer Verlag, 2006.

[EM93] H. Ehrig, Michael Löwe. Parallel and distributed derivations in the single-pushout approach. *Theoretical Computer Science* 109:123–143, 1993.

[Fol07] A. Folli. UML model refactoring using graph transformation. Master's thesis, Institut d'Informatique, Université de Mons-Hainaut, 2007.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[GZ04] L. Geiger, A. Zündorf. Statechart Modeling with Fujaba. *Electronic Notes in Theoretical Computer Science*, 2004.

[HEET99] R. Heckel, G. Engels, H. Ehrig, G. Taentzer. *Classification and comparison of module concepts for graph transformation systems*. Pp. 669–689. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[KK99] H.-J. Kreowski, S. Kuske. *Graph transformation units and modules*. Pp. 607–638. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[KS06] A. Königs, A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In Heckel (ed.), *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*. Electronic Notes in Theoretical Computer Science 148, pp. 113–150. Elsevier Science Publ., Amsterdam, 2006.

[Men06] T. Mens. On the use of graph transformations for model refactoring. In Ralf Lämmel (ed.), *Generative and transformational techniques in software engineering*. Lecture Notes in Computer Science 4143, pp. 219–257. Springer, 2006.

[MT04] T. Mens, T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Software Engineering* 30(2):126–162, February 2004.

[MTR07] T. Mens, G. Taentzer, O. Runge. Analysing Refactoring Dependencies Using Graph Transformation. *Software and Systems Modeling*, pp. 269–285, September 2007. doi:10.1007/s10270-006-0044-6

[MVD06] T. Mens, R. Van Der Straeten, M. D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In Nierstrasz et al. (eds.), *Model Driven Engineering Languages and Systems*. Lecture Notes in Computer Science 4199, pp. 200–214. Springer-Verlag, October 2006.

[NZ00] J. Niere, A. Zündorf. Using Fujaba for the development of production control systems. In Nagl et al. (eds.), *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Lecture Notes in Computer Science 1779, pp. 181–191. Springer-Verlag, 2000.

[Obj05] Object Management Group. Unified Modeling Language: Superstructure version 2.0. formal/2005-07-04, August 2005.

[Opd92] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1: Foundations. World Scientific, February 1997.

[SAZ99] A. Schürr, Andreas Winter, A. Zündorf. *The PROGRES approach: Language and environment*. Pp. 487–550. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[SK03] S. Sendall, W. Kozaczynski. Model Transformation: The heart and soul of model-driven software development. *IEEE Software* 20(5):42–45, 2003. Special Issue on Model-Driven Software Development.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Lecture Notes in Computer Science 3062, pp. 446–453. Springer-Verlag, 2004.

[VV07] G. Varró, D. Varró. Recursive Graph Pattern Matching. In *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Pp. 453–467. Wilhelmshöhe, Kassel, Germany, 2007.