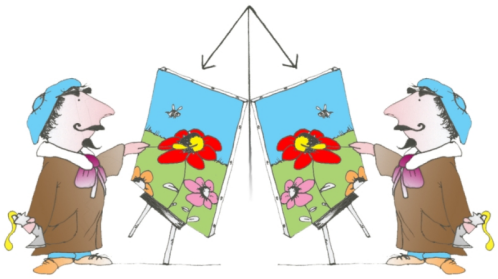
Electronic Notes in Theoretical Computer Science

Graph Transformation and Visual Modeling Techniques

GT-VMT 2006



Vienna, Austria
April 1–2, 2006

Guest Editors:

Roberto Bruni
Dániel Varró

Contents

Preface

This volume contains the proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006), held in Vienna, Austria on April 1 and 2, 2006, as a satellite event to the European Joint Conference on Theory and Practice of Software (ETAPS'06). The GT-VMT workshop series serves as a forum for all researchers and practitioners interested in the use of graph-based notation, techniques and tools for the specification, modeling, validation, manipulation and verification of complex systems. Previous workshops have been organized in Geneva (2000), Crete (2001), Barcelona (2002 and 2004).

Due to the variety of languages and methods used in different domains, the aim of the workshop is to promote engineering approaches that starting from high-level specifications and robust formalizations allow for the design and the implementation of such visual modeling techniques, hence providing effective tool support at the semantic level (e.g., for model analysis, transformation, and consistency management). In fact, the workshop series attracts the interest of communities working on popular visual modeling notations like UML, Graph Transformation, Business Process/Workflow Models.

This year's workshop had an additional focus on Models for Mobile Systems and Services (including Service-Oriented and Global Computing architectures) where huge and highly dynamic graph-like structures offer a challenging ground for the application of graph transformation techniques and tools.

We are very glad to announce that GT-VMT 2006 received a record number of submissions (exactly 40), requiring unexpected efforts from the PC members. In fact, to guarantee the fairness and quality of the selection, each paper received at least three reviews. Given that many submissions proposed very promising ideas and tools, even if not fully developed yet, we decided to include some of them both in the workshop programme, to stimulate the discussion among participants, and in these proceedings, tagged as *short contributions*.

The Program Committee selected 13 regular contributions for presentation plus 10 short contributions. These were grouped in six sessions on "Theory of Graph Transformation", "QVT and Graph transformation", "Verification of Validation", "Models, Code, Metrics", "Programming and Implementation Techniques", "UML and OCL".

Additionally, the programme included two invited talks by

- Jeff Magee (Imperial College, London, UK);
- Jana Koehler (IBM Research, Zurich, Switzerland).

Jana Koehler's lecture is included in these proceedings.

The organizers acknowledge the support by the European Research Training Network SegraVis (Syntactic and Semantic Integration of Visual Modelling Techniques), and by the IST Integrated Project SENSORIA (Software Engineering for Service-Oriented Overlay Computers) funded by the European Union in the 6th framework programme as part of the Global Computing Initiative. We

warmly thank Reiko Heckel and Andrea Corradini for proposing us to organize the workshop in connection with ETAPS 2006 (for the second time, after the venue of ETAPS 2004 in Barcelona). We also thank all our colleagues in the Program Committee and those who helped us as external reviewers for their tremendous efforts in producing their reports under a very tight schedule, that coincided essentially with Christmas holidays. A special thank goes to Silvia Masini for drawing the workshop logo. We are very grateful to the ETAPS organizers, especially to Andreas Krall, for taking care of all the local organization, including the printing of these proceedings and for accommodating all our special requests.

These proceedings will be published in the series Electronic Notes in Theoretical Computer Science (ENTCS). ENTCS is published electronically through the facilities of Elsevier Science B.V. and under its auspices. The volumes in the ENTCS series are available online at `http://www.elsevier.com/locate/entcs`. We are grateful to ENTCS for their support, in particular to Michael Mislove, Managing Editor of the ENTCS series.

April 2006

Roberto Bruni and *Dániel Varró*

Program committee

Marco Aldinucci (ISTI-CNR, Italy)
Paolo Baldan (University of Venice, Italy)
Luciano Baresi (Politecnico di Milano, Italy)
Roberto Bruni (University of Pisa, Italy) **[co-chair]**
Andrea Corradini (University of Pisa, Italy)
Hartmut Ehrig (TU Berlin, Germany)
Gregor Engels (University of Paderborn, Germany)
Reiko Heckel (University of Leicester, UK)
Gabor Karsai (Vanderbilt University, US)
Mark Minas (Universität der Bundeswehr München, Germany)
Francesco Parisi Presicce (George Mason University, US)
Arend Rensink (University of Twente, Netherlands)
Andy Schürr (University of Darmstadt, Germany)
Gabi Taentzer (TU Berlin, Germany)
Dániel Varró (Budapest University of Technology and Economics, Hungary) **[co-chair]**
Martin Wirsing (Ludwig-Maximilians-Universität München, Germany)

List of Referees

As already mentioned, the papers were refereed by the program committee and by several outside referees, whose help is gratefully acknowledged.

András Balogh	Martin Gogolla	Ulrike Prange
Stefano Bistarelli	László Gönczy	Stefan Sauer
Andrea Bracciali	Sam Guinea	Laura Semini
Florian Brieler	Jan Hendrik Hausmann	Tom Staijen
Marzia Buscemi	Frank Hermann	Laurence Tratt
Nadia Busi	Dan Hirsch	Emilio Tuosto
Dario Colazzo	Harmen Kastenberg	Gergely Varró
Juan de Lara	Anneke Kleppe	Robert Wagner
Karsten Ehrig	Barbara König	Jessica Winkelmann
Alexander Förster	Hernán Melgratti	Andreas Winter
Fabio Gaducci	Tony Modica	Corrado Zoccolo

The Role of Visual Modeling and Model Transformations in Business-driven Development

Jana Koehler Rainer Hauser Jochen Küster
Ksenia Ryndina Jussi Vanhatalo Michael Wahler

IBM Zurich Research Laboratory, Business Integration Technologies
Säumerstr. 4, CH-8803 Rüschlikon, Switzerland

Abstract

This paper explores the emerging paradigm of business-driven development, which presupposes a methodology for developing IT solutions that directly satisfy business requirements and needs. At the core of business-driven development are business processes, which are usually modeled by combining graphical and textual notations. During the business-driven development process, business-process models are taken down to the IT level, where they describe the so-called choreography of services in a Service-Oriented Architecture. The derivation of a service choreography based on a business-process model is simple and straightforward for toy examples only—for realistic applications, many challenges at the methodological and technical level have to be solved. This paper explores these challenges and describes selected solutions that have been developed by the research team of the IBM Zurich Research Laboratory.

Key words: Business-process modeling, business-driven development, Service-Oriented Architecture

1 Introduction

An improved alignment of the IT infrastructure of an enterprise with its business needs and requirements is a trend that has dominated and driven innovations in information technology over the past couple of years. Terms such as Web services [24], Service-Oriented Architecture [7], model-driven [23] and agile [18] development, and industry standards such as the Web Service Description Language, WSDL [5], and the Business Process Execution Language,

[1] Email: koe@zurich.ibm.com, http://www.zurich.ibm.com/csc/bit

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

BPEL [1], or recent specifications such as the Service-Component Architecture [2] all relate to this trend.

The new technologies focus on improving the agility of the enterprise software development process to match the pace at which the business needs to change in order to keep up with market trends and competition. A key to improved software development agility is the capability of IT departments to create solutions that directly realize business goals through well-designed business processes.

Business-driven development (BDD) [19] is a methodology for developing IT solutions that directly satisfy business requirements and needs. BDD requires that "a mechanism needs to be devised by which IT efforts are interlocked with business strategy and requirements through an execution framework that is standardized, well understood, and can be executed repeatedly and successfully" [19]. The main phases of such a "mechanism" are illustrated in Figure 1.

The *Model* phase comprises the identification of business goals and requirements and the modeling of the underlying business processes. The business-process models are an essential means to create a link between the business needs and the IT implementations. In the *Develop* phase, the business-process models are refined through a number of transformations until an implementation is obtained that can then be integrated with the existing IT infrastructure in the *Deploy* phase. The resulting deployed solution is *monitored* to measure how it achieves the originally stated business goals. Finally, needs for changes and *adaptation* of the running business processes can be derived and fed back into the original business-process models. In the BDD process, business requirements flow downwards from the business level to the IT level, while IT requirements flow upwards from the IT level to the business level. What sounds so straightforward and easy in theory turns out to be a very challenging endeavor in practice. In this paper, we will focus on the two predominant of the challenges encountered:

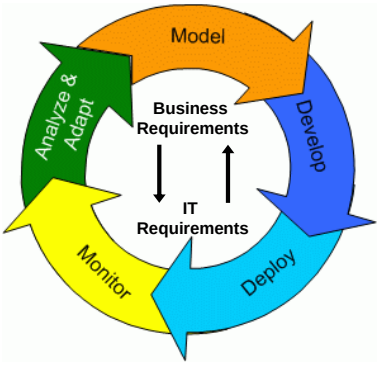


Fig. 1 Main BDD phases.

(i) A business-process model that has been designed with business requirements and goals in mind is not necessarily a model that describes a scalable, reliable, and performant choreography of reusable IT services.

(ii) A service choreography derived in a top-down manner from a business-process model may not so easily or even not at all integrate with an existing IT infrastructure.

The first challenge results from a large gap between the world and perspec-

tive of a business analyst and the realities of today's programming models and software-engineering approaches. The second challenge results from the gap between an ideally designed new solution and the realities of the existing IT infrastructure involving software, hardware, and network topology. The next section will work through a small example to discuss in which form these challenges occur and how they can be addressed.

2 From Analysis Models to Design Models of Business Processes

There seems to be a widely accepted belief that *the* model of a business process exists, i.e., that there is a single model that describes the business process and that this single model is suitable for the business expert as well as the IT expert who is supposed to implement the business process. In our own work, we came to the conclusion that this assumption is very unrealistic. Instead, it is necessary (as it is common today in object-oriented programming) to distinguish between the *analysis model* of a business process and its *design model*, and to develop a methodology that enables the seamless transition from the analysis model to the design model. To illustrate the problem, let us consider the example in Figure 2, which describes a very simplified process for handling insurance claims as it may be initially depicted by a claims specialist. [2] The process consists of three subprocesses *Search Information*, *Record Claim*, and *Settle Claim*.

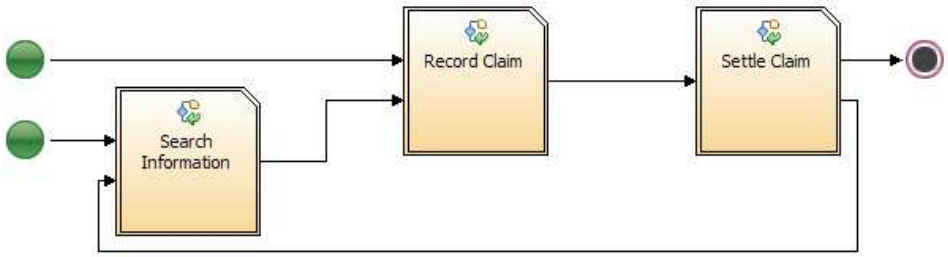


Fig. 2. Analysis model of a simplified claim handling process in insurance.

The intended meaning is described by the claims specialist as follows: "The process starts when a new claim is recorded or by revisiting an existing one to search for additional information on the claim. New information about the claim is recorded, and then the settling of the claim is attempted. The process finishes if the claim was settled, i.e., if either a benefit is payed or the claim is rejected because it is not covered by the insurance policy of the claimant, for example. If a settlement cannot be achieved, e.g., the benefit offered by the

[2] A realistic claim-handling process such as the reference process contained in the IBM Insurance Application Architecture [12] contains about a dozen subprocesses and more than 100 individual process activities.

insurance is not accepted by the claimant, the process has to be resumed by searching for additional information."

The graphical model shown in Figure 2 depicts each of the subprocesses by a rectangle, explicitly denotes start and end points of the process, and shows the control-flow inside the process by connecting the modeling elements with directed edges.[3] Multiple incoming or outgoing edges denote alternative, sequential paths of execution in the process, i.e., there is no parallelism involved in this example. The decisions that select among these alternative execution paths are not explicitly represented in this model. The model is a typical *analysis model* of a business process. An analysis model describes what the process is doing. It shows the initial partitioning of the process into subprocesses and activities with the main flow of control and, optionally, of data. It completely abstracts from IT-related aspects, but can be used for simulation and discussion with business analysts.

However, this model is not yet ready for implementation. First, we do not know which data will drive the process and represent the claim information. Secondly, we have no information on the underlying decision logic or business rules that guide the selection of the alternative execution paths. This information is added to the model in Figure 3, which shows a data-flow model with explicit decision and merge points. The figure shows the initial *design model* for the claim-handling process. In general, a design model contains a refined partitioning of the process that reflects existing application systems and shows an IT-based flow of data and control. It must be ready to be mapped to the desired target programming model. A fully refined design model in addition describes how the process is realized using hardware, software, and people.

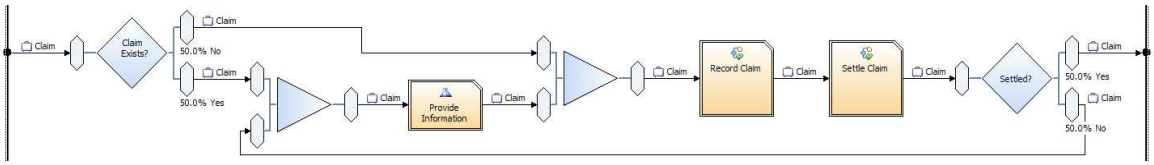


Fig. 3. Initial design model with data-flow, decision and merge points made explicit.

The transition from Figure 2 to Figure 3 is a process that consists of manual and automatic steps, involving the domain expert and an IT or Business architect who is able to work between the business and IT worlds, taking input from both sides:

- In a fully automatic step, the sequential and parallel branching and merging in a process model can be made explicit by adding explicit control actions to the process model such as *Join*, *Fork*, *Decision*, *Merge*, e.g., known from UML activity diagrams [21]. In our example, the original control-flow model is transformed into a control-action normal form, which explicitly adds con-

[3] The figures show screenshots of the example model represented in IBM WebSphere Business Modeler, version 6.

trol actions to the graphical model and restricts activity nodes to having only a single incoming and outgoing edge. Two decision and merge nodes are automatically added to the model, whereas the names of the decisions, *Claim Exists?* and *Settled?*, are added by the user.

- The flow of business information is made explicit in a manual top-down analysis step, e.g., it means that one annotates the input and outputs of the subprocesses with data abstractions such as *claim*, *policy*, *customer information*, and these inputs and outputs are possibly connected to model the data-flow inside the process.
- In a manual bottom-up step, data structures existing in the available IT infrastructure are revisited, which can reveal a reusable data type that can capture the required business information and drive the process. Similarly, reusable services that exist in this infrastructure can be identified to implement parts of the process. In our example, an existing Web service *Provide Information* can be reused to implement the *Search Information* subprocess. This Web service uses a message of type *Claim* that is found suitable for representing all necessary information to drive the claim-handling process.
- In a fully automatic bottom-up step, reusable data types and services can be imported into the business-process modeling tool, where they become available as modeling elements. Service models will usually be added manually to a process model, where they are used to replace or refine existing process activities, whereas a control-flow can be refined fully automatically into a data-flow once the data type has been selected by the user. In our example, the *Search Information* subprocess is replaced by the *Provide Information* service and the *Claim* message type is assigned to the control-flow of Figure 2 to yield the corresponding data-flow model in Figure 3.

The two automatic steps in our example can be implemented as model transformations that transform the business-process model in Figure 2 to the model shown in Figure 3. First, the original control-flow model is transformed into a control-action normal form. Second, a message type *Claim* was selected by the user and then used in an automatic model transformation that changes the control-flow into a data-flow driven by *Claim*. Once the data or message type entering each decision point in the process is known, the decision conditions can be expressed by referring to the attributes and values of the data or message type. The specification of the decision condition based on the process data is a further manual refinement step of the design model.

However, we are still not done. Specific target platforms may further restrict the control- and data-flows of design models that can be mapped to code. Our simple example contains a forward edge from the *Claim Exists?* decision to the *Record Claim* activity and a backward edge from the *Settled?* decision to a merge preceding the *Provide Information* service. The backward edge leads to a cyclic process model. Unfortunately, a language such as BPEL does not support unstructured cycles, but only offers well-structured loops

in the form of *while*-activities [1]. This means, another transformation must be applied that transforms models with unstructured cycles into models with loops. Figures 4 and 5 show the result of such a fully automatic cycle-removal transformation [11,15].

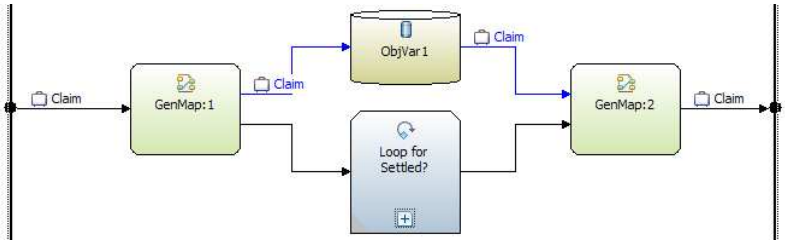


Fig. 4. Design model with loop.

Figure 4 shows a newly created loop activity that was added by the model transformation and encapsulates all those activities of the process that were reachable by the cycle. A *map* activity was added that receives the arriving *Claim* message, puts it into a data store visualized by a repository symbol (which can also represent a variable in the target programming model, e.g., BPEL), and then triggers the loop via a control-flow edge. Once the loop has terminated, another map activity receives the modified *Claim* message and the control-flow from the loop to pass the message on to the process interface.

Figure 5 shows the loop body contents. In our example, all activities have been placed inside a loop and a *Bypass Region* decision has been added to encapsulate the *Provide Information* service, which is only invoked in some of the process executions. The access of the loop body to the data in the repository is not directly visible in the graphical visualization, it is visible only via additional textual attributes, which we do not show here because of space restrictions.

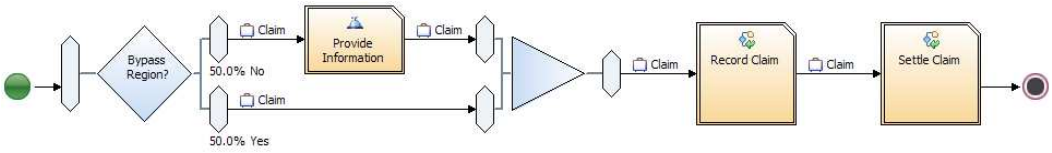


Fig. 5. The loop body containing the repeatable process activities.

Our final example design model is no longer in a representation format that would be ideal for business people. However, only from this model of the process, can now runtime code be generated automatically that meets the requirements of our intended target programming model using BPEL/WSDL. With our transformations, we have made a systematic step-by-step transition from the business view to the IT view of the process, in which each model results from its predecessor through a well-defined, gradual change. The changes were necessary to reuse existing services and data structures and address restrictions of the target programming model. One can easily imagine that

scalability and performance requirements will also significantly influence how the analysis model is refined into the corresponding design model, for example when we have a choice of more than one reusable service or when quality-of-service considerations become important. The example also showed that even a simple scenario requires a sophisticated mixture between manual and automatic, bottom-up and top-down steps.

In the next section, we will review business-driven development from a larger perspective and discuss what methodological and tooling underpinnings are required to make this new style of software development successful.

3 Methodologies and Tools for Business-Driven Development

Figure 6 positions business-driven development as a software development process focusing on business processes and facing a tradeoff between the need to preserve customer investments, while moving towards a modern Service-Oriented Architecture.

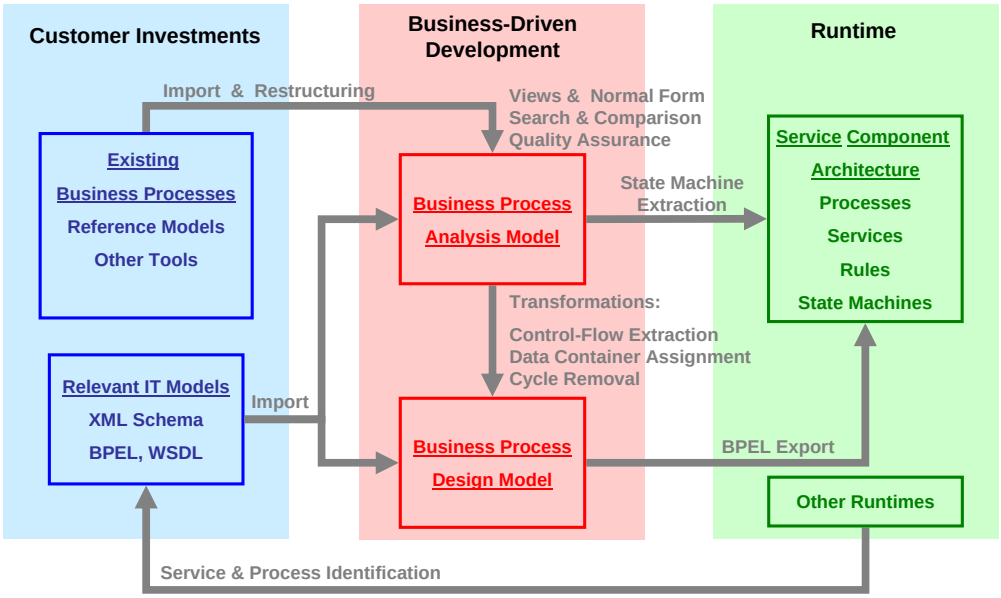


Fig. 6. Business-driven development and contributions by our research team.

In the following, we will use this figure to discuss interesting research problems and briefly review selected contributions made by our research team. The work in Zurich started in 2001 driven by the general question of how software based on Web services should be developed in the future. The work was influenced by trends such as Model Driven Architecture [8] and requirements of business integration. Very quickly, we focused on the problem of generating BPEL from business-process models [14] — an improved and extended BPEL code generation is part of IBM WebSphere Business Modeler

today [16]. We realized quickly that cyclic process models are very natural for business analysts, but their transformation into BPEL is not at all straightforward. Extending our process-model-to-BPEL transformation to a larger class of process models resulted in another model transformation, called *cycle removal* [11,15]. Our team continues to focus on model transformations in the context of business-driven development, and we pursue our work today in two strands.

In one strand, we develop methodologies that underpin business-driven development and make it usable in concrete customer scenarios. At the core of these methodologies is the distinction between four types of process models: the *analysis* and *design models*, which we discussed in the previous section, the usage of *reference models* that describe best practices for an industry, and *legacy process models*, i.e., process models that many customers have built, but that have only been used indirectly as input into a software development process. The investments into these legacy models must be preserved; however, the models must also be further enhanced in their quality to serve as starting points for the generation of high-quality code and architectural solutions. This often requires their *import* into our own tools and a *restructuring* that quite often reveals semantic errors that need to be corrected.

Reference models can help in producing better To-Be analysis models and can also serve to guide the refinement of the analysis model into the design model if a reference model contains not only process models but also ready-to-use service components and data models as it is for example the case for the IBM Insurance Application Architecture [12]. The systematic usage of reference models to improve an existing business process is a challenging task when we think beyond toy examples. It requires a comparison of different types of models that can be facilitated for the human expert by using a *normal-form* representation and condensed *process views*. Specific transformations of process models such as *control-flow extraction*, which changes a data-flow model into a control-flow model, *data container assignment*, which changes a control-flow model into a data-flow model, and *cycle removal* can be fully automated. Other transformations that often require refinement and refactoring steps of the current model must be done by the user, but can nevertheless be supported by tools.

The need to interleave bottom-up and top-down steps in the development process leads us directly to the problem of roundtripping for business-process models. The theoretical boundaries for roundtripping between BPEL and an expressive business-process modeling language, for example, are easy to determine. Given the Turing equivalence of both languages, it is in general undecidable whether an arbitrary, e.g., modified, BPEL program is equivalent to a given business-process model. However, incomplete forms of reasoning about model equivalence make a lot of sense in many scenarios. For example, it makes sense to reimport a modified BPEL into a business-process modeling tool and resort to a human expert to compare it with the original

business-process model from which it was generated. This can make it easier to communicate changes from the IT level back to the business and to support bottom-up steps in business-driven development that require the extraction of business-process models from the IT level.

In our methodological work, we develop detailed methodology guidelines that show how manual and automatic steps interleave, when they are done and why. A specific interest of us is the interleaving of bottom-up and top-down development steps, i.e., the problem of how business requirements flow down and how IT requirements flow up. One of the most challenging and important tasks that needs to take place at the business and the IT level is *service and process identification*—finding the right granularity of services and thinking about reuse very early in the modeling and software development process. Today, this still is a quite poorly understood step, rather than a well-understood and teachable skill.

Our methodologies also address specific questions that result from the use of different forms of models. Historically, business processes have mostly been represented by flow diagrams. Today, there is a trend to link them with business objects [20] and business state machines [3]. The semantic relationship between the various forms of models and their role in business-level and programming models are a very interesting and wide field for research.

In our second strand of research, we investigate fundamental questions that occur in our methodologies and tooling, which are often not specific to business-process models. Many academic solutions have been developed to describe model transformations declaratively, e.g., [17], but the industrial reality is usually plain code. To speed up and improve the quality of our own transformation development, we developed a model transformation framework that provides a convenient API to business-process models represented in IBM WebSphere Business Modeler and a set of elementary transformations for reuse.

The testing of our transformations still is a tedious task, and creating in particular the test examples is a huge effort. The efficient generation of model instances that can be influenced by model transformations is another problem of interest to us [6]. Previous approaches either rely on exponential methods [13] or require scripts to be written [10].

Our transformations often contain many lines of code that validate the source and target models of the transformations, i.e., they check whether the model is eligible for the transformation and whether the target model that was produced satisfies a set of given design constraints. The management of these validation and design constraints at the code level creates a huge software maintenance problem, which is even aggravated by the fact that models evolve further with each new version of a software product and that transformations must consider numerous dependencies between models at the business and IT level [4].

Quality assurance for models is another fundamental question that is also

related to the management of design constraints for models. There seems to be an intuitive feeling of what constitutes a good model in contrast to a bad model, but so far no really practical and user-friendly solutions to improve the quality of process models by enforcing design constraints exist. Similarly, scalable (perhaps incomplete) algorithms to detect typical design errors in process models that would lead to poor runtime code are not yet part of any business-process modeling tool.

We are also interested in algorithmic techniques that help to improve the visualization of large process models and the search capabilities of tools to find models in large collections [22]. Finally, the semantics, the definition of normal forms, and the development of algorithms and tools that allow us to compare process models with each other (be they given in the same or in different modeling languages) are also on our research agenda.

IBM's policy of basing its products on the open-source Eclipse platform makes it much easier for us to prototype our transformations and tools as plugins that extend these products, to cooperate with other research teams, and to test out our solutions in customer engagements. Nevertheless, much work remains to be done. Working with models in a tool is still a heavy-weight undertaking today. For example, many of our transformations, which physically produce a new model, can be thought of as *views*, with a view being a transformation that is not persisted [9]. However, generating such views on models, maintaining them and letting the user freely decide whether the view should be persisted as a new model or update an existing one, leads to many technical and theoretical challenges such as maintaining the consistency among several related models.

A possible vision for the future of business-driven development could be a complete fusion of process model and code, instead of keeping physically distinct platform-independent and platform-specific models. The model is the code, in which graphical and textual elements are combined and an initial (perhaps mostly) graphical model is refined until it becomes executable. Refinement and abstraction steps will allow a user to move between different editions of a model and will be supported by quality-ensuring methods.

4 Summary

The paper presents an overview on the paradigm of business-driven development that centers around business processes and aims at a tighter alignment of the IT infrastructure with business needs and requirements. We review the main phases in a business-driven development cycle and then focus on challenges that arise from the need to transform business-process models into executable services within a Service-Oriented Architecture. We discuss a methodology that distinguishes between the analysis model of a business process and its corresponding design model and describe bottom-up and top-down model transformations that we developed to fill gaps in the business-driven devel-

opment cycle. Several open or only partially solved research problems are identified and positioned within the business-driven development paradigm.

References

[1] T. Andrews et al. Business process execution language for web services. http://www.ibm.com/developerworks/webservices/library/ws-bpel, 2002.

[2] BEA Systems, IBM, IONA, Oracle, SAP AG, Siebel Systems, and Sybase. Service component architecture. http://www.ibm.com/developerworks/library/specification/ws-sca, 2005.

[3] D. Chappell. Introducing microsoft windows workflow foundation. MSDN Library, 2005.

[4] S.-K. Chen, H. Lei, M. Wahler, H. Chang, K. Bhaskaran, and J. Frank. A model driven XML transformation framework for business performance management. In *Proceedings of the IEEE Conference on e-Business Engineering*, pages 71–78. IEEE Press, 2005.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. The web services description language WSDL. http://www.ibm.com/software/solutions/webservices/resources.html, 2001.

[6] K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Automatically Generating Instances of Meta Models based on a Graph Grammar Approach. Technical Report 2005–09, Technical University of Berlin, Dept. of Computer Science, November 2005.

[7] T. Erl. *Service-Oriented Architecture: Concept, Technology, and Design.* Prentice Hall, 2005.

[8] D. Frankel. *Model-Driven Architecture: Applying MDA to Enterprise Computing.* Wiley, 2003.

[9] T. Gardner, C. Griffin, J. Koehler, and R. Hauser. A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard. MetaModelling for MDA Workshop, York, England. Also available as an OMG position paper at http://www.omg.org/docs/ad/03-08-02.pdf, 2003.

[10] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling*, 4(4):386–398, 2005.

[11] R. Hauser and J. Koehler. Compiling process graphs into executable code. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE-04)*, volume 3286 of *Lecture Notes in Services and Software*, pages 317–336. Springer, 2004.

[12] IBM Insurance Application Architecture. http://www.ibm.com/industries/financialservices/iaa.

[13] D. Jackson et al. The Alloy modelling language and analyzer. http://alloy.mit.edu.

[14] J. Koehler, R. Hauser, S. Kapoor, F. Wu, and S. Kumaran. A model-driven transformation method. In *Proceedings of the 7th Conference on Enterprise Distributed Object Computing (EDOC-03)*, pages 186–197. IEEE Press, 2003.

[15] J. Koehler, R. Hauser, S. Sendall, and M. Wahler. Declarative techniques for model-driven business process integration. *IBM Systems Journal*, 44(1):47–65, 2005.

[16] R. Kong. Transform WebSphere Business Integration Modeler process models to BPEL. IBM developerWorks article, http://http://www.ibm.com/developerworks/websphere/library/techarticles/0504_kong/0504_kong.html, IBM, 2005.

[17] J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 2006. in press.

[18] R. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[19] T. Mitra. Business-driven development. IBM developerWorks article, http://www.ibm.com/developerworks/webservices/library/ws-bdd, IBM, 2005.

[20] P. Nandi and S. Kumaran. Adaptive business objects: A new component model for business integration. In *Proceedings of the 7th International Conference on Enterprise Information Systems*, pages 179–188, 2005.

[21] Object Management Group (OMG). *The Unified Modeling Language 2.0*, 2005.

[22] B. Srivastava, J. Vanhatalo, and J. Koehler. Managing the life cycle of plans. In *Proceedings of the 17th Innovative Applications of Artificial Intelligence Conference*, pages 1569–1575. AAAI Press, 2005.

[23] M. Völter and T. Stahl. *Model-Driven Software Development : Technology, Engineering, Management*. Wiley, 2006.

[24] O. Zimmermann, M. Tomlinson, and S. Peuser. *Perspectives on Web Services - Applying SOAP, WSDL and UDDI to real-world projects*. Springer, 2003.

Efficient Conflict Detection in Graph Transformation Systems by Essential Critical Pairs

Leen Lambers , Hartmut Ehrig [1,2]

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
Germany

Fernando Orejas [3]

Dept. L.S.I.
Tech. Univ. Catalonia
Barcelona, Spain

Abstract

The well-known notion of critical pairs already allows a static conflict detection, which is important for all kinds of applications and already implemented in AGG. Unfortunately the standard construction is not very efficient. This paper introduces the new concept of essential critical pairs allowing a more efficient conflict detection. This is based on a new conflict characterization, which determines for each conflict occuring between the rules of the system the exact conflict reason. This new notion of conflict reason leads us to an optimization of conflict detection. Efficiency is obtained because the set of essential critical pairs is a proper subset of all critical pairs of the system and therefore the set of representative conflicts to be computed statically diminishes. It is shown that for each conflict in the system, there exists an essential critical pair representing it. Moreover each essential critical pair possesses a unique conflict reason and thus represents each conflict not only in a minimal, but also in a unique way. Main new results presented in this paper are a characterization of conflicts, completeness and uniqueness of essential critical pairs and a local confluence lemma based on essential critical pairs. The theory of essential critical pairs is the basis to develop and implement a more efficient conflict detection algorithm in the near future.

Key words: conflict, confluence, critical pair, graph transformation

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Static conflict detection is a well-known important task for all kinds of rewriting systems especially also for graph transformation systems. To enable a *static* conflict detection the notion of critical pairs was developed at first for hypergraph rewriting [12] and then for all kinds of transformation systems fitting into the framework of adhesive high-level replacement categories [6]. Usually a straightforward way (i.e. directly according to the definition) is used to compute the set of all critical pairs of a graph transformation system. This is very important for all kinds of applications like for example graph parsing [2], conflict detection in graph transformation based modeling [8] [1] and model transformation [3] [4], refactoring [11], etc. Up to now, however, there is almost no theory which allows an efficient implementation of conflict detection. Therefore our paper [9] and this paper concentrate on exactly this subject.

In [9] it was already explained which optimizations lead to a more efficient conflict detection in a graph transformation system. Unfortunately this efficiency could only be obtained for conflicts induced by a pair of rules with one of the rules non-deleting. This is quite a strong restriction, since in particular a lot of conflicts are induced by a pair of deleting rules. Therefore this paper formulates a characterization of conflicts, covering also these kind of conflicts. Moreover this conflict characterization leads us to the identification of the conflict reason of each conflict.

The notion of critical pair introduced in [12], [6] expresses each conflict in its minimal context. In some cases though two different critical pairs express the same kind of conflict. Therefore exploiting the uniqueness of each conflict reason mentioned above, it is possible to further reduce the set of critical pairs to a subset of essential critical pairs. This subset expresses each kind of conflict which can occur in a graph transformation system in a minimal context and moreover in a unique way. This uniqueness property and the constructive conflict reason definition facilitates the optimization of detecting all conflicts of a graph transformation system.

The following sections explain how to characterize conflicts and what the conflict reason is, how we come to the definition of essential critical pairs and which properties they fullfill. Main new results presented in this paper are a characterization of conflicts, completeness and uniqueness of essential critical pairs and a local confluence lemma based on essential critical pairs. More details concerning well-known definitions and new proofs are given in the long version of this paper [10] to show the mature status of the theory. The theory of essential critical pairs is the basis to develop and implement a more efficient conflict detection algorithm in the near future.

[1] Email: leen@cs.tu-berlin.de
[2] Email: ehrig@cs.tu-berlin.de
[3] Email: orejas@lsi.upc.edu

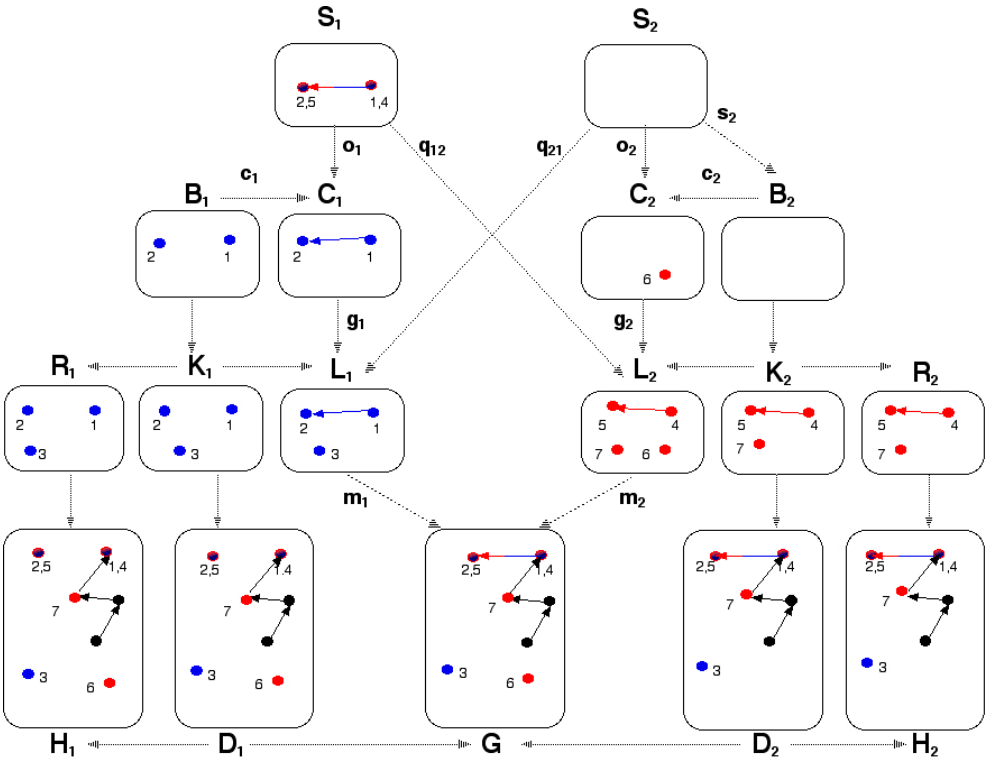
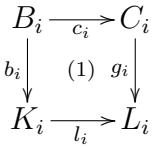


Fig. 1. asymmetrical delete-use-conflict

2 Conflict Characterization and Conflict Reason

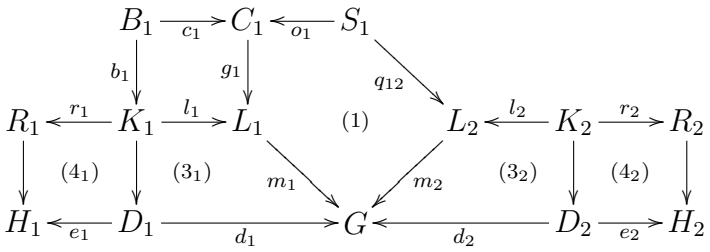
In this section we formulate a theory which leads us to the identification of the conflict reason for each occuring conflict in a graph transformation system where we only consider injective matches. This new notion of conflict reason will help us consequently in the next sections to detect in a static way all representative conflicts of a graph transformation system. At first, we look at an example of two direct transformations $H_1 \overset{p_1,m_1}{\Leftarrow} G \overset{p_2,m_2}{\Rightarrow} H_2$ in conflict in Fig. 1, generated by two deleting rules $p_1 : L_1 \leftarrow K_1 \rightarrow R_1$ and $p_2 : L_2 \leftarrow K_2 \rightarrow R_2$. Looking at both direct transformations we can describe the reason for the conflict between them as follows. The left transformation deletes edge $(1, 4 - 2, 5)$ and that is why rule p_2 can not be applied anymore to the same location on graph H_1. The structure (S_1, o_1, q_{12}), constructed as pullback of $(m_1 \circ g_1, m_2)$, captures exactly the conflict reason for this conflict, because it holds the edge $(1, 4 - 2, 5)$ to be deleted by the left transformation, but used by the other one. The following definitions and theorem explain how to formalize this new notion of conflict reason. Please note, that for all subsequent definitions and theorems the following pair of rules $p_i : L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i$ with boundary B_i and context C_i, defining an inital pushout (1) over l_i (see [6]) and injective graph morphisms b_i, c_i, g_i, l_i are given, i.e. b_i, c_i, g_i, l_i in

M $(i = 1, 2)$, where M is the set of all injective graph morphisms.

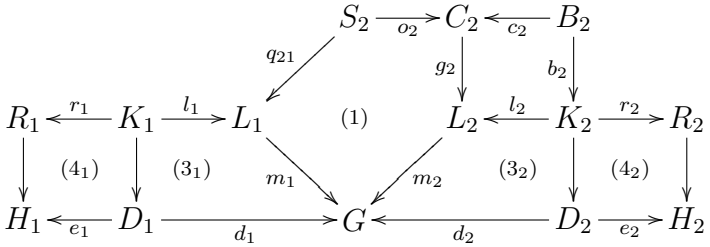


Definition 2.1 [conflict condition] Given a pair of direct transformations $H_1 \overset{p_1,m_1}{\Leftarrow} G \overset{p_2,m_2}{\Rightarrow} H_2$

- $(S_1, o_1 : S_1 \to C_1, q_{12} : S_1 \to L_2)$ the pullback of $(m_1 \circ g_1, m_2)$ satisfies the *conflict condition* if: $\nexists s_1 : S_1 \to B_1 \in M$ such that $c_1 \circ s_1 = o_1$



- $(S_2, q_{21} : S_2 \to L_1, o_2 : S_2 \to C_2)$ the pullback of $(m_1, m_2 \circ g_2)$ satisfies the *conflict condition* if: $\nexists s_2 : S_2 \to B_2 \in M$ such that $c_2 \circ s_2 = o_2$



In the example in Fig. 1 $(S_1, o_1 : S_1 \to C_1, q_{12} : S_1 \to L_2)$ satisfies, but $(S_2, q_{21} : S_2 \to L_1, o_2 : S_2 \to C_2)$ doesn't satisfy the conflict condition. The idea behind this conflict condition is that a conflict occurs if graph parts which are deleted are overlapped with parts to be used by the other transformation. This idea is expressed formally by a new characterization of conflicts in the next theorem.

Theorem 2.2 (Characterization Conflict) *Given a pair of direct transformations $H_1 \overset{p_1,m_1}{\Leftarrow} G \overset{p_2,m_2}{\Rightarrow} H_2$ with $(S_1, o_1 : S_1 \to C_1, q_{12} : S_1 \to L_2)$ the pullback of $(m_1 \circ g_1, m_2)$ and $(S_2, q_{21} : S_2 \to L_1, o_2 : S_2 \to C_2)$ the pullback of $(m_2, m_1 \circ g_1)$ then the following equivalence holds:*

$$H_1 \overset{p_1,m_1}{\Leftarrow} G \overset{p_2,m_2}{\Rightarrow} H_2 \text{ are in conflict}$$

$$\Leftrightarrow$$

$$(S_1, o_1, q_{12}) \vee (S_2, q_{21}, o_2) \text{ satisfies the conflict condition}$$

Theorem 2.2 (proof see [10]) teaches us, that a pair of direct transformations $H_1 \overset{p_1,m_1}{\Leftarrow} G \overset{p_2,m_2}{\Rightarrow} H_2$ is in conflict, because one of the following three

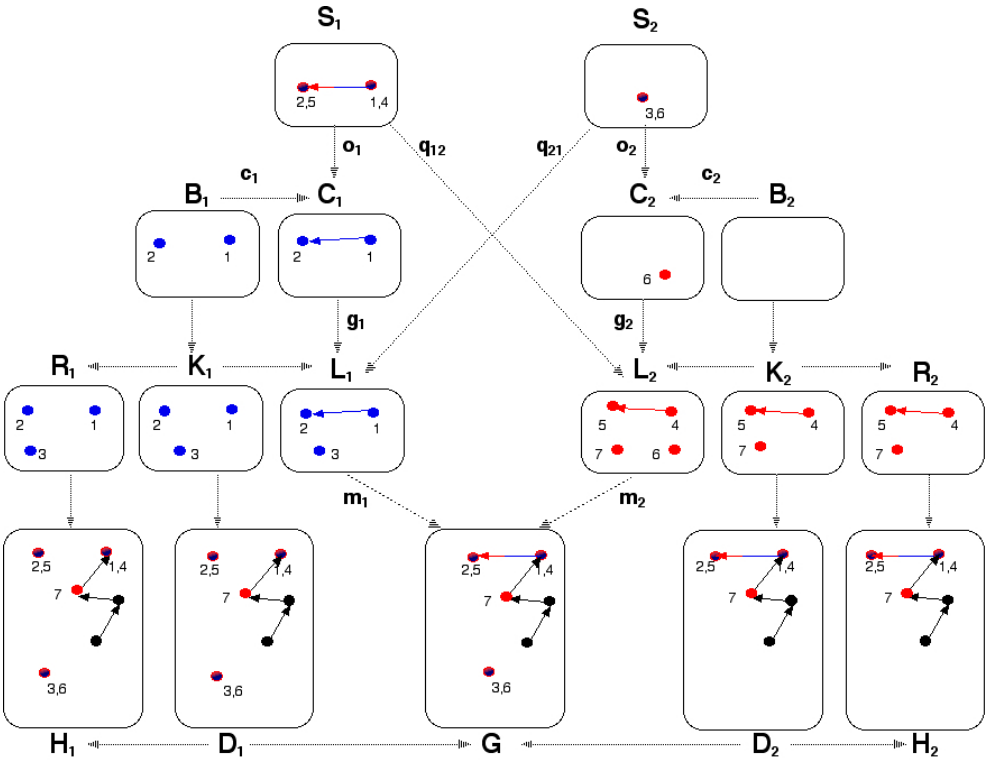


Fig. 2. symmetrical conflict

reasons:

(i) (S_1, o_1, q_{12}) satisfies and (S_2, q_{21}, o_2) doesn't satisfy the conflict condition (asymmetrical delete-use-conflict)

(ii) (S_1, o_1, q_{12}) doesn't satisfy and (S_2, q_{21}, o_2) satisfies the conflict condition (asymmetrical use-delete-conflict)

(iii) both (S_1, o_1, q_{12}) and (S_2, q_{21}, o_2) satisfy the conflict condition (symmetrical conflict)

In the case of asymmetrical conflicts rule p_1 (resp. p_2) deletes something, what is used by rule p_2 (resp. p_1), but *not* the other way round. Let us consider in more detail the case of symmetrical conflicts. In Fig. 2 you can see an example of two direct transformations, having a symmetrical conflict. Then (S_1, o_1, q_{12}) expresses the part which is deleted by p_1 and used by rule p_2 and (S_2, p_1, o_2) expresses the part which is deleted by p_2 and used by rule p_1. In order to summarize both parts into one graph expressing exactly the graph parts of L_1 and L_2 responsible for the conflict, we make the construction depicted in Fig. 3. In this construction (S', a_1, a_2) is the pullback of $(m_1 \circ g_1 \circ o_1 : S_1 \to G_1, m_2 \circ g_2 \circ o_2 : S_2 \to G_2)$ and (S, s'_1, s'_2) is the pushout of (S', a_1, a_2). This is, we determine the part S', which is deleted by both rules and glue S_1 and S_2 together over this part leading to S. Note, that in the example in Fig. 2 S' would be the empty graph. Now we have $g_1 \circ o_1 \circ a_1 = q_{21} \circ a_2$ and similar $g_1 \circ o_2 \circ a_2 = q_{12} \circ a_1$ because m_1 is mono and $m_1 \circ g_1 \circ o_1 \circ a_1 = m_2 \circ g_2 \circ o_2 \circ a_2 =$

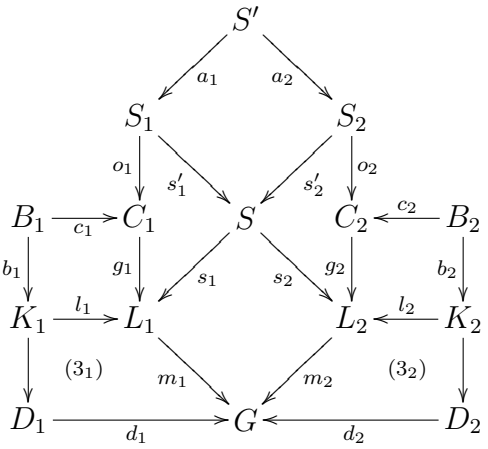


Fig. 3. construction of the conflict reason for symmetrical conflicts

$m_1 \circ q_{21} \circ a_2$. Together with the pushout property of S this implies, that there exists a unique $s_1 : S \to L_1$ (resp. $s_2 : S \to L_2$) s.t. $g_1 \circ o_1 = s_1 \circ s'_1$ and $q_{21} = s_1 \circ s'_2$ (resp. $g_2 \circ o_2 = s_2 \circ s'_2$ and $q_{12} = s_2 \circ s'_1$). Moreover using PO-property of S we can conclude $m_1 \circ s_1 = m_2 \circ s_2$. Please note, that in Fig. 3 we left out q_{21} and q_{12}. Thus in the end (S, s_1, s_2) summarizes which parts of L_1 and L_2 are responsible for the symmetrical conflict. **Remark:** $S = S_1 = S_2$ if and only if all elements deleted by p_1 are also deleted by p_2 and the other way round (pure delete-delete-conflict). $S' = \emptyset$ if and only if all elements deleted by p_1 are not deleted, but used by p_2 and the other way round (pure delete-use-conflict as in the example in Fig. 3).
We can resume these observations into the following definition.

Definition 2.3 [conflict reason span] Given a pair of direct transformations $H_1 \overset{p_1,m_1}{\Leftarrow} G \overset{p_2,m_2}{\Rightarrow} H_2$ in conflict, the *conflict reason span* of $H_1 \overset{p_1,m_1}{\Leftarrow} G \overset{p_2,m_2}{\Rightarrow} H_2$ is one of the following spans using the notation of Def.2.1:

- $(S_1, g_1 \circ o_1, q_{12})$ if (S_1, o_1, q_{12}) satisfies and (S_2, q_{21}, o_2) doesn't satisfy the conflict condition
- $(S_2, q_{21}, g_2 \circ o_2)$ if (S_1, o_1, q_{12}) doesn't satisfy and (S_2, q_{21}, o_2) satisfies the conflict condition
- (S, s_1, s_2) if (S_1, o_1, q_{12}) and (S_2, q_{21}, o_2) both satisfy the conflict condition and (S, s_1, s_2) is constructed as above

3 Definition of Essential Critical Pairs

By means of the new notion of conflict reason it is possible to define the new notion of essential critical pairs. The idea behind this notion is that for each conflict reason we have an essential critical pair, expressing the conflict caused by exactly this conflict reason in a minimal context.

Definition 3.1 [essential critical pair] A pair of direct transformations $P_1 \overset{p_1,m_1}{\Leftarrow}$

$K \overset{p_2,m_2}{\Rightarrow} P_2$ is an *essential critical pair* for the pair of rules (p_1, p_2) if the following holds: $P_1 \overset{p_1,m_1}{\Leftarrow} K \overset{p_2,m_2}{\Rightarrow} P_2$ are in conflict and (K, m_1, m_2) is a pushout of the conflict reason span $(S_1, g_1 \circ o_1, q_{12})$,$(S_2, q_{21}, g_2 \circ o_2)$ or (S, s_1, s_2) of $P_1 \overset{p_1,m_1}{\Leftarrow} K \overset{p_2,m_2}{\Rightarrow} P_2$ according to Definition 2.3.

Fact 3.2 *Each essential critical pair* $P_1 \overset{p_1,m_1}{\Leftarrow} K \overset{p_2,m_2}{\Rightarrow} P_2$ *of* (p_1, p_2) *is a critical pair of* (p_1, p_2).

Proof. Each essential critical pair is a pair of direct transformations in conflict. The overlappings (m_1, m_2) of an essential critical pair are jointly surjective, because they are constructed via a pushout. □

Remark: The main idea shown in the next section is that it is sufficient to consider essential critical pairs and not every critical pair is an essential critical pair. This is shown in the example in Fig. 4. The essential critical pair $P_1 \overset{p_1,m_1}{\Leftarrow} K \overset{p_2,m_2}{\Rightarrow} P_2$ of (p_1, p_2) only overlaps the edge $(1-2)$ with $(4-5)$, since this is exactly the reason for the delete-use-conflict. However the matches (m_1', m_2') of the critical pair $P_1' \overset{p_1,m_1'}{\Leftarrow} K' \overset{p_2,m_2'}{\Rightarrow} P_2'$ (with $m_1' = m_1 \circ m$ and $m_2' = m_2 \circ m$) overlap in addition node 7 with node 3, which are not responsible for the conflict at all. The pair of rules, used in the example in Fig. 1,2 and 4 induces, according to the critical pair detection in [13] AGG 14 critical pairs, but only 3 of them are essential critical pairs.

4 Properties of Essential Critical Pairs

In this section we will prove that it is enough to compute all essential critical pairs to detect all conflicts, occuring in a graph transformation system. Therefore we show, that the set of essential critical pairs fullfills the following three properties. At first, we demonstrate that each conflict, occuring in the system can be expressed by an essential critical pair (*completeness*). The second property says, that each essential critical pair is induced by a *unique* conflict reason. Finally we will prove a *local confluence lemma* based on essential critical pairs.

Theorem 4.1 (Completeness and Uniqueness of Essential Critical Pairs) *For each critical pair* $P_1' \overset{p_1,m_1'}{\Leftarrow} K' \overset{p_2,m_2'}{\Rightarrow} P_2'$ *of* (p_1, p_2) *there exists a unique essential critical pair* $P_1 \overset{p_1,m_1}{\Leftarrow} K \overset{p_2,m_2}{\Rightarrow} P_2$ *of* (p_1, p_2) *with the same conflict reason span and extension diagrams (1) and (2).*

$$\begin{array}{ccccc} P_1 & \Longleftarrow & K & \Longrightarrow & P_2 \\ \downarrow & (1) & \downarrow m & (2) & \downarrow \\ P_1' & \Longleftarrow & K' & \Longrightarrow & P_2' \end{array}$$

Remark: $m : K \to K'$ *is an epimorphism, but not necessarily a monomorphism.*

The proof of this theorem is given in appendix C in [10].

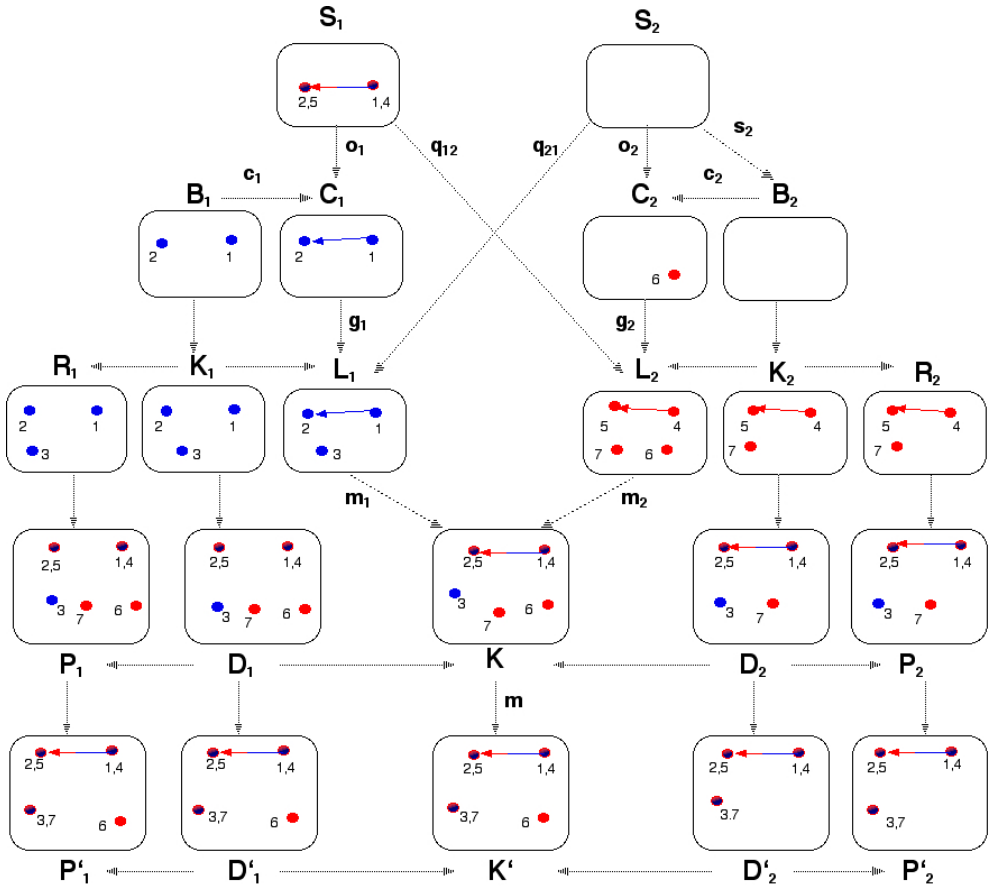


Fig. 4. essential crit. pair $P_1 \overset{p_1,m_1}{\Leftarrow} K \overset{p_2,m_2}{\Rightarrow} P_2$ into crit. pair $P_1' \overset{p_1,m_1'}{\Leftarrow} K' \overset{p_2,m_2'}{\Rightarrow} P_2'$

The set of essential critical pairs is unique in the following sense:

Theorem 4.2 (Uniqueness of Essential Critical Pairs) *Each essential critical pair possesses a unique conflict reason span.*

Proof. This follows directly from Theorem 4.1 and Fact 3.2. □

Note, that the set of critical pairs doesn't possess this uniqueness property. The example in Fig. 4 shows two different critical pairs (a normal critical pair $P_1' \overset{p_1,m_1'}{\Leftarrow} K' \overset{p_2,m_2'}{\Rightarrow} P_2'$ and an essential critical pair $P_1' \overset{p_1,m_1'}{\Leftarrow} K' \overset{p_2,m_2'}{\Rightarrow} P_2'$) possessing the same conflict reason span.

The following theorem states that it is enough to check each essential critical pair for strict confluence as defined in [12][6] to obtain local confluence of a graph transformation system.

Theorem 4.3 (Local Confluence Lemma based on Essential Critical Pairs) *If all essential critical pairs of a graph transformation system are strictly confluent, then this graph transformation system is locally confluent.*

The proof of this theorem is given in appendix D in [10]. It is similar to the proof of the local confluence lemma in [6], but avoids to assume that

$m : K \to K'$ is a monomorphism. Note, that the theory of essential critical pairs not only simplifies static conflict detection, but in addition confluence analysis of the conflicts in the system. This is because the number of conflicts to be analyzed for strictly confluence diminishes, since the essential critical pairs are a subset of the critical pairs.

5 Summary and Outlook

In this paper we have introduced the new notion of essential critical pairs and corresponding results which are the basis of a more efficient conflict detection and local confluence analysis than the standard techniques based on usual critical pairs. In a forthcoming paper we will give on this basis an efficient correct construction of all essential critical pairs for each pair of rules and a corresponding algorithm which will improve the current critical pair algorithm of AGG [13]. In addition we assume and will verify that an extension of this theory to graph transformation with non-injective matches is possible, provided that the conflict condition is slightly generalized. Moreover the following question in the context of conflict detection for graph transformation systems is subject of future work. What kind of new conflicts occur and which new critical pair notion is necessary to describe the conflicts in graph transformation systems with application conditions and constraints [5] and what about the more general case of typed, attributed graph transformation systems [7]?

References

[1] Baresi, L., R. Heckel, S. Thöne and V. D., *Modeling and analysis of architectural styles based on graph transformation*, in: I. Crnkovic, H. Schmidt, J. Stafford and K. Wallnau, editors, *Proc. of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction Portland* (2003), pp. 67–72.

[2] Bottoni, P., A. Schürr and G. Taentzer, *Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation*, in: *Proc. IEEE Symposium on Visual Languages*, 2000, long version available as technical report SI-2000-06, University of Rom.

[3] de Lara, J. and G. Taentzer, *Automated Model Transformation and its Validation using AToM^3 and AGG*, in: A. Blackwell, K. Marriott and A. Shimojima, editors, *Diagrammatic Representation and Inference* (2004).

[4] Ehrig, H. and K. Ehrig, *Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation*, in: *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, Electronic Notes in Theoretical Computer Science (2005).
URL http://tfs.cs.tu-berlin.de/publikationen/Papers05/EE05.pdf

[5] Ehrig, H., K. Ehrig, A. Habel and K.-H. Pennemann, *Constraints and application conditions: From graphs to high-level structures*, in: F. Parisi-Presicce, P. Bottoni and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, LNCS 3256 (2004), pp. 287–303.
URL http://www.cs.tu-berlin.de/~ehrig/publications/ICGT04paper3.pdf

[6] Ehrig, H., A. Habel, J. Padberg and U. Prange, *Adhesive high-level replacement categories and systems*, in: F. Parisi-Presicce, P. Bottoni and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, LNCS 3256 (2004), pp. 144–160.
URL http://www.cs.tu-berlin.de/~ehrig/publications/ICGT04paper1.pdf

[7] Ehrig, H., U. Prange and G. Taentzer, *Fundamental theory for typed attributed graph transformation*, in: F. Parisi-Presicce, P. Bottoni and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, LNCS 3256, Springer, 2004 pp. 161–177.
URL http://www.cs.tu-berlin.de/~ehrig/publications/ICGT04paper2.pdf

[8] Hausmann, J., R. Heckel and G. Taentzer, *Detection of Conflicting Functional Requirements in a Use Case-Driven Approach*, in: *Proc. of Int. Conference on Software Engineering 2002*, Orlando, USA, 2002, to appear.

[9] Lambers, L., H. Ehrig and F. Orejas, *Efficient detection of conflicts in graph-based model transformation*, in: *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, Electronic Notes in Theoretical Computer Science (2005).
URL http://tfs.cs.tu-berlin.de/publikationen/Papers05/LEO05.pdf

[10] Lambers, L., H. Ehrig and F. Orejas, *Efficient conflict detection in graph transformation systems by essential critical pairs*, Technical report, Technische Universität Berlin (2006).

[11] Mens, T., G. Taentzer and O. Runge, *Detecting Structural Refactoring Conflicts using Critical Pair Analysis*, in: R. Heckel and T. Mens, editors, *Proc. Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra'04), Satellite Event of ICGT'04)* (2004).
URL http://www.cs.tu-berlin.de/~gabi/gMTR04.pdf

[12] Plump, D., *Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence*, in: M. Sleep, M. Plasmeijer and M. C. van Eekelen, editors, *Term Graph Rewriting*, Wiley, 1993 pp. 201–214.

[13] Taentzer, G., *AGG: A Graph Transformation Environment for Modeling and Validation of Software*, in: J. Pfaltz, M. Nagl and B. Boehlen, editors, *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, LNCS 3062, Springer, 2004 pp. 446 – 456.

Exploiting user-definable synchronizations in graph transformation ⋆

Ivan Lanese [1]

Computer Science Department, University of Bologna, Bologna, Italy

Abstract

Parametric Synchronized Hyperedge Replacement (PSHR) is a *graph transformation* formalism where productions specifying the behavior of single components can be synchronized to give full transitions. The main feature of PSHR is that *the synchronization model is user-definable*. To enhance the applicability of the approach we propose a simplified and more suggestive semantics, preserving however the expressive power of the original one. We also show how some common synchronization models can be formalized and exploited inside PSHR. This allows to simplify the modelling step, and the produced model too. We apply this approach to the airport case study of FET-GC project AGILE.

Key words: Graph transformation, Synchronized Hyperedge Replacement, synchronization algebras, mobility.

1 Introduction

Architectural modelling is the step of the design of a system that fixes the structure of the system, that is its components and the connections among them, and its evolution over time. Since these aspects have a large impact on all the following phases of the development process, it is important that the decisions made are clearly stated in the model. This requires modelling frameworks with a formal syntax and semantics.

Many approaches to this problem have been presented in the literature, from UML [RJB99] to different Architecture Description Languages [Arc]. We choose as framework *Synchronized Hyperedge Replacement* (SHR) [DM87], which is a *graph transformation* framework. Thus the system is modelled as an (hyper)graph, where (hyper)edges are components connected through common nodes. This provides both sound mathematical foundations and a suggestive visual representation. In SHR the behavior of components is specified

⋆ Research supported by the Project FET-GC II Sensoria.

[1] Email: lanese@cs.unibo.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

by productions which can be synchronized to build transitions. In particular, productions perform actions on nearby nodes, and actions performed on the same node must be compatible according to some *synchronization model*. We also use *mobility* [HM01,FMT01], allowing actions to carry nodes as parameters, and synchronization to merge them thus reconfiguring the system. In particular, we consider Parametric SHR (PSHR) [LM04], where both action synchronization and mobility patterns are specified by a user-defined *Synchronization Algebra with Mobility* (SAM). This allows to choose each time the most suitable synchronization model for the application at hand.

PSHR is very expressive, as shown in [LM04], but its formal semantics is quite heavy and difficult to understand. This problem is common to other SHR variants, and is aggravated in PSHR by the need to manage different synchronization models. The problem is due to the fact that the standard semantics of SHR is based on inference rules that exploit a representation of graphs as terms in an algebra. In this presentation each part of the transition is obtained as a result of many inference steps, thus it is not easy to guess the global effect of a set of productions. We propose a more extensional semantics, where the synchronizations allowed on a node by a specific SAM are directly characterized, and an algorithm specifies how to build a full transition. Also, the semantics is based on a set-theoretic representation of graphs instead of on an algebraic one.

We also show how a synchronization model can be formalized as a SAM, and how PSHR can be used to model a system using this SAM. We apply this approach to the airport case study [And02], which has been proposed inside the FET-GC project AGILE [AGI] on architectures for mobility. We show that parametric synchronization allows a simpler model than the one presented in [BCG04], where a synchronized version of Double Pushout [EPS73] based on a fixed two-parties synchronization is used.

Structure of the paper. § 2 defines graphs and SHR transitions. § 3 presents SAMs, characterizes their effects, and analyzes the modelling of synchronization policies as SAMs. § 4 contains the algorithm to derive transitions from productions. § 5 details the application of the approach to the airport case study. Finally, § 6 presents conclusions and plans for future work.

2 Hypergraphs and SHR transitions

SHR [DM87] is an approach to (hyper)graph transformation that defines *global transitions* using *local productions*. Productions define how a single (hyper)edge can be rewritten and the *conditions* that this rewriting imposes. Conditions are specified as compatibility requirements among *actions* performed by productions on nearby nodes. The exact requirements depend on the chosen synchronization model. We use the extension of SHR with *mobility* [HM01,FMT01], that allows edges to send node references together with actions, and nodes whose references are matched during synchronization are

merged. In this work we use Parametric SHR (PSHR) [LM04], where the used synchronization model and mobility patterns can be freely chosen by specifying them via a Synchronization Algebra with Mobility (SAM). A detailed description of different SHR frameworks can be found in [Lan06].

The usual presentation of SHR is based on a representation of graphs as terms in a suitable term algebra and on inference rules to derive transitions from productions. This presentation allows to easily prove properties of the framework exploiting techniques from the process calculi field, but it is not so suggestive, since transitions are built as a result of many inference steps, and this makes difficult to understand the actual interactions. Also, the general mechanism is hidden because of heavy technicalities.

We propose here an original and more suggestive semantics, where transitions are built using an ad-hoc algorithm that highlights the main features of the synchronization and mobility mechanisms, and we present a direct description of the interactions allowed by a SAM. Also, our semantics is based on a set-theoretic presentation of graphs instead of on an algebraic one.

We always assume to have a countable set of nodes $\mathcal{N}$, a countable set of edges $\mathcal{E}$, and a countable ranked set of edge labels LE. Given $L \in LE$, $\mathrm{rank}(L)$ is its rank.

Definition 2.1 (Hypergraph)
A (hyper)graph *is a tuple* $\langle E, \mathrm{lab}, N, \mathrm{conn}, \Gamma\rangle$ *where* $E \subseteq \mathcal{E}$ *is the set of edges,* $\mathrm{lab} : E \to LE$ *is the labelling function for edges,* $N \subseteq \mathcal{N}$ *is the set of nodes,* $conn : E \to N^*$ *is a function mapping each edge* e *to a* n*-tuple of nodes where* n *is the rank of the label* $\mathrm{lab}(e)$*, and* $\Gamma \subseteq N$ *is the set of nodes in the interface. Nodes not in the interface are said hidden.*

Graphs are considered up to bijective renamings of edges and of hidden nodes.

In the above description *conn* specifies to which nodes each edge is attached. We present now the steps of an SHR computation.

Definition 2.2 (SHR transition) *Let Act be a set of actions, and given* $a \in Act$ *let* $\mathrm{ar}(a)$ *be its arity. An* SHR transition *is of the form:*

$$G \xrightarrow{\Lambda,\pi} G'$$

where G *and* G' *are graphs. Let* Γ_G *be the interface of* G*. Then* $\Lambda : \Gamma_G \to (Act \times \mathcal{N}^*)$ *is a total function and* $\pi : \Gamma_G \to \Gamma_G$ *is an idempotent substitution. Function* Λ *assigns to each node* x *the action* $a \in Act$ *and the vector* $\boldsymbol{y}$ *of node references sent to* x *by the transition. If* $\Lambda(x) = \langle a, \boldsymbol{y}\rangle$ *then we define* $\mathrm{n}_\Lambda(x) = \boldsymbol{y}$*. We require that* $\mathrm{ar}(a) = |\boldsymbol{y}|$*. We define the set of communicated names* $\mathrm{n}(\Lambda)$ *as* $\{z|\exists x.z \in \mathrm{n}_\Lambda(x)\}$*. Substitution* π *allows to merge nodes. Since* π *is idempotent, it maps every node into a standard representative of its equivalence class. We require that* $\forall x \in \mathrm{n}(\Lambda).x\pi = x$*, i.e., only references to representatives can be sent.*

SHR transitions are obtained by synchronizing productions using a specified synchronization model.

Definition 2.3 (SHR production) *An* SHR production *is an SHR transition* $G \xrightarrow{\Lambda,\mathrm{id}} G'$ *such that* G *is a graph containing exactly one edge* e*. Also, each node in* G *occurs exactly once in* $\mathrm{conn}(e)$*. Furthermore the node substitution in the label is* id *and the interface of* G' *is* $\Gamma_G \cup \mathrm{n}(\Lambda)$*.*

For each G *of the above form there is an idle production* $G \xrightarrow{\Lambda_\epsilon,\mathrm{id}} G$ *where* $\Lambda_\epsilon(x) = \langle \epsilon, \langle\rangle\rangle$ *for each* $x \in \Gamma_G$ *(*ϵ *is a special "idle" action with* $\mathrm{ar}(\epsilon) = 0$*). Idle productions are included in all sets of productions, which are also closed under bijective renamings of nodes.*

3 Synchronization Algebras with Mobility

We formalize a synchronization model as a Synchronization Algebra with Mobility (SAM). SAMs were first introduced in [LM04], extending Winskel's synchronization algebras (SAs) [Win84] to deal with mobility of nodes.

As a notation, we use $\uplus$ to denote disjoint set union. In $A \uplus B$ we denote with $[1, x]$ (resp. $[2, x]$) the element that corresponds to $x \in A$ (resp. $x \in B$).

Definition 3.1 (Synchronization algebra with mobility) *A* Synchronization Algebra with Mobility $< Act, \mathrm{ar}, \bullet, \epsilon, \mathrm{mob}, Fin >$ *consists of a binary partial operator* $\bullet$ *on a set of actions* Act*, a set of mobility patterns* mob *and a subset* Fin *of* Act*. Function* $\mathrm{ar} : Act \to \mathbb{N}$ *maps each action* $a \in Act$ *to its arity* $\mathrm{ar}(a)$*, and* $\epsilon \in Act$ *is an action of arity* 0*. Here* mob *is a set indexed by pairs of actions* (a, b) *such that* $a \bullet b$ *is defined, and* $\mathrm{mob}_{a,b}$ *is a partial function from* $\{1, \ldots, \mathrm{ar}(a)\} \uplus \{1, \ldots, \mathrm{ar}(b)\}$ *to* $\mathbb{N}$*.*

We impose the following conditions:

(i) *the* $\bullet$ *operator is associative and commutative;*

(ii) $\forall a, a' \in Act. a \bullet a' = \epsilon \Rightarrow a = a' = \epsilon$*;*

(iii) $\forall a \in Act. a \bullet \epsilon$ *is defined* $\Rightarrow$
$(a \bullet \epsilon = a \wedge \forall x \in \{1, \ldots, \mathrm{ar}(a)\}. \mathrm{mob}_{a,\epsilon}([1, x]) = x)$*;*

(iv) $\epsilon \in Fin$*;*

(v) $\forall a, b, c \in Act$
$\forall x \in \{1, \ldots, \mathrm{ar}(a)\}. \mathrm{mob}_{a\bullet b,c}([1, \mathrm{mob}_{a,b}([1, x])]) = \mathrm{mob}_{a,b\bullet c}([1, x])$,
$\forall x \in \{1, \ldots, \mathrm{ar}(b)\}. \mathrm{mob}_{a\bullet b,c}([1, \mathrm{mob}_{a,b}([2, x])]) = \mathrm{mob}_{a,b\bullet c}([2, \mathrm{mob}_{b,c}([1, x])])$,
$\forall x \in \{1, \ldots, \mathrm{ar}(c)\}. \mathrm{mob}_{a\bullet b,c}([2, x]) = \mathrm{mob}_{a,b\bullet c}([2, \mathrm{mob}_{b,c}([2, x])])$*;*

(vi) $\forall a, b \in Act, x \in \{1, \ldots, \mathrm{ar}(a)\}. \mathrm{mob}_{a,b}([1, x]) = \mathrm{mob}_{b,a}([2, x])$*;*

(vii) $\forall a, b \in Act. \mathrm{mob}_{a,b}$ *is surjective on* $\{1, \ldots, \mathrm{ar}(a \bullet b)\}$*.*

As in SAs, we have a set of actions Act and an operator $\bullet$ of action composition. Here $a \bullet b = c$ means that actions a and b can synchronize giving action c as a result. If $a \bullet b$ is undefined then a and b are not compatible. For instance in CCS an action a can synchronize with a coaction $\overline{a}$ producing τ as a result.

Action ϵ stands for "not taking part to the synchronization", and it allows to specify in a uniform way action synchronization and asynchronous execution of actions. In fact, $a \bullet \epsilon = a$ means that a is executed asynchronously. With respect to SAs, now actions a in Act have a specified arity $\mathrm{ar}(a)$, which corresponds to the number of node references carried by a. A mobility pattern $\mathrm{mob}_{a,b}$ specifies how to build the references attached to $a \bullet b$ starting from the references attached to a and b. The correspondence is just positional as in usual procedure calls, but many parameters can be assigned to just one position. In that case the parameters are merged and the result is assigned to the chosen position. Using $\mathbb{N}$ as codomain instead of $\{1, \ldots, \mathrm{ar}(a \bullet b)\}$ allows to specify merges among parameters even if the chosen representative of the equivalence class defined in this way does not occur in the final label.

Simple message passing is specified by a set of mobility patterns MP that merges corresponding references and assigns the result to the corresponding position. Formally, $MP_{a_1,a_2}([n,x]) = x$ for each $n \in \{1,2\}$, $x \in \{1, \ldots, \mathrm{ar}(a_n)\}$.

A mobility pattern mob_{a_1,a_2} included in a SAM S can be applied to two actions $\langle a_1, \boldsymbol{y}_1 \rangle$ and $\langle a_2, \boldsymbol{y}_2 \rangle$ to compute both the substitution σ performing the merge of parameters and the vector of parameters of the result, given respectively by the two functions:

$$\begin{aligned}
&\sigma = \mathrm{sub}(S, \langle a_1, \boldsymbol{y}_1 \rangle, \langle a_2, \boldsymbol{y}_2 \rangle) = \\
&\qquad \mathrm{mgu}(\{\boldsymbol{y}_i[j] = \boldsymbol{y}_h[k] \mid \mathrm{mob}_{a_1,a_2}([i,j]) = \mathrm{mob}_{a_1,a_2}([h,k])\}) \\
&\mathrm{par}(S, \langle a_1, \boldsymbol{y}_1 \rangle, \langle a_2, \boldsymbol{y}_2 \rangle)[i] = (y_h[k])\sigma \\
&\qquad \text{iff } \exists h,k.\, \mathrm{mob}_{a_1,a_2}([h,k]) = i \wedge i \leq \mathrm{ar}(a_1 \bullet a_2)
\end{aligned}$$

For instance, let S be a SAM with actions a, b and c of arity 1, 3 and 2 respectively, such that $a \bullet b = c$. Then $\mathrm{sub}(S, \langle a, \langle x_1 \rangle \rangle, \langle b, \langle y_1, y_2, y_3 \rangle \rangle) = \{x_1/y_1\}$ (also $\{y_1/x_1\}$ is a valid choice) and $\mathrm{par}(S, \langle a, \langle x_1 \rangle \rangle, \langle b, \langle y_1, y_2, y_3 \rangle \rangle) = \langle x_1, y_2 \rangle$. If we consider an action a' of arity 3 with parameters $\langle x_1, x_2, x_3 \rangle$ instead of a, then $\sigma = \{x_1/y_1, x_2/y_2, x_3/y_3\}$, but x_3 is not a parameter of the resulting action.

Fin is the set of complete synchronizations, that is synchronizations that are allowed on hidden nodes. For instance, in CCS-style synchronization just τ is allowed on those nodes.

Conditions (i) and (ii) are from SAs. The former specifies that the result of an n-ary synchronization does not depend on the order in which actions are synchronized. The latter specifies that non ϵ actions can not disappear giving ϵ. Condition (iii) specifies that synchronization with ϵ, if allowed, just propagates the other action. Condition (iv) assures that all the edges can stay idle on any node. Conditions (v) and (vi) state that mobility patterns are associative and commutative, extending condition (i) to the mobility part. Finally, condition (vii) guarantees that each reference attached to the composed action can be computed, that is it corresponds to a non empty set of references from component actions.

We now characterize the effects of the synchronization specified by a SAM S on a n-uple of actions $\langle\langle a_1, \boldsymbol{y}_1\rangle, \ldots, \langle a_n, \boldsymbol{y}_n\rangle\rangle$. The effects of the synchronization are an action c_n with a tuple of parameters $\boldsymbol{w}_n$ and a substitution ρ_n. We use $\mathrm{eqn}(\{t_1/x_1, \ldots, t_m/x_m\})$ to denote $\{t_1 = x_1, \ldots, t_m = x_m\}$.

Definition 3.2 (Effects of a synchronization) *The* effects of a synchronization *are computed by induction on the number n of actions.*

$n = 1$) $c_1 = a_1$, $\boldsymbol{w}_1 = \boldsymbol{y}_1$, $\rho_1 = \mathrm{id}$.

Inductive case) Let c_n, $\boldsymbol{w}_n$ and ρ_n be the effects of the synchronization among the first n actions. Then:

$c_{n+1} = c_n \bullet a_{n+1}$,
$\rho_{n+1} = \mathrm{mgu}(\mathrm{eqn}(\rho_n) \cup \mathrm{eqn}(\mathrm{sub}(S, \langle c_n, \boldsymbol{w}_n\rangle, \langle a_{n+1}, \boldsymbol{y}_{n+1}\rangle)))$,
$\boldsymbol{w}_{n+1} = \mathrm{par}(S, \langle c_n, \boldsymbol{w}_n\rangle, \langle a_{n+1}, \boldsymbol{y}_{n+1}\rangle)\rho_{n+1}$.

Conditions (i), (v), (vi) in Definition 3.1 ensure that the result is independent w.r.t. the order of $a_1, \ldots, a_n$.

We present now some simple SAMs which can be used as building blocks for more complex ones, highlighting the technical aspects of the formalization of a synchronization model as SAM. We just write the cases where $\bullet$ is defined. We also skip cases that can be derived by commutativity. Furthermore, in the examples, unless explicitly stated, we use $\mathrm{mob}_{a,b} = MP_{a,b}$. The following SAMs use the minimal number of actions necessary to model a synchronization of the chosen type, but sets of actions sharing just ϵ can be merged in a unique SAM allowing different policies. In this case the action performed chooses the protocol to be used, since it can interact only with other actions from the same group. An example of this kind is presented in § 5.

Example 3.3 (Mutual exclusion SAM)
The mutual exclusion SAM is defined by:

- $Fin = Act = \{a, \epsilon\}$;
- $\lambda \bullet \epsilon = \lambda$ *for each* $\lambda \in Act$.

Mutual exclusion ensures that in each transition at most one non ϵ action can be performed on each node. Synchronization of a with ϵ is necessary to allow transitions when more than one component is attached to the node. The SAM obtained by removing this synchronization allows to detect if an edge is the only one attached to a node, and it is attached just one time to it.

Example 3.4 (Milner SAM) *The Milner SAM is defined by:*

- $Act = \{a, \overline{a}, \tau, \epsilon\}$ *with* $\mathrm{ar}(a) = \mathrm{ar}(\overline{a})$ *and* $\mathrm{ar}(\tau) = 0$;
- $a \bullet \overline{a} = \tau$, $\lambda \bullet \epsilon = \lambda$ *for each* $\lambda \in Act$;
- $Fin = \{\tau, \epsilon\}$.

The Milner SAM, so called since it is inspired by π-calculus synchronization, models message passing, where actions a and $\overline{a}$ are input and output respectively and τ stands for a complete message exchange. During synchro-

nization corresponding parameters are merged. Technically this is a refinement of the mutual exclusion SAM, in fact mutual exclusion is imposed between different synchronizations. Having just τ and ϵ in Fin ensures that on a hidden node x either nothing happens or a complete message exchange is performed. Note that a SAM that uses many actions, all interacting using the Milner protocol can be built, and many variations are possible. For instance, the desired possibilities of input-output interactions can be specified, e.g., allowing an input to interact with all the outputs in a given set.

We present now an extension of Milner SAM where communication has to be authorized by a particular action ok, thus allowing a simple form of traffic control. Thus a τ is here obtained as a result of a synchronization among a, $\overline{a}$ and ok. We consider the simpler case of actions and coactions having arity 1.

Example 3.5 (Controlled Milner SAM) *The controlled Milner SAM is defined by:*

- $Act = \{a, \overline{a}, ok, (a, \overline{a}), (a, ok), (\overline{a}, ok), \tau, \epsilon\}$ *with* $\mathrm{ar}(\lambda) = 1$ *for each* $\lambda \in Act \setminus \{ok, (a, \overline{a}), \tau, \epsilon\}$, $\mathrm{ar}((a, \overline{a})) = 2$ *and* $\mathrm{ar}(\lambda') = 0$ *for each* $\lambda \in \{ok, \tau, \epsilon\}$;
- $a \bullet \overline{a} = (a, \overline{a})$ *with* $\mathrm{mob}_{a,\overline{a}}([1,1]) = 1$, $\mathrm{mob}_{a,\overline{a}}([2,1]) = 2$, $(a, \overline{a}) \bullet ok = \tau$ *with* $\mathrm{mob}_{(a,\overline{a}),ok}([1,x]) = 1$ *for each* $x \in \{1, 2\}$, $a \bullet ok = (a, ok)$, $\overline{a} \bullet ok = (\overline{a}, ok)$, $(a, ok) \bullet \overline{a} = \tau$, $(\overline{a}, ok) \bullet a = \tau$, $\lambda \bullet \epsilon = \lambda$ *for each* $\lambda \in Act$;
- $Fin = \{\tau, \epsilon\}$.

We have used here a technical trick: actions $(a, \overline{a})$, (a, ok) and $(\overline{a}, ok)$ are generally not used in productions, but they are used as intermediate results in the computation of the full synchronization. Note that here mobility patterns are not always specified by MP.

Example 3.6 (Broadcast SAM) *The broadcast SAM is defined by:*

- $Act = \{a, \overline{a}, \epsilon\}$ *with* $\mathrm{ar}(a) = \mathrm{ar}(\overline{a})$;
- $a \bullet \overline{a} = \overline{a}$, $a \bullet a = a$, $\epsilon \bullet \epsilon = \epsilon$;
- $Fin = \{\overline{a}, \epsilon\}$.

The broadcast SAM models secure broadcast, where one component performs an output and all the others perform input. Notice that here reaction with ϵ is not allowed, and this requires all the components to participate in a non idle way to the synchronization. The requirement can be weakened by allowing some components to stay idle, thus obtaining multicast.

Example 3.7 (Multicast SAM) *The multicast SAM is defined by:*

- $Act = \{a, \overline{a}, \epsilon\}$ *with* $\mathrm{ar}(a) = \mathrm{ar}(\overline{a})$;
- $a \bullet \overline{a} = \overline{a}$, $a \bullet a = a$, $\lambda \bullet \epsilon = \lambda$ *for each* $\lambda \in Act$;
- $Fin = \{\overline{a}, \epsilon\}$.

We show here the effects of the synchronization of a tuple of actions

$\langle\langle a_1, \boldsymbol{y}_1\rangle, \ldots, \langle a_n, \boldsymbol{y}_n\rangle\rangle$ according to multicast SAM. The synchronization is allowed provided that at most one action is $\overline{a}$. On a hidden node exactly one action must be $\overline{a}$. Also, ρ is an mgu of $\{\boldsymbol{y}_{i_1} = \boldsymbol{y}_{i_2} = \cdots = \boldsymbol{y}_{i_m}\}$ where $\{i_1, \ldots, i_m\}$ are the indexes of the non ϵ actions. Finally, $\boldsymbol{w} = \boldsymbol{y}_{i_1}\rho$.

4 Deriving transitions from productions

In this section we present an algorithm to derive all the transitions starting from a graph G (without isolated nodes) specified by a set of productions $\mathcal{P}$ using a SAM S. All the actions used in $\mathcal{P}$ are required to belong to Act.

The steps of the algorithm are described below.

(i) For each edge e a production $P_e = L_e \xrightarrow{\Lambda_e,\mathrm{id}} R_e$ is chosen, in such a way that there exists an idempotent substitution $\sigma_e : \Gamma_{L_e} \to \Gamma_{L_e}$ such that $L_e\sigma_e$ is the subgraph of G composed by the edge e and the attached nodes. Essentially L_e is equal to the desired subgraph, but if e is attached many times to the same node x then all the occurrences of x but one are renamed in L_e using fresh names: σ_e performs the inverse substitution. Furthermore nodes created by the productions (i.e., in Λ_e but not in L_e) must be fresh.

(ii) For each node x, all the actions performed by productions P_e on nodes y such that $y\sigma_e = x$ are instantiated by applying σ_e to their tuples of parameters and then composed, producing an action c_x with parameters $\boldsymbol{w}_x$ and a substitution ρ_x.

(iii) If there is at least a node x for which the above action composition is not defined, or $x \notin \Gamma_G$ but $c_x \notin Fin$, then no transition can be derived for this choice of productions.

(iv) A global substitution ρ is defined as the composition of all the substitutions ρ_x, that is $\rho = \mathrm{mgu}\{\bigcup_{x\in N} \mathrm{eqn}(\rho_x)\}$. Among the possible mgus we choose one where nodes in Γ_G are taken as representatives of their equivalence classes whenever possible.

(v) Λ maps each node x in Γ_G to the pair $\langle c_x, \boldsymbol{w}_x\rho\rangle$.

(vi) π is the restriction of ρ to the nodes in Γ_G.

(vii) The final graph is obtained as follows:
- a graph is obtained by merging the instances $R_e\sigma_e$ of the RHSs of all the productions P_e (choosing different representatives for hidden nodes and edges in different RHSs);
- the substitution ρ is applied to the graph;
- only nodes in $\Gamma_G \cup \mathrm{n}(\Lambda)$ that occur in the resulting graph are kept in the interface;
- isolated nodes are deleted.

We will not state here a formal theorem relating our semantics with the one presented in [LM04], since the present semantics formalizes a more refined

approach to PSHR w.r.t. the older one available in [LM04]. In fact, the two approaches allow slightly different classes of SAMs. We will however discuss informally the differences between the two semantics.

In [LM04] graphs are represented as syntactic judgements $\Gamma \vdash T$ where T is a term and Γ is the set of nodes in the interface (corresponding to Γ_G here). The term T is built using constants for edges and the empty graph, and operators for composing graphs (merging common nodes) and hiding nodes. Term T is considered up to a structural congruence that abstracts from the order of edges and of hidings and allows α-conversion of nodes. Edges are not explicitly named: just the labels are considered. Judgements up to structural congruence can be interpreted into graphs, obtaining a bijective correspondence.

From a dynamic point of view the main difference between the semantics presented here and the standard one is that here isolated nodes are forbidden in the starting graph and removed from the result, while they are allowed in the standard one. This is not an important restriction since isolated nodes can not influence the other parts of the graph. They are actually needed in the standard semantics for the internal steps of the derivation of some transitions. In our case it is enough to allow them in productions. This difference allows to remove the component $Init$, used in [LM04], from SAM definition. Also, the standard semantics allows a non identity substitution π also in productions, but this is superfluous since the same effect can be obtained by synchronizing two actions on a hidden node. However this feature can be added also to our semantics. If we restrict our attention to SAMs that can be specified in both the frameworks (and we find a suitable set $Init$ for [LM04]-style SAMs), and to productions having just id as node substitution, then the two semantics are equivalent up to isolated nodes (and actions performed on them).

5 The airport case study

We show here how the approach described above can be applied to the airport case study [And02] of FET-GC project AGILE [AGI]. Since we are not aiming at tackling the whole case study, but just at showing how PSHR can be applied, we will do some simplifications. For a more complete approach to this modelling problem see [BCG04].

The airport case study concentrates on modelling planes landing and tacking off at airports, with passengers boarding the planes. We model entities (which are classes in UML class diagrams [RJB99]) as edges with attributes modelled as nodes. In particular, we have edges for airports, planes and passengers. In this example, the first connection of each edge represents the attribute AtLoc, proposed in an extension of UML diagrams with mobility [BKKW02]. The value of attribute AtLoc represents the location containing a mobile object. Also, objects that are locations such as airports and planes have the dual attribute Containing. Furthermore, planes have an

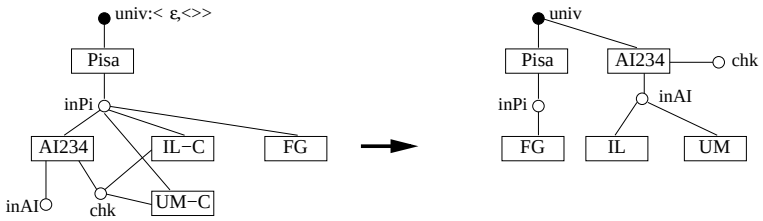


Fig. 1. A sample transition.

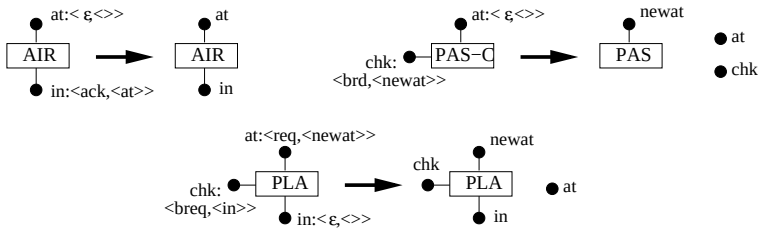


Fig. 2. Productions for the example.

attribute CheckedIn, whose value is the set of passengers that have already checked in for next flight. Passengers that have already checked in have the dual attribute. A simple graph modelling a system of this kind is the left graph in Figure 1, featuring one airport ($Pisa$) located in the universe ($univ$), a plane ($AI234$) in the airport and three passengers, two which have already checked in ($IL\text{-}C$, $UM\text{-}C$), and one which has not (FG). We represent edges as rectangles containing the label and connected to bullets representing nodes. Bullets are solid for nodes in the interface and empty otherwise.

We want to specify a transition that models the boarding of all the passengers who have checked in and the take off of the plane. This transition requires multiple checks and reconfigurations: essentially all the passengers who have checked in must move (changing their location), and the airport must allow the plane to take off. The plane must change its location too.

This is modelled by the productions in Figure 2 (which are schemas drawn for generic labels AIR, PLA, $PAS\text{-}C$ and PAS for airports, planes, checked in passengers and not checked in passengers respectively), where Λ is represented by decorating each node with the corresponding action.

We have now to specify the SAM S that we want to use. Actions ack and req have to synchronize using Milner synchronization, since they model a message exchange between the airport and the plane allowing the take off, while actions $breq$ and brd have to synchronize using broadcast synchronization with $breq$ as output action, since all the checked in passengers have to board. Thus we can build the wanted SAM using the Milner SAM and the broadcast SAM as building blocks. The resulting SAM is defined by:

- $Act = \{req, ack, breq, brd, \tau, \epsilon\}$ with $\mathrm{ar}(\lambda) = 1$ for each $\lambda \in Act \setminus \{\tau, \epsilon\}$ and $\mathrm{ar}(\tau) = 0$;
- $req \bullet ack = \tau$, $breq \bullet brd = breq$, $brd \bullet brd = brd$,

$\lambda \bullet \epsilon = \lambda$ for each $\lambda \in \{req, ack, \tau, \epsilon\}$;

- $Fin = \{\tau, breq, \epsilon\}$.

We can thus derive the transition in Figure 1. Let us see how the different steps of the algorithm are performed.

(i) For each edge but FG the corresponding production in Figure 1 is used, for edge FG an idle production is used. For nodes in the LHSs the names in the graph can be used, since no edge is attached two times to the same node. For new nodes (all called *newat* in the figure) different names must be chosen. To this end we add the label of the corresponding edge to the nodes created by passenger edges.

(ii) Let us consider node $inPi$ as example. The actions performed here are $\langle ack, \langle univ \rangle\rangle, \langle req, \langle newat \rangle\rangle, \langle \epsilon, \langle\rangle\rangle, \langle \epsilon, \langle\rangle\rangle, \langle \epsilon, \langle\rangle\rangle$. These can be composed producing $\rho_{inPi} = \{univ/newat\}$ and action $\langle \tau, \langle\rangle\rangle$.

(iii) The transition is allowed since the compositions are all defined and for nodes different from $univ$ the resulting action is in Fin.

(iv) The substitution ρ is $\{univ/newat, inPi/newat_{IL-C}, inPi/newat_{UM-C}\}$.

(v) Λ maps just $univ$ to $\langle \epsilon, \langle\rangle\rangle$.

(vi) π is the identity substitution.

(vii) The final graph is obtained from the union of the RHSs, applying substitution ρ and leaving only $univ$ in the interface.

It is interesting to notice that, when a suitable SAM is chosen for synchronization, the implementation of the communication protocol becomes trivial. In [BCG04] instead just a simple binary synchronization is used, thus a complex procedure is required to implement broadcast. In particular, this adds to the model a subgraph used for synchronization purposes which does not correspond to any entity in the real system. Our choice allows models at a more abstract level, as suited for modelling complex systems.

6 Conclusion and future work

We have provided a more direct characterization of the behavior of a SAM and of the transitions allowed by PSHR w.r.t. [LM04]. We think that this is useful to make PSHR more usable. The result applies also to most of the SHR frameworks in the literature, which are instances of PSHR with a suitable SAM. Our approach can be also straightforwardly extended to deal with nondeterministic synchronizations and the use of many SAMs inside the same graph as presented in [LT05].

As future work we want to formalize different forms of SAM composition using categorical tools and analyze the observational semantics of SHR systems.

References

[AGI] AGILE: Architectures for mobility. http://www.pst.informatik.uni-muenchen.de/projekte/agile/.

[And02] L. F. Andrade et al. AGILE: Software architecture for mobility. In *Proc. of WADT'02*, volume 2755 of *LNCS*, pages 1–33. Springer, 2002.

[Arc] Architecture description languages. http://www.sei.cmu.edu/architecture/adl.html.

[BCG04] P. Baldan, A. Corradini, and F. Gadducci. Specifying and verifying UML activity diagrams via graph transformation. In *Proc. of Global Computing 2004*, volume 3267 of *LNCS*, pages 18–33. Springer, 2004.

[BKKW02] H. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing. Extending activity diagrams to model mobile systems. In *Proc. of NetObjectDays'02*, volume 2591 of *LNCS*, pages 278–293. Springer, 2002.

[DM87] P. Degano and U. Montanari. A model for distributed systems based on graph rewriting. *Journal of the ACM*, 34(2):411–449, 1987.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *Proc. of SWAT '73*, pages 167–180, IEEE, 1973.

[FMT01] G. Ferrari, U. Montanari, and E. Tuosto. A LTS semantics of ambients via graph synchronization with mobility. In *Proc. of ICTCS '01*, volume 2202 of *LNCS*, pages 1–16. Springer, 2001.

[HM01] D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In *Proc. of CONCUR '01*, volume 2154 of *LNCS*. Springer, 2001.

[Lan06] I. Lanese. *Synchronization strategies for global computing models*. PhD thesis, Computer Science Department, University of Pisa, Pisa, Italy, 2006. Forthcoming.

[LM04] I. Lanese and U. Montanari. Synchronization algebras with mobility for graph transformations. In *Proc. of FGUC'04 – Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 43–60. ES, 2004.

[LT05] I. Lanese and E. Tuosto. Synchronized hyperedge replacement for heterogeneous systems. In *Proc. of COORDINATION 2005*, volume 3454 of *LNCS*, pages 220–235. Springer, 2005.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Book. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[Win84] G. Winskel. Synchronization trees. *TCS*, 34:33–82, 1984.

Towards a Notion of Transaction in Graph Rewriting [1]

P. Baldan [a], A. Corradini [b], F.L. Dotti [c], L. Foss [b,d,2]
F. Gadducci [b], L. Ribeiro [d]

[a] *Università Ca' Foscari di Venezia, Italy*

[b] *Università di Pisa, Italy*

[c] *Pontifícia Universidade Católica do Rio Grande do Sul, Brasil*

[d] *Universidade Federal do Rio Grande do Sul, Brasil*

Abstract

We define *transactional graph transformation systems* (T-GTSs), a mild extension of the ordinary framework for the double-pushout approach to graph transformation, which allows to model transactional activities. Generalising the work on *zero-safe nets*, the new graphical formalism is based on a typing discipline which induces a distinction between stable and unstable items. A transaction is then a suitably defined minimal computation which starts and ends in stable states. After providing the basics of T-GTSs, we illustrate the expected results, needed to bring the theory to full maturity, and some possible future developments.

Key words: Graph transformations, zero-safe nets, transactions.

1 Introduction

Graphs and graph transformations represent the core of most visual languages [2]. In fact, graphs can be naturally used to provide a structured representation of the states of a system, which highlights its subcomponents and their logical or physical interconnections. Then, the events occurring in the system, which are responsible for the evolution from one state into another, can be modelled as the application of graph transformation rules. Such a representation is not only precise enough to allow the formal analysis of the system under scrutiny, but it is also amenable of an intuitive, visual representation, which can be easily understood also by a non-expert audience.

[1] Research partially supported by the CNPq-CNR Project IQ-Mobile II, the EC RTN 2-2001-00346 SegraVis, the EU IST-2004-16004 SENSORIA and the MIUR Project ART.
[2] Supported by CAPES and CNPq.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

A *graph transformation system* (GTS) consists of a set of rewriting rules, also called graph productions [14]. In their basic formulation, GTSs do not provide mechanisms for synchronising or structuring computations, even if, since the left-hand side of productions can be arbitrarily large, a kind of synchronisation among items in the state (graph items) can be expressed.

Along the years several enrichments of the basic framework have been proposed, extending GTSs with mechanisms for expressing synchronisation between productions as well as for tackling modularity and refinement issues, which are features deemed necessary for a high-level specification formalism.

Instead, to our knowledge, scarce attention has been devoted to the idea of extending GTSs in order to allow the specification of transactional activities. Abstractly, a transaction is an activity, involving the execution of a group of events, which can either bring the system to a successful state or fail. In the last case the partial execution of the transaction is discarded and has no effect on the system. In concrete implementations this is achieved with a roll-back mechanism which restores the starting state when a failure is detected.

In this paper we face, from a foundational perspective, the problem of equipping graph transformation with mechanisms for modelling transactions. More precisely, we propose a mild extension to the double-pushout (DPO) approach to graph transformation, introducing *transactional graph transformation systems* (T-GTSs), a framework which provides a simple way of expressing transactional activities. Our formalism is deeply influenced and generalists the *zero-safe nets* proposal, introduced in [3] to solve an analogous modelling problem in the setting of Petri nets.

The basic tool is a typing mechanism for graphs which induces a distinction between *stable* and *unstable* graph items. Given a typed graph, representing a system state, we can identify a subgraph which represent its "stable" part, i.e., the fragment of the state which is visible from an external observer. The "valid" computations of a T-GTS may start from a completely stable graph, evolve through graphs with unstable items and eventually end up in a new stable state; and the valid computations which are minimal, in a certain sense to be made precise in the paper, represent *transactions*.

The paper introduces the T-GTSs formalism, provides the basic concepts and illustrates a simple case study. In a concluding section we outline how the internal structure of transactions can be abstracted away by considering the so-called *abstract* GTS associated to the T-GTS, where unstable items disappear and each distinct transaction becomes a single atomic production, which rewrites the starting stable state to the final stable state. Thus "unfinished" transactions have no counterpart at the abstract level. Finally, we outline future venues of research, pointing out the technical issues which need to be further elaborated upon, such as the precise functorial correspondence between a T-GTS and its abstract counterpart.

2 Typed Graph Transformation Systems

In this section we introduce the basics of the double-pushout (DPO) algebraic approach to graph rewriting [9]. We remark that, although our approach will be developed for DPO rewriting over directed (multi-)graphs, it could have been easily adapted to other approaches to graph rewriting, e.g., to the single-pushout approach and to different notions of graph (e.g., to hypergraphs, which are used indeed in the example in Section 4).

An essential ingredient of our theory is a typing discipline for graphs which will allow us to distinguish between stable and unstable items in a given graph. Typing for graphs (e.g., [5]) can be seen as a labelling technique, which allows to label each graph over a structure that is itself a graph (called the *type graph*). The labelling function is required to be a graph morphism.

Formally, a *graph* is a tuple $\langle V, E, s, t\rangle$, where V and E are sets of nodes and edges, and $s, t : E \to V$ are the source and target functions. Given a graph T, a *typed graph* G over T is a graph $|G|$, together with a total graph morphism $t_G : |G| \to T$. A *morphism* between T-typed graphs $f : G_1 \to G_2$ is a graph morphism $f : |G_1| \to |G_2|$ consistent with the typing, i.e., such that $t_{G_1} = t_{G_2} \circ f$. The category of T-typed graphs and typed graph morphisms is denoted by T-**Graph**.

Rewriting rules, called *(T-typed) productions*, are of the kind $q = L_q \stackrel{l_q}{\leftarrow} K_q \stackrel{r_q}{\rightarrow} R_q$, where L_q, K_q and R_q are T-typed graphs (called the left-hand side, the interface and the right-hand side of the production, respectively), and l_q, r_q are injective morphisms. A rule intuitively specifies that an occurrence of the left-hand side L_q in a larger graph can be rewritten into the right-hand side graph R_q, preserving the interface K_q. Formally, given a typed graph G, a production q, and a *match* $g : L_q \to G$, a *direct derivation δ from G to H using q, g* exists, written $\delta : G \stackrel{q,g}{\Longrightarrow} H$, if the diagram

$$\begin{array}{ccccc} q: L_q & \xleftarrow{l_q} & K_q & \xrightarrow{r_q} & R_q \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle k} & & \downarrow{\scriptstyle h} \\ G & \xleftarrow[b]{} & D & \xrightarrow[d]{} & H \end{array}$$

can be constructed, where both squares are pushouts in T-**Graph**.

A graph transformation system is then defined as a collection of rules, over a fixed graph of types.

Definition 2.1 [graph transformation system] A T-typed *graph transformation system* (GTS) is a tuple $\mathcal{G} = \langle T, P, \pi\rangle$, where T is a graph, P is a set of production names and π is a function mapping production names in P to T-typed DPO productions.

A *derivation* in a GTS $\mathcal{G}$ is a sequence of direct derivations using productions of $\mathcal{G}$

$$G_0 \stackrel{q_0,g_0}{\Longrightarrow} G_1 \stackrel{q_1,g_1}{\Longrightarrow} \ldots\ldots \stackrel{q_n,g_n}{\Longrightarrow} G_{n+1}.$$

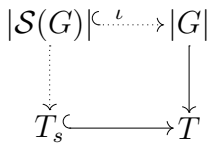
3 Transactional Graph Transformation Systems

In this section we introduce the basics of transactional graph transformation systems. After discussing the typing discipline which allows to distinguish between stable and unstable items in a given typed graph, we show how this can be used to define a notion of transaction.

The distinction between stable and unstable items is induced by specifying a subgraph of the type graph, which is intended to represent the stable types.

Definition 3.1 [Transactional GTS] A *transactional* GTS is a pair $\langle \mathcal{G}, T_s \rangle$, where $\mathcal{G}$ is a T-typed GTS and T_s is a subgraph of the type graph T of $\mathcal{G}$, called the *stable type graph*.

Given a graph G typed over T we can single out its stable part $\mathcal{S}(G)$, i.e., the subgraph consisting of its stably-typed items only. Formally, $\mathcal{S}(G)$ can be defined as the graph typed over T_s obtained by considering the pullback



Without loss of generality, we will assume a concrete choice for $\mathcal{S}(G)$, by imposing that the morphism ι in the pullback diagram above is an inclusion.

We say that a graph is stable if it consists only of stable items. This is formalised in the next definition.

Definition 3.2 [stable graph] A T-typed graph G is called *stable* if $|\mathcal{S}(G)| = |G|$ (i.e., if the morphism ι in the pullback diagram is the identity). It is called *unstable* otherwise.

It can be shown that the above transformation is functorial: given a morphism of T-typed graphs $f : G \to H$, the transformation above uniquely induces a morphism $\mathcal{S}(f) : \mathcal{S}(G) \to \mathcal{S}(H)$ (which is, given the concrete choice for $\mathcal{S}(G)$, the restriction of f to $\mathcal{S}(G)$). The corresponding functor $\mathcal{S} : T\text{-}\mathbf{Graph} \to T_s\text{-}\mathbf{Graph}$ is called *stabilising functor*.

The stabilising functor can be applied point-wise to any production of a given T-GTS, thus producing a GTS typed over the stable type graph.

Definition 3.3 [stabilised GTS] Given a T-typed T-GTS $\langle \mathcal{G}, T_s \rangle$, the *stabilised* GTS $\mathcal{S}(\mathcal{G})$ is given by $\langle T_s, P, \pi' \rangle$, where $\pi'(q) = \mathcal{S}(\pi(q))$ for any $q \in P$.

The functor $\mathcal{S}$, when applied to a derivation in a given T-GTS $\langle \mathcal{G}, T_s \rangle$, produces a derivation in $\mathcal{S}(\mathcal{G})$. An indirect proof of this fact can be obtained by observing that there exists a typed GTS morphism $f : \mathcal{G} \to \mathcal{S}(\mathcal{G})$, in the sense of [1], which essentially forgets about the non-stable items. Then, using the fact that GTS morphisms are simulations, one can infer the result below.

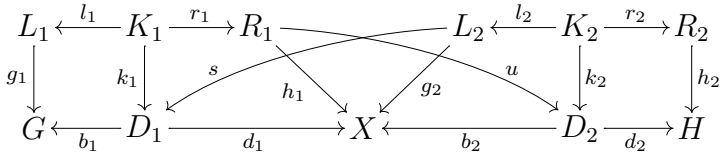


Fig. 1. Sequential independent derivations.

Proposition 3.4 *Let $\langle \mathcal{G}, T_s \rangle$ be a* T-GTS *and let $d = G_0 \overset{q_1,g_1}{\Longrightarrow} G_1 \overset{q_2,g_2}{\Longrightarrow} \ldots \overset{q_n,g_n}{\Longrightarrow} G_n$ be a derivation in $\mathcal{G}$. Then*

$$\mathcal{S}(d) = \mathcal{S}(G_0) \overset{q_1,\mathcal{S}(g_1)}{\Longrightarrow} \mathcal{S}(G_1) \overset{q_2,\mathcal{S}(g_2)}{\Longrightarrow} \ldots \overset{q_n,\mathcal{S}(g_n)}{\Longrightarrow} \mathcal{S}(G_n)$$

is a derivation in $\mathcal{S}(\mathcal{G})$.

Let us come to the definition of a transaction in a T-GTS $\langle \mathcal{G}, T_s \rangle$. Inspired by the approach for Petri nets proposed in [3] and extended to nets with read arcs in [4], we introduce *stable steps*, *stable transactions* and *abstract stable transactions*. In the following $\langle \mathcal{G}, T_s \rangle$ is a fixed T-GTS.

A stable step is, intuitively, a computation which starts and ends in stable states. Moreover, once generated, stable items are "frozen", in the sense that they cannot be read or consumed by other productions inside the same step. Therefore, the dependencies between productions occurring in a step are induced by unstable items: this implies that at the abstract level, where unstable items are forgotten, all such productions will be applicable in parallel.

To give a formal definition we need to briefly review some notions. A derivation $G \overset{q_1,g_1}{\Longrightarrow} X \overset{q_2,g_2}{\Longrightarrow} H$ as in Figure 1 is called *sequential independent* [6] if there are two morphisms $s : L_2 \to D_1$ and $u : R_1 \to D_2$ such that $d_1 \circ s = g_2$ and $b_2 \circ u = h_1$. Intuitively, the images in X of the left-hand side of q_2 and of the right-hand side of q_1 overlap only on items that are preserved by both derivation steps. In this case we can apply the two productions in the reverse order, obtaining derivation $G \overset{q_2,g_2'}{\Longrightarrow} X' \overset{q_1,g_1'}{\Longrightarrow} H$ and we can also apply them concurrently, obtaining a parallel direct derivation $G \overset{q_1+q_2,g}{\Longrightarrow} H$.

Definition 3.5 [stable step] A *stable step* is a derivation $d = G_0 \overset{q_1,g_1}{\Longrightarrow} G_1 \overset{q_2,g_2}{\Longrightarrow} \ldots \overset{q_n,g_n}{\Longrightarrow} G_n$ which enjoys the following properties:

(i) G_0 and G_n are stable graphs;

(ii) the derivation $\mathcal{S}(d)$ is equivalent to a parallel direct derivation $\mathcal{S}(G_0) \overset{q_0+\ldots+q_n,\mathcal{S}(g)}{\Longrightarrow} \mathcal{S}(G_n)$ in $\mathcal{S}(\mathcal{G})$.

Definition 3.6 [stable transaction] A *stable transaction* is a stable step $d = G_0 \overset{q_1,g_1}{\Longrightarrow} G_1 \overset{q_2,g_2}{\Longrightarrow} \ldots \overset{q_n,g_n}{\Longrightarrow} G_n$ such that, if $\mathcal{S}(G_0) \overset{q_0+\ldots+q_n,\mathcal{S}(g)}{\Longrightarrow} \mathcal{S}(G_n)$ in $\mathcal{S}(\mathcal{G})$ is the induced parallel derivation, then

(i) g is an epimorphism;

(ii) any intermediate graph G_i $(i \neq 0, n)$ is not stable.

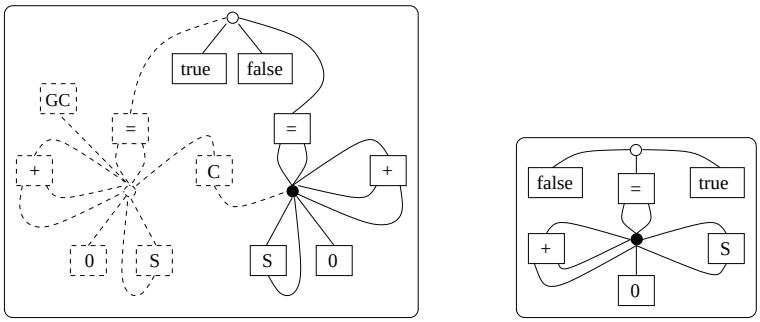


Fig. 2. The type graph (left) and its stable component (right).

By condition (i), the start graph contains exactly what the transaction needs to be brought to a successful end, while by condition (ii) no sub-derivation of d is a transaction, thus guaranteeing atomicity.

Actually, since we are considering a concurrent model of computation, the fact that all the intermediate graphs are not stable should not be related to the specific order in which productions are applied. Rather, this property should still hold for any derivation which is obtained from the original one by exchanging independent steps of computation, i.e., any *shift-equivalent* (see, e.g., [13,6]) derivation. When combining shift-equivalence with an equivalence which abstracts also with respect to the concrete identities of items in the involved graphs, i.e., which considers graphs up to isomorphism, we obtain the so-called *abstract truly-concurrent equivalence* [6]. The equivalence class of a derivation d with respect to such equivalence will be denoted by $[d]_c$ and called *abstract trace*.

Definition 3.7 [abstract stable transaction] An *abstract stable transaction* is an abstract trace $[d]_c$, such that for any $d' \in [d]_c$ the derivation d' is a stable transaction.

It follows from the definition that if two abstract stable transactions can be applied in parallel to a stable graph, then all the direct derivations of either of them are independent of the direct derivations of the other one. Thus, as desired, the transactions can be interleaved in an arbitrary way.

Clearly, a more manageable characterisation of abstract stable transactions would be desirable: even if the corresponding theory is not yet completely developed, we will sketch in the concluding section how such a characterisation could be obtained by means of suitable graph processes.

4 A simple example on integer equality

We now present a simple GTS for testing the equality between integer expressions involving natural numbers represented as sequences $\mathsf{S}(\mathsf{S}(\ldots\mathsf{S}(\mathsf{0})\ldots))$ and a sum operator. Despite its small size we hope that this example will pinpoint the key features of our approach.

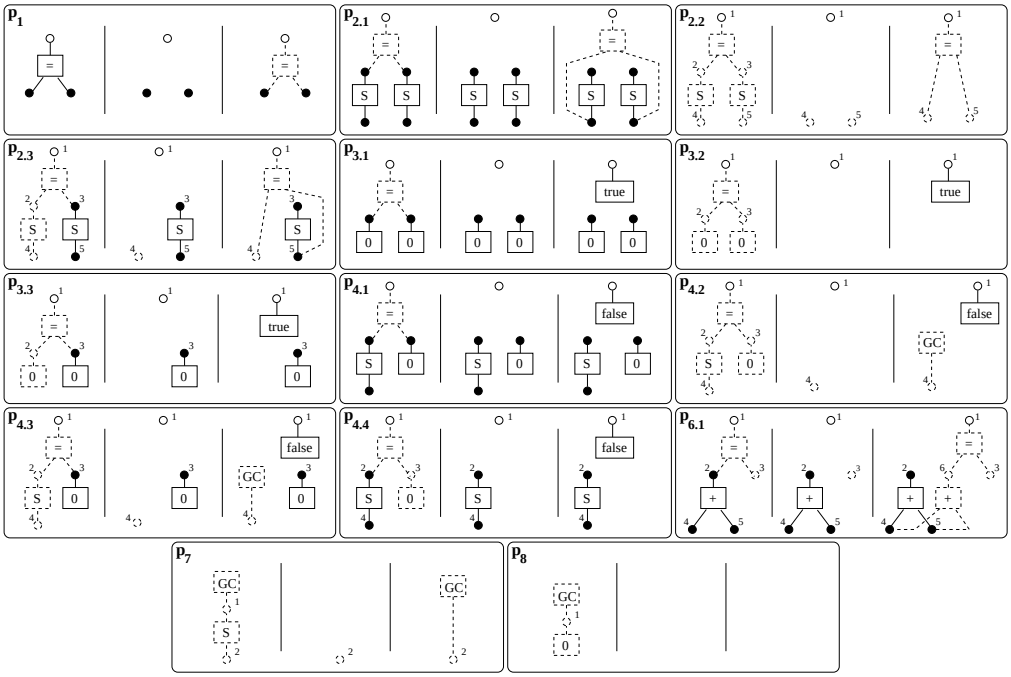
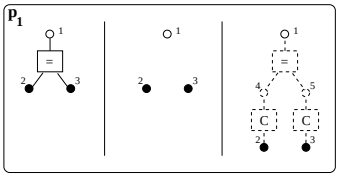


Fig. 3. Productions for the equality operator and for garbage collection.

The type (hyper-)graph and its stable subgraph are depicted in Figure 2. Explicitly stated, the dashed items (dashed boxes representing (hyper-)edges and dashed circles for nodes) are not stable. Notice that, as usual for hypergraphs, each edge is connected to an ordered list of nodes. The order is implicit in our drawings: the first connection leaves the edge from the top, and the others follow counter-clockwise.

As a sample expression to be evaluated we consider $\mathsf{S}(\mathsf{S}(\mathsf{0})) + \mathsf{S}(\mathsf{0}) = \mathsf{S}(\mathsf{0})$, as represented by the stable graph G_0 on the left of Figure 5. For the sake of simplicity, G_0 is a tree, but the system also works for acyclic graphs, where subexpressions can be shared. In order to ensure that a shared subexpression is not affected by the evaluation of an outer expression, some rules duplicate the part of the structure that needs to be accessed in a destructive manner.

Let us consider the production p_1 in Figure 3: the graph on the left (center, right) represents the left-hand side (interface and right-hand side, respectively) of the production. Note that, according to the shape of the type graph, an unstable operator can be connected to a stable node only through an additional unstable node and a C-labelled edge. In order to simplify the presentation, such node and the C-labelled edge will be omitted in the figures. For instance, production p_1 should be read as



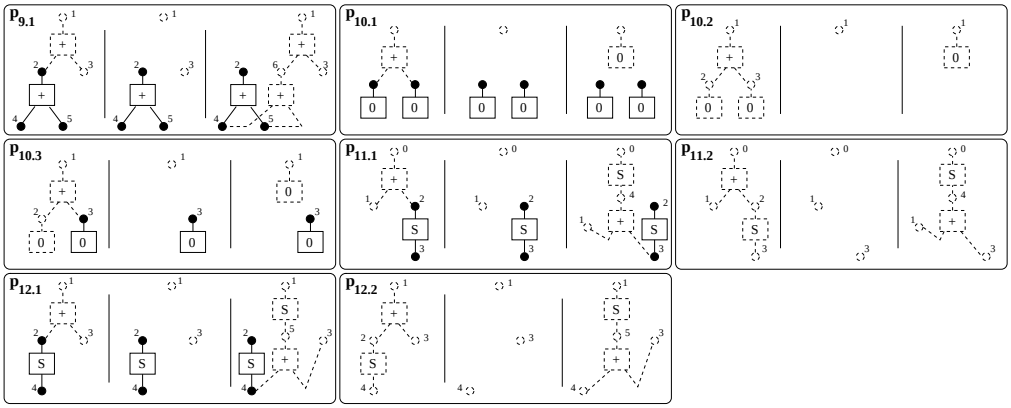


Fig. 4. Productions for the sum operator.

Intuitively, a computation proceeds as follows. The only production that can be applied to a stable graph like G_0 is p_1, which starts a transaction by replacing the stable edge $=$ with its unstable, dashed counterpart $\vdots$. Next, conceptually, the equality operator traverses the expression (see production $p_{2.1}$), triggering, whenever it is needed, the evaluation of the sum operators by generating an unstable copy of them (production $p_{6.1}$). In turn, the evaluation of the sum generates as its result a chain of unstable successor operators (see productions $p_{12.1}$ and $p_{11.1}$ in Figure 4), recursively triggering the evaluation of nested additions (as in production $p_{9.1}$), and stopping when both arguments are zero (as in production $p_{10.1}$). The equality operator can then proceed, consuming the chain of unstable successors generated by the sum, till when either one or two zeros are reached. At this point the boolean result is generated (as in productions $p_{3.2}$), and, if needed, the "garbage collection" of the remaining unstable items is started (productions $p_{4.2}$, p_7 and p_8).

The presence of stable and unstable versions of both operators and constants motivates the existence of several variants for each production. For example, all the productions $p_{2.1}$, $p_{2.2}$ and $p_{2.3}$ (as well as its symmetric version $p_{2.4}$ which is not depicted) basically replace, conceptually, the subexpression $\mathsf{S}(\mathsf{x}) = \mathsf{S}(\mathsf{y})$ by the equivalent one $\mathsf{x} = \mathsf{y}$. Such productions do not have the same structure, though, because stable S-edges have to be preserved, as they may belong to a shared subexpression, while unstable S-edges have to be deleted, as they should not appear in the final state: this can be done safely, because unstable S-edges are generated by the productions in a way that guarantees that they are never shared.

The same observation applies to each other group of productions, like $p_{3.\_}$ (modelling the rule $\mathsf{0} = \mathsf{0} \rightsquigarrow \mathsf{true}$), $p_{4.\_}$ (modelling $\mathsf{S}(\mathsf{x}) = \mathsf{0} \rightsquigarrow \mathsf{false}$), and so on. Notice that several rules have a symmetric version (exchanging the left and right arguments of the main binary operator) which are not depicted. For example the productions in the missing $p_{5.\_}$ family model the evaluation of $\mathsf{0} = \mathsf{S}(\mathsf{x})$ to false. They are obtained from the $p_{4.\_}$ productions by exchanging the arguments of the equality operator.

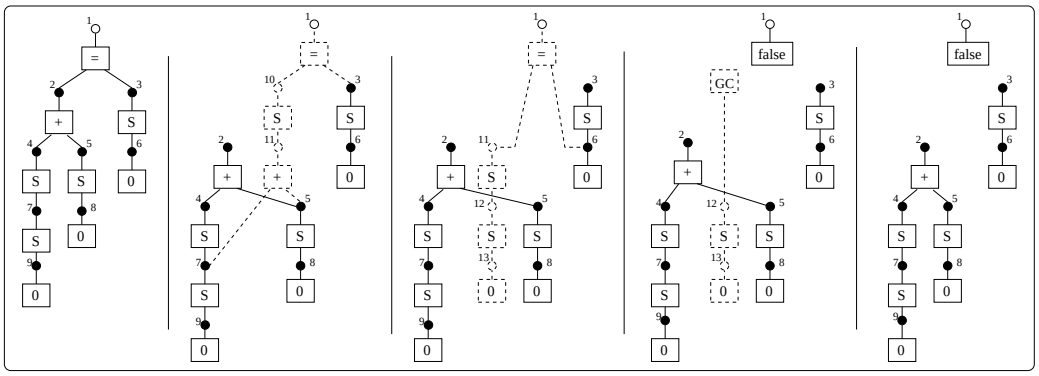


Fig. 5. An expression (left), some unstable states (center), and the result (right).

Some states of the derivation starting from $\mathsf{S}(\mathsf{S}(\mathsf{0})) + \mathsf{S}(\mathsf{0}) = \mathsf{S}(\mathsf{0})$ and reaching the final state, which represents the result false, are depicted in Figure 5. From the starting state productions p_1, $p_{6.1}$ and $p_{12.1}$ are applied, reaching the second state; next, applying productions $p_{12.1}$, $p_{11.1}$, $p_{10.1}$ and $p_{2.3}$ the third state is reached; then, applying $p_{4.3}$ the fourth state is reached; and finally the application of p_7 and p_8 produces the final state. Note that all the intermediate states are unstable, due to the presence of at least one unstable item. Hence, the only visible states in the derivation, which can be shown to be a stable transaction, are the initial and final ones.

The corresponding abstract stable transaction includes all the derivations which are obtained by switching sequential independent direct derivations, such as the one which applies the productions in the order p_1, $p_{6.1}$, $p_{12.1}$, $p_{2.3}$, $p_{12.1}$, $p_{4.3}$, $p_{11.1}$, $p_{10.1}$, p_7 and p_8. It can be shown that each abstract stable transaction in the proposed system performs the evaluation of exactly one equality operation, building as an unstable intermediate structure the result of the sum operators, and destroying them at the end.

5 Future perspectives

This paper introduces *transactional graph transformation systems*, a formalism which is intended to enrich the classical DPO approach to graph rewriting with a built-in notion of transaction. Our work so far outlined the basic notions underlying the framework, and further results are now needed to bring the theory to full maturity.

Abstract GTS associated to a transactional GTS
A first line of research concerns the definition of the abstract GTS associated to a T-GTS. As discussed in the paper, a T-GTS can be seen at two different levels of abstraction. It can be viewed as a standard graph transformation system, where both stable and unstable states, and thus also the internal structure of transactions, are visible. But we can also abstract away from the unstable states and observe only complete transactions. Formally, this gives

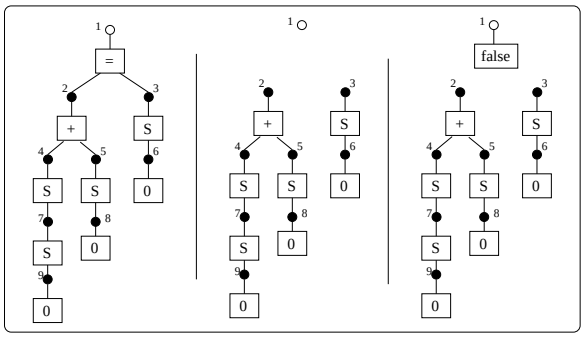


Fig. 6. The abstract production induced by the transaction of Figure 5.

rise to another GTS, whose definition requires the notion of the production *induced* by a derivation sequence, a known construction in the literature. The production induced by a derivation $d : G_0 \Rightarrow^* G_n$ has G_0 as left-hand side and G_n as right-hand side. The interface graph is the subgraph of G_0 which, intuitively, consists of all the items which are preserved by all the direct derivations occurring in the sequence.

Definition 5.1 [Abstract GTS] Let $\langle \mathcal{G}, T_s \rangle$ be a T-GTS. Given an abstract stable transactions $[d]_c$, a production induced by d is called *abstract production* for the transaction $[d]_c$.

The *abstract* GTS associated to the given T-GTS, denoted by $\mathcal{A}(\langle \mathcal{G}, T_s \rangle)$, is the GTS $\langle T_s, P', \pi' \rangle$ where P' is the set of abstract stable transactions $[d]_c$ and $\pi'([d]_c)$ is an abstract production for the transaction $[d]_c$.

As an example, the abstract production that corresponds to the transaction depicted in Figure 5 is shown in Figure 6.

As it should be evident from the proposed example, the abstract GTS associated to a T-GTS can have, in general, an infinite number of productions. Indeed, the notion of transaction allows one to model an abstract system with infinitely many productions by means of a lower level system, with a finite number of productions.

From a theoretical point of view the definition of the abstract GTS associated to a T-GTS might not be yet fully satisfactory, since it lacks an extensional presentation, as it is offered by categorical means in terms of adjunctions.

However, note that any GTS $\mathcal{G}$ can be naturally seen as a T-GTS $\langle \mathcal{G}, T \rangle$ by considering the entire type graph T as stable. Hence, turning the classes of GTSs and of T-GTSs into categories **GTS** and **TGTS**, respectively, there would be an obvious inclusion functor of **GTS** into **TGTS**. Thus, a solid justification for the construction of the abstract GTS associated to a T-GTS could come from a characterisation of the mapping $\mathcal{A}$ (Def. 5.1) as a functor from the category of T-GTSs to the category of GTSs, and from its characterisation as the right adjoint to the inclusion functor in the opposite direction. Intuitively, this would mean that, given a T-GTS $\langle \mathcal{G}, T_s \rangle$, the abstract GTS $\mathcal{A}(\langle \mathcal{G}, T_s \rangle)$, given in Definition 5.1, is the "best approximation" of $\langle \mathcal{G}, T_s \rangle$ in the category **GTS**.

We foresee two possible ways of proving a result of this kind:

(i) Freely generated category of systems with transactions as productions. Inspired by the work on zero-safe Petri nets [3], the idea consists of freely generating complex computations of a given T-GTS, starting and ending in stable states, by suitably composing its original productions. The considered composition operation should act differently on the stable and unstable items, composing the former in parallel and the latter sequentially. Moreover, it should be subject to axioms which identify computations differing only for the order of independent steps. In this setting transactions would be identified as computations that cannot be decomposed as the parallel composition of (non trivial) computations.

(ii) Transactions as special processes. Graph processes [5] are structures which provide a truly concurrent representation of a deterministic computation in a given GTS, by explicitly representing the start and ending state, as well as all the intermediate items produced in the computation and their causal dependencies. A transaction can be characterised as a process which starts and ends in stable states, where only direct causal dependencies between stable items exist, and which satisfies suitable atomicity properties.

In both cases, it seems that the appropriate choice of morphisms in the category of T-GTS should be that of implementation or refinement morphisms [10,11], which allow to map a single production into a computation.

Multi-level transactional GTSs

Another issue to be addressed concerns the "binary" distinction between stable and unstable items, which can be unsatisfactory in certain situations. In fact, a system can be viewed at several levels of abstractions, and what appears to be as an atomic production can be refined to a computation at a lower level and can be the building block of more complex transactions at a higher-level. In the proposed framework, the situation could be recast by replacing the stable/unstable dichotomy by a multi-layered structure, consisting of a set of graphs T_0, T_1, ..., T_n such that T_{i+1} is a subgraph of T_i, representing the stable types for layer i.

The functorial characterisation of abstract GTSs that we envision could also be helpful to provide a modular semantics to the multi-layered T-GTSs.

Relations with refinement and modularity for GTSs

We do believe that our semantical analysis of transactional mechanisms in graph transformation is original. However, the same notion of abstract GTS calls for a comparison with the approaches to refinement and modularisation proposed in the literature (see [12] and the references therein).

Transactions could be exploited to simulate modules, since the atomicity of some computations is induced by the fact that some states are classified as non-observable or unstable at the abstract level. We leave to future work the further elaboration of these ideas, as well as a comparison with the literature.

Acknowledgments. We are mostly grateful to Roberto Bruni for insightful discussions and his careful reading of a preliminary version of this paper.

References

[1] P. Baldan, A. Corradini, and U. Montanari. Unfolding of double-pushout graph grammars is a coreflection. In Ehrig et al. [7], pages 145–163.

[2] R. Bardohl, M. Minas, A. Schurr, and G. Täntzer. Application of graph transformation to visual languages. In Ehrig et al. [8], chapter 3, pages 105–180.

[3] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Info. & Co.*, 156(1-2):46–89, 2000.

[4] R. Bruni and U. Montanari. Transactions and zero-safe nets. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Advances in Petri Nets: Unifying Petri Nets*, volume 2128 of *LNCS*, pages 380–426. Springer, 2001.

[5] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265, 1996.

[6] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In Rozenberg [14], chapter 3, pages 163–245.

[7] G. Ehrig, G. Engels, H.J. Kreowski, and G. Rozenberg, editors. *Proceedings of the International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *LNCS*. Springer, 1999.

[8] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.

[9] H. Ehrig, M. Pfender, and H.J. Schneider. Graph-grammars: An algebraic approach. In R.V. Book, editor, *IEEE Conf. on Automata and Switching Theory*, pages 167–180. IEEE Computer Society Press, 1973.

[10] M. Grosse-Rhode, F. Parisi Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [7], pages 368–382.

[11] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of graph transformation systems. *Mathematical Structures in Computer Science*, 6(6):613–648, 1996.

[12] R. Heckel, H. Ehrig, G. Engels, and G Täntzer. Classification and comparison of module concepts for graph transformation systems. In Ehrig et al. [8], chapter 17, pages 669–689.

[13] H.-J. Kreowski. *Manipulation von Graphmanipulationen*. PhD thesis, Technische Universität Berlin, 1977.

[14] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, 1997.

Graph Transformation Semantics for a QVT Language

Arend Rensink [1] Ronald Nederpel

University of Twente, Computer Science Department
P.O.Box 217, 7500 AE Enschede, The Netherlands

Abstract

It has been claimed by many in the graph transformation community that model transformation, as understood in the context of Model Driven Architecture, can be seen as an application of graph transformation. In this paper we substantiate this claim by giving a graph transformation-based semantics to one of the original QVT language proposals; that is, any model transformation definition in the QVT language is translated to a graph production system whose effect is to apply that model transformation. The translation has been fully implemented.

1 Introduction

In order to better understand and structure the process of software engineering, the Object Management Group (OMG) has put forward the Model Driven Architecture (MDA) approach (see [22]), with as core concepts *metamodelling* and *model transformation*. The first of these is based on the insight that, in order to provide a unified view upon the software engineering process, it is necessary to organise and relate the different (visual and textual) languages used along the way, i.e., the languages in which the different design models and artefacts are written. The organisational structure proposed in MDA is the *metamodelling hierarchy*: there, the languages are understood as instances of a single top-level *meta-metamodel*, the MOF (see [20]). According to the terminology used in this context (to which we will adhere in this paper), the languages are called metamodels, and the artefacts written in those languages (including executable programs) are called models.

The second core concept in MDA, model transformation, is based on the insight that many if not all of the activities in software engineering involve the creation of new models (in the sense explained above) based on existing models; and that this "creation based on" can be interpreted as a *transformation* of the existing models

[1] The research described in this paper has been carried out in the context of the Duth NWO projects GROOVE (612.000.314) and GRASLAND (612.063.408).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

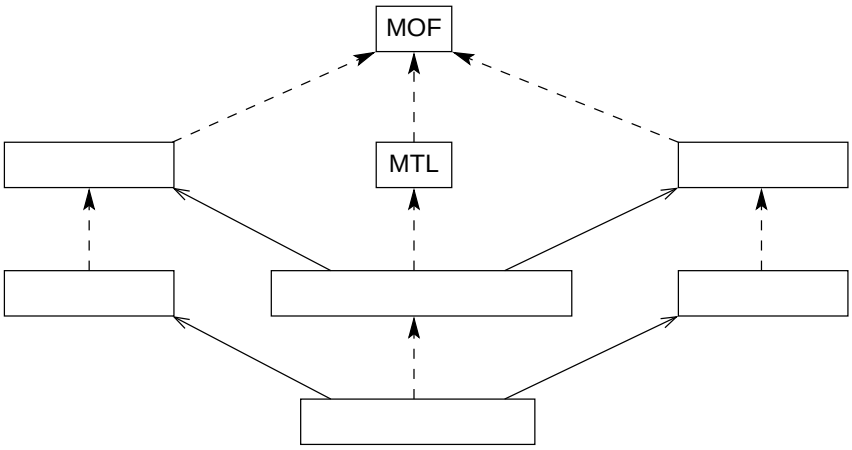


Fig. 1. Overview of MDA concepts

into the new. A further observation is that such transformations typically have some guiding principles that are not specific to the models, but rather can be articulated on the level of the metamodels involved. Such an "articulation of guiding principles" is called a *transformation rule*; a collection of transformation rules is called a *transformation definition*. See also [15] for a further discussion. Fig. 1 gives an overview of the resulting set of concepts.

Since the MOF provides a unifying view on the metamodels and (therefore, indirectly) the models, we have available the ingredients necessary to write down transformation definitions uniformly and, once that is done, to execute them automatically. This has led the OMG some time ago to request proposals for a *QVT language*, where QVT stands for Query/View/Transformation (see [21]). An overview of the submissions received on this call is given in [10]. In this paper we take one of these submissions, namely the one by IBM, DSTC and CBOP [6]; we call this the *Model Transformation Language* (MTL) in the remainder of this paper. In terms of the ontology proposed by Czarnecki and Helsen [4], important characteristics of MTL (which determine its suitability for the approach of this paper) are that it has *syntactic separation* of source and target model elements, that it is *unidirectional* and *declarative*. Moreover, it supports *traceability* of model transformations.

The essence of any language is its *semantics*, which captures the effect of "sentences" of the language. So, too, with MTL, where the "sentences" are transformation definitions. The effect of a transformation definition can be captured by a binary relation between the set of models residing under the source meta-model to the set of models under the target meta-model; in other words, a set of *transformation pairs* in the sense of Fig. 1. The semantics of MTL, therefore, is a partial mapping that assigns such binary relations to transformation definitions written in MTL. The partiality of the mapping is due to the fact that a syntactically valid transformation definition may fail to be semantically correct, for instance due to inconsistencies with the source or target metamodel; in such a case it has no semantic image. Formally, the semantic mapping is a partial function

$$[\![\_]\!] : \mathsf{MTL}[A|B] \rightharpoonup \mathbf{2}^{\mathsf{Mod}[A]\times\mathsf{Mod}[B]}$$

where $\mathsf{MTL}[A|B]$ is the set of transformation definitions in MTL with source meta-model A and target metamodel B, and $\mathsf{Mod}[A]$ (resp. $\mathsf{Mod}[B]$) stands for the set of models under meta-model A (resp. B).[2] For any transformation definition $D \in \mathsf{MTL}[A|B]$, we write $[\![D]\!]$ for the set of transformation pairs generated by D (hence $[\![D]\!] \subseteq \mathsf{Mod}[A] \times \mathsf{Mod}[B]$). In many cases, $[\![D]\!]$ is in fact total and one-to-one, meaning that it can also be seen as a function $\mathsf{Mod}[A] \to \mathsf{Mod}[B]$; then we write $[\![D]\!](M) \in \mathsf{Mod}[B]$ to denote the target model obtained by applying D to the source model $M \in \mathsf{Mod}[A]$.

Unfortunately, as is the rule rather than the exception, in [6] the semantics of MTL is not defined formally but just described in natural language. This means that the basis for reasoning about, verifying and guaranteeing correctness of the model transformations, which is also mentioned in [18] as one of the success criteria for a transformation language or tool. This is the problem we set out to address in the current paper, which is based on the MSc thesis [19] by the second author. In the next section we describe the approach we have followed, of which the essence is that we translate model transformation definitions to graph production systems, which do have the desired formal basis. In Sect. 3 we show the approach on the basis of an example (actually taken from the QVT submission [6]). In Sect. 4 we evaluate the achievements and discuss some related approaches. Note that within the scope of this paper it is quite impossible to describe the actual semantics in any detail. However, see [19] for a full description.

2 Approach

There is already a large body of research that supports the use of graph transformation techniques as a basis for the formal semantics of model transformation, both practical (in the sense of tools, e.g. [12,2,26,3]) and theoretical (for instance confluence and termination properties as in [17,8]). In this paper we also follow that route in order to define semantics of MTL. We will omit most of the theoretical background; see, e.g., [24] for an extensive discussion.

2.1 Principles

Graph transformation works on the basis of *graph production rules* defined over a given universe of graphs, Graph. Finite sets of graph production rules, called *graph production systems*, are used as transformation specifications. Each rule $\mathcal{R}$ describes a single-step transformation of certain graphs into others, and so defines a binary relation $\to_{\mathcal{R}}$ over Graph such that $G \to_{\mathcal{R}} H$ if and only if $\mathcal{R}$ turns G into H. For a production system $\mathcal{P}$, the transitive closure of the union of $\to_{\mathcal{R}}$ for all $\mathcal{R} \in \mathcal{P}$ then gives rise to a partial ordering over Graph, which we denote $\leq_{\mathcal{P}}$. Finally, we say that $\mathcal{P}$ *eventually transforms* G *into* H, denoted $G \Rightarrow_{\mathcal{P}} H$, if $G \leq_{\mathcal{P}} H$ and H is a $\leq_{\mathcal{P}}$-maximal element of Graph, i.e., it cannot be transformed further. (In

[2] As usual, $\mathbf{2}^X$ denotes the *powerset* of a set X, i.e., the set of all subsets of X.

other words, $G \Rightarrow_{\mathcal{P}} H$ if H is reachable from G along a path of $\rightarrow_{\mathcal{R}}$-single-step transitions with $\mathcal{R} \in \mathcal{P}$, and $H \not\rightarrow_{\mathcal{R}}$ for all $\mathcal{R} \in \mathcal{P}$.)

Let us write Rule for the universe of graph production rules. In order to define the semantics of MTL using these principles, we define the following ingredients:

- For any meta-model A, a set $\mathsf{Graph}[A] \subseteq \mathsf{Graph}$ and an injective mapping $g_A \colon \mathsf{Mod}[A] \rightarrow \mathsf{Graph}[A]$ to connect the MDA model world to the graph world. The mapping is in general not surjective, meaning that the inverse may be undefined.
- For any pair of meta-models A and B, a set of "linked graphs" $\mathsf{Graph}[A|B] \subseteq \mathsf{Graph}$ that essentially consist of pairs of graphs from $\mathsf{Graph}[A]$ and $\mathsf{Graph}[B]$, together with two graph production systems, $\mathsf{Left}[A|B], \mathsf{Right}[A|B] \subseteq \mathsf{Rule}$ that "extract" the constituent graphs, such that for any $G \in \mathsf{Graph}[A|B]$, there are unique $H_A \in \mathsf{Graph}[A]$ and $H_B \in \mathsf{Graph}[B]$ with $G \Rightarrow_{\mathsf{Left}[A|B]} H_A$ and $G \Rightarrow_{\mathsf{Right}[A|B]} H_B$.
- A mapping $\mathsf{GPS} \colon \mathsf{MTL}[A|B] \rightarrow \mathbf{2}^{\mathsf{Rule}}$ that yields a graph production system from an arbitrary MTL transformation definition $D \in \mathsf{MTL}[A|B]$, such that for any $G \in \mathsf{Graph}[A]$ there is a unique linked graph $H \in \mathsf{Graph}[A|B]$ that satisfies $G \Rightarrow_{\mathsf{GPS}(D)} H$; and, moreover, for which $H \Rightarrow_{\mathsf{Left}[A|B]} G$.

We then define, for any $D \in \mathsf{MTL}[A|B]$ and $M \in \mathsf{Mod}[A]$,

$$[\![D]\!](M) = g_B^{-1}(H) \quad \text{where } g_A(M) \Rightarrow_{\mathsf{GPS}(D)} G \Rightarrow_{\mathsf{Right}[A|B]} H \ .$$

In words, the semantics of D is defined, for an arbitrary model $M \in \mathsf{Mod}[A]$, by first mapping M to the graph $g_A(M) \in \mathsf{Graph}[A]$, then transforming that to a linked graph $G \in \mathsf{Graph}[A|B]$ using the dedicated graph production system $\mathsf{GPS}(D)$, then extracting $H \in \mathsf{Graph}[B]$ from this combined graph, and finally converting H to a model $g_B^{-1}(H) \in \mathsf{Mod}[B]$. Due to the fact (noted above) that g_B may fail to be surjective, in general it is possible that $g_B^{-1}(H)$ is not defined, in which case neither is $[\![D]\!](M)$. We currently have no way to detect or ensure statically which D give rise to a total function $[\![D]\!]$.

2.2 Implementation

The steps described above have all been implemented, resulting in the tool chain depicted in Fig. 2, where boxes are products and ovals represent processes steps. The fat boxes are the inputs and output products of the chain; the thin ones are auxiliary products that are both produced and consumed in the course of the transformation. The grey ovals were implemented in the course of this research; the white ovals, labelled "*apply*", stand for the application of a graph production system using a pre-existing tool. We will now discuss the individual steps and some relevant design decisions.

(1) In general the metamodels are, as shown in Fig. 1, instances of the MOF. However, the MOF is itself still quite extensive; for the purpose of this paper we concentrated on a subset, depicted in Fig. 3. For the purpose of this work the metamodels were created as class diagrams and stored in (a version of)

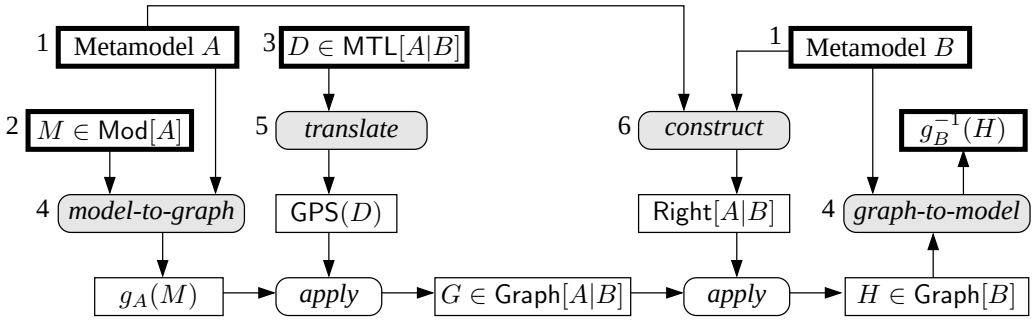


Fig. 2. Transformation tool chain

XMI using Borland Together.

(2) The models, likewise, were created (as object diagrams) and stored in XMI using Together. Note that we need to have the models *as instances of their metamodels*, i.e., the instantiation relation depicted in Fig. 1 needs to be clear.

(3) The concrete and abstract syntax of the language MTL are defined in [6] by an EBNF grammar, resp. an abstract syntax meta-model. An example is given there also, which we use as well in the next section. Unfortunately, here a problem exists, in that the example cannot be parsed according to the grammar and cannot be matched to the abstract syntax; nor do the concrete and abstract syntax seem to be consistent. In [19] we report a host of problems and propose solutions; in the remainder of this paper we follow [19] in actually working from the proposed solutions.

For the purpose of this paper we just show the main rule of the grammar, which defines a transformation rule:

```
trule ::= "RULE" rname formals ( relatedRules )*
          "FORALL" ranges ( "WHERE" conjunct )?
          "MAKE" targets ( "LINKING" trackingUses )?
          SEMI
```

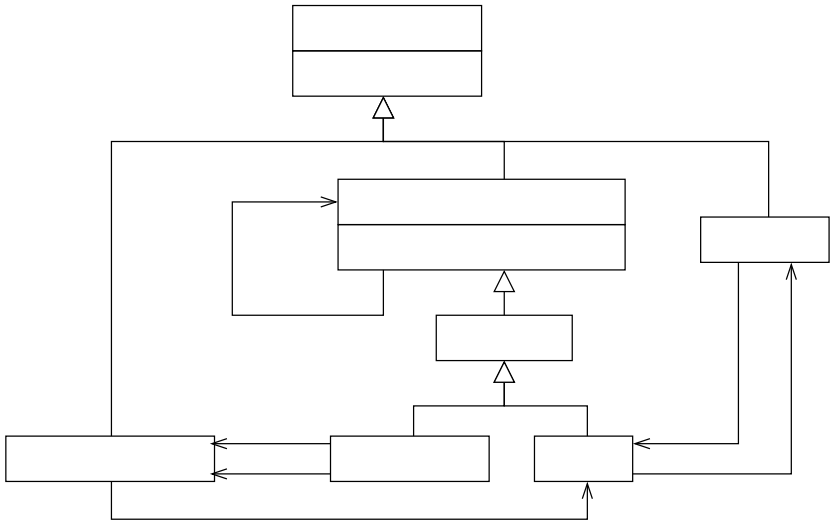


Fig. 3. Fragment of the MOF used in this paper

The lowercase identifiers are further nonterminals; the quoted strings and uppercase identifier are terminals of the concrete syntax; the rest is EBNF syntax.

- The `rname` is the rule name; `formals` are formal names used in the `FORALL` and `MAKE` parts; and the `relatedRules` provide a reuse (i.e., inheritance) mechanism between rules.
- The `FORALL` part specifies where the rule applies, i.e., what needs to be matched in the source model. In the `ranges` more formal names are introduced, referring either to the elements in the source model that are to be matched, or to elements of the target model that were already created. The `WHERE` clause essentially contains equations based on these names and connected model elements; these serve to constrain the local structure of the model where the rule is to be applied.
- the `MAKE` part specifies what needs to be created in the target model whenever a match for the `FORALL` part is provided: the `targets` part lists new model elements and their relation to the existing elements matched in the `FORALL`. The `LINKING` clause can be used to specify traceability information, in the form of links between elements of the source model and the newly created target elements.

(4) In this step we translate models to graphs and back. For the purpose of the current paper, we have chosen a very "poor" graph formalism: the graphs in Graph just consist of unlabelled nodes and labelled directed binary edges; parallel edges, attributes, hierarchy and typing are not included. This is the type of graphs supported by the GROOVE tool [23]; the choice was determined pragmatically by the local availability of the tool. It should be noted that we do not consider the choice of graph formalism to be a relevant part of the research reported in this paper; in Sect. 4 we discuss alternatives.

In the representation of models as graphs, embodied by the functions g_A discussed above, we have had to make some representation choices. In particular, we chose to represent MOF `Class` instances as nodes inscribed with the names of their direct types, bidirectional `Associations` as pairs of labelled edges in opposite directions, and `Attributes` as single edges. Supertypes are not represented in the models; instead, any rule dealing with a metamodel type has to be duplicated for all (combinations of) subtypes.

(5) Essentially, every `trule` of the transformation definition is turned into a graph production rule $\mathcal{R}$. Its left hand side is generated from the `FORALL` clause: each formal name in `trule` gives rise to a graph node, and the `WHERE` clause is translated to edges between those nodes. $\mathcal{R}$'s right hand side is a supergraph of the left hand side (so nothing is deleted), and contains the following further elements (which will therefore be created upon rule application):

(a) A node labelled `LINK`, with edges to those nodes that match source and target model elements involved in the rule, as well to another new node labelled with the rule name. This "rule node" indicates that the rule has been applied here. A correspondig *negative application condition* is also added to the rule, to prevent it from being applied more than once.

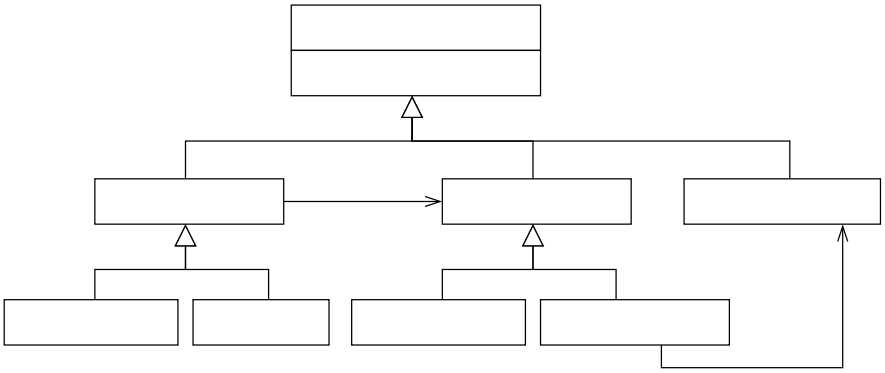


Fig. 4. Simplified UML metamodel

(b) Nodes and edges corresponding to the target model elements named in the MAKE part.
(c) Nodes for each of the elements named in `trackingUses`, to model the `LINKING` clause, as well as edges connecting to the relevant source and target model nodes. These nodes and edges eventually make up the linking structure in Graph$[A|B]$.

By using this setup, the rules in GPS(D) only *add* elements to the graphs, rather than doing in-place model transformation. This makes it quite easy to mimic each MTL rule. As a consequence, the resulting graph will be a combination of the source and target graphs linked together by `LINK` and `trackingUse` nodes and their corresponding edges.

(6) The Left$[A|B]$- and Right$[A|B]$-production systems extract the A- and B-instance graphs from the linked graph produced by the rules discussed above. This is done by stripping away the nodes and edges introduced by the (a) and (c)-parts of the rules, as described in (5) above, as well as the elements of the target (for Left) resp. source model (for Right) that they link to.

3 Example

To illustrate the steps described in the previous section, we take an example from [6] of *UML-to-Java* model transformation. Another example is presented in [19]. Fig. 4 shows the (hugely simplified) UML metamodel used in this example.

Transforming UML to Java is not very challenging, since the models are already very close: mainly it is a matter of changing the metamodel names, for instance from `UMLClass` to `JavaClass`. The most interesting part of the transformation is the required addition of a constructor, which is mandatory according to the Java metamodel used. This transformation is specified by the following couple of rules in the MTL transformation definition:

```
RULE UmlClassifierToJavaClass(uc,jc)
FORALL UMLClassifier uc
MAKE JavaClass jc,
     jc.name = uc.name
```

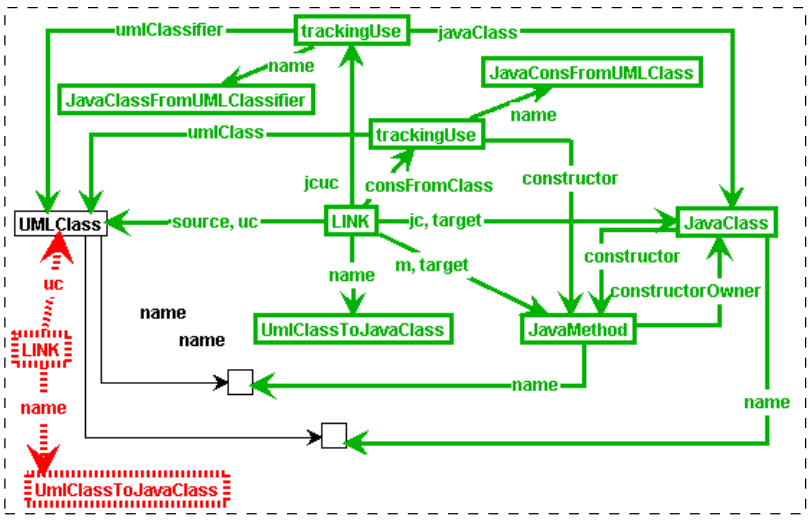


Fig. 5. Transformation rule (in GROOVE) for a `UMLClass`, introducing a constructor

```
LINKING JavaClassFromUMLClassifier jcuc
WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;

RULE UmlClassToJavaClass(uc,jc)
EXTENDS UmlClassifierToJavaClass(uc,jc)
FORALL UmlClass uc
MAKE JavaMethod m,
     m.name = uc.name,
     jc.constructor = m
LINKING JavaConsFromUMLClass consFromClass
WITH consFromClass.constructor = m,
     consFromClass.umlClass = uc;
```

The second of these rules give rise to the graph production rule displayed in Fig. 5 (in the GROOVE format). The fat grey nodes and edges (green, in a coloured representation) are to be added, the even fatter dashed part (red) is the negative condition that prevents the rule from being applied twice. The rule is layed out so that source model nodes are on the left, auxiliary nodes (parts (a) and (c) in step (5) described above) are in the middle and target nodes are on the right of the figure.

Other rules, such as the one for transforming the attributes, actually use the `trackingUse` information introduced by the above rule to link the new elements to the correct nodes. Finally, Fig. 6 shows an example source and target graph. We omit the corresponding models and the linked graph for lack of space.

4 Conclusion

We briefly evaluate the work presented in this paper, suggest some possible extensions and discuss related work.

Evaluation. The contribution of the research reported in this paper is not theoret-

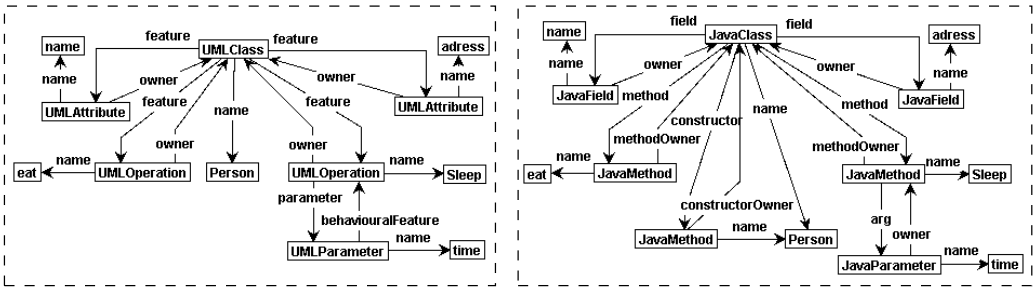


Fig. 6. Example source and target graphs

ical but rather in the nature of a proof-of-concept: none of the steps implemented is new, rather the novelty lies in having actually carried through all of them consistently, which among other things involves taking many small design decisions. Based on this exercise we can confirm some widespread beliefs.

- Even using a very "poor" graph model, it is possible to give a semantics to a sizeable fragment of a full-blown model transformation language;
- Defining a semantics in this way is a useful step, because it helps to uncover flaws and ambiguities in the language definition;

The language studied in this paper, as described in the official submission [6] to the QVT request for proposals, indeed contained many flaws and ambiguities: the full thesis [19] reports 11 issues in the grammar that either prevent the example in the submission itself from being parsed or give rise to parsable texts that are clearly unintended, and 7 issues in the abstract syntax metamodel that are either poorly documented or inconsistent with the grammar and the example.

Possible extensions and future work. There are several directions in which this work should be extended and improved before we can claim to have a full MTL semantics.

- The work reported in this paper is pragmatic rather than theoretic. In particular, the mapping GPS discussed in Sect. 2, which maps MTL transformation definitions to graph production systems, has not been worked out in full formal detail; instead it has been "defined" in the form of a tool implementation.
- The fragment of the MOF that we have treated (see Fig. 3) should be extended. For instance, one of the more prominent features currently missing is association ordering. However, we believe that this requires no fundamental change to the framework: one just has to extend the model-to-graph and graph-to-model conversions with a suitable graph representation of the order, and take this into account in the transformation rules.
- MTL rule paramaterisation and generalisation are not supported. Although this can be mimicked through syntactically copying and substituting the rules, that is a very poor solution which, for one thing, blows up the number of rules. To cope with this in a more fundamental way, one can use for instance node type inheritance in the graph transformation formalism, as proposed in [1,27].

- Attributes are supported only poorly in the graph transformation formalism we have used (however, see [14] for initial ideas on improving this). Choosing a category of graphs (Graph) that is closer to the category of models (Mod), such as attributed graphs and the transformation tool AGG [9,5], would improve the simplicity of both the function g_A that maps the model space into the graph space, and the function GPS that defines the actual graph production systems.
- The QVT proposal on which this research was based has now been subsumed by the actual QVT standard. A future iteration of this work should therefore be aimed at the actual standard.

Related work. Although there has been a lot of research on using graph transformation for model transformation, some of which we have reported in Sect. 2, we have not seen the question studied in this paper, namely to give a graph transformation semantics to a pre-existing model transformation language, addressed elsewhere. Instead, precisely the inverse trajectory has been followed in [11], where model transformations specified originally in a graph transformation formalism (viz., FuJaBA) are translated to the language MTL that we have also studied in this paper, after which they are interpreted by the tool Tefkat [7].

Another source of related work is the Triple Graph Grammar approach (see [25,16,11]), which is an alternative basis for defining model transformation semantics compared to the "simple" graph transformations we have used. Triple graph grammars have the advantage of offering a more fundamental solution to the problem of linking source and target graph, which we have had to solve by introducing an ad hoc graph encoding.

Acknowledgement. We wish to thank Klaas van den Berg, who co-supervised this work and without whose contribution it would never have attained its current form.

References

[1] Bardohl, R., H. Ehrig, J. de Lara and G. Taentzer, *Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation*, in: M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering (FASE)*, LNCS **2984** (2004), pp. 214–228.

[2] Burmester, S., H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals and A. Zündorf, *Tool integration at the meta-model level: The FuJaBA approach*, Int'l Journal on Software Tools for Technology Transfer **6** (2004), pp. 203–218.

[3] Csertán, G., G. Huszerl, I. Majzik, Z. Pap, A. Pataricza and D. Varró, *VIATRA — visual automated transformations for formal verification and validation of UML models*, in: *Int'l Conference on Automated Software Engineering (ASE)* (2002), pp. 267–270.

[4] Czarnecki, K. and S. Helsen, *Classification of model transformation approaches*, in: *OOPSLA 2003 Workshop on Generative Techniques in the Context of Model-Driven Architectures*, 2003.

[5] de Lara, J. and G. Taentzer, *Automated model transformation and its validation with Atom 3 and AGG*, in: A. Blackwell, K. Marriott and A. Shimojima, editors, *DIAGRAMS'2004*, LNAI **2900** (2004), pp. 182–198.

[6] DSTC, IBM and CBOP, *MOF Query/Views/Transformations, second revised submission*, OMG Document ad/2004-01-06, Object Management Group (2004).

[7] DSTC QVT Team, *Tefkat tutorial*, See www.dstc.edu.au.

[8] Ehrig, H., K. Ehrig, J. de Lara, G. Taentzer, D. Varró and S. Varró-Gyapay, *Termination criteria for model transformation*, in: M. Cerioli, editor, *Fundamental Approaches to Software Engineering (FASE)*, LNCS **3442** (2005), pp. 49–63.

[9] Ermel, C., M. Rudolf and G. Taentzer, *The AGG approach: Language and environment*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume II: Applications, Languages and Tools*, World Scientific, Singapore, 1999 pp. 551–604.

[10] Gardner, T., C. Griffin, J. Koehler and R. Hauser, *A review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard*, in: *First International Workshop on Metamodelling for MDA*, 2003.

[11] Grunske, L., L. Geiger and M. Lawley, *A graphical specification of model transformations with triple graph grammars*, in: A. Hartman and D. Kreische, editors, *Model Driven Acritecture — Foundations and Applications (ECMDA)*, LNCS **3748** (2005), pp. 284–298.

[12] Kalnins, A., J. Barzdins and E. Celms, *Model transformation language MOLA*, in: U. Aßmann, M. Aksit and A. Rensink, editors, *Model Driven Architecture*, LNCS **3599** (2005), pp. 62–76.

[13] Karsai, G. and G. Taentzer, editors, "International Workshop on Graph and Model Transformation (GRAMOT)," 2005, to appear.

[14] Kastenberg, H., *Toward attributed graphs in GROOVE (work in progress)*, in: R. Heckel, A. Rensink and B. König, editors, *Graph Tranformation for Verification and Concurrency*, ENTCS, 2005, to appear.

[15] Kleppe, A., J. Warmer and W. Bast, "MDA Explained, the Model Driven Architecture: Practise and Promise," Addison-Wesley, 2003.

[16] Koenigs, A. and A. Schürr, *Multi-domain integration with MOF and extended Triple Graph Grammars*, in: J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings (2005).

[17] Lambers, L., H. Ehrig and F. Orejas, *Efficient detection of conflicts in graph-based model transformation*, in: Karsai and Taentzer [13], to appear.

[18] Mens, T. and P. Van Gorp, *A taxonomy of model transformation*, in: Karsai and Taentzer [13], to appear.

[19] Nederpel, R., "A QVT model transformation language represented by graph production systems," Master's thesis, Department of Computer Science, University of Twente (2005), see http://www.cs.utwente.nl/~rensink/papers/nederpel2005.pdf.

[20] OMG, *Meta Object Facility (MOF) specification, version 1.4*, OMG Document formal/2002-04-03, Object Management Group (2002).

[21] OMG, *MOF 2.0 Query/View/Transformations RFP, version 1.1*, OMG Document ad/2002-04-10, Object Management Group (2002).

[22] OMG, *MDA guide, version 1.0.1*, OMG Document omg/2003-06-01, Object Management Group (2003).

[23] Rensink, A., *The GROOVE simulator: A tool for state space generation*, in: J. Pfalz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, LNCS **3062** (2004), pp. 479–485.

[24] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformation, Volume I: Foundations," World Scientific, Singapore, 1997.

[25] Schürr, A., *Specification of graph translators with triple graph grammars.*, in: E. W. Mayr, G. Schmidt and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, LNCS **903** (1995), pp. 151–163.

[26] Sprinkle, J., A. Agrawal, T. Levendovszky, F. Shi and G. Karsai, *Domain model translation using graph transformations*, in: *International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS)* (2003), pp. 159–168.

[27] Taentzer, G. and A. Rensink, *Ensuring structural constraints in graph-based models with type inheritance*, in: M. Cerioli, editor, *Fundamental Approaches to Software Engineering (FASE)*, LNCS **3442** (2005), pp. 64–79.

Transformational pattern system - some assembly required

Mika Siikarla [1] and Tarja Systä [2]

Institute of Software Systems
Tampere University of Technology
Tampere, Finland

Abstract

In the context of Model Driven Architecture (MDA), most model transformation mechanisms aim for rigorously and unambiguously defined, fully automatic transformations. We argue that such techniques, even when fully mature, are not applicable in all cases of software development. These difficult cases would benefit from flexible and semi-automatic open transformations. We present a mechanism, so called transformational pattern system, and show how it can combine human made decisions and intentionally vague and incomplete rules to perform a transformation.

Key words: graph transformation, open model transformation, pattern, mda, pattern system

1 Introduction

Model Driven Architecture [7] (MDA) is the most recent and most prominent attempt to raise the level of abstraction used in defining software. The level has previously been successfully raised from machine code, symbolic assembler, and primitive programming languages to modern high level programming languages and in some cases even generating code from models. Now the goal is to use models from earlier and earlier design and perhaps even requirements capture phases and derive implementation from them.

The benefits of achieving the MDA vision would of course be significant. Production efficiency would rise due to higher abstraction level. Maintainability would be improved when design models would always be up-to-date. Because rising the abstraction level has been so successful previously, some believe this next step will be just as successful, as soon as good enough methods

[1] Email: mika.siikarla@tut.fi
[2] Email: tarja.systa@tut.fi

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and tools have been developed. We argue that such expectations are reasonable only when certain restrictions apply. When the level of abstraction gets higher, automatic transformations get more complicated, their cost goes up, and they have to make decisions with greater consequences all leading to fewer cases where the transformation is usable.

Models from early design phases have less details than ones from later phases. They do not just show less details, they actually have less details. After all, an important reason for using high abstract level is to avoid committing to details too early. Details are added later, refining the model. Some of them are inconsequential, but some are important design decisions. The more abstract the model is, the bigger impact the decisions have on the end result. A guess can be made at source code level, knowing that at worst it will be off an opcode or two. A guess at the architecture level can go wrong a subsystem or two.

An automatic transformation can only succeed, if it knows what the design decisions should be. This is more likely in the context of, e.g. a single problem domain, company or product line or versions of a product, where the situation is well understood and rather stable. For example, C++ has standard fixed semantics, so a C++ compiler does not need to (must not!) make behaviour affecting decisions. If the context is not limited in any way, there are infinitely many possibilities, too many to take into account beforehand.

Automatic transformations do not get rid of complexity. Instead of relying on the expertise and wisdom of a designer to create a target model, we rely on the transformation engineer to create a transformation. The transformation must solve a more generic problem and apply to more cases than one, and is therefore more difficult to build. The relative development cost is reduced, if the transformation is applied to several products. For a one-of-a-kind product or for a small organization, it might not be cost-efficient to develop (and maintain!) another piece of software, i.e. the transformation itself.

We argue that in some cases where an automatic transformation is not feasible or even possible, some of the MDA benefits can still be achieved. Dropping the requirement for full automation and instead incorporating a human in the transformation process, by interacting with him and allowing manual changes, enables more flexible transformation mechanisms. In order for the human to be able to make a difficult design decision, he needs to understand its context. There is need for open transformation mechanisms, i.e. ones that are transparent, accessible, interactive, and flexible.

We present an experiemental semi-automatic transformation mechanism based on so called *transformational patterns*. This paper extends our previous work [10], where transformational patterns were used alone, by adding a method for joining several patterns together. The mechanism is fully transparent and allows the user to choose the order of tasks and make manual changes to the models. At this time, we do not attempt to tackle problems caused by incremental changes to the source model. We illustrate the use of

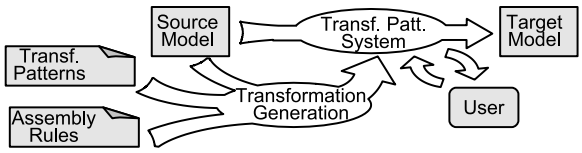


Fig. 1. Generating and applying a transformational pattern system

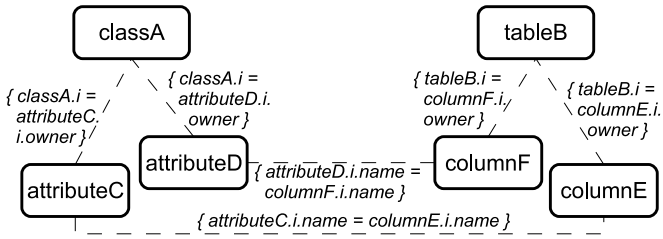


Fig. 2. A pattern with six roles and six constraints

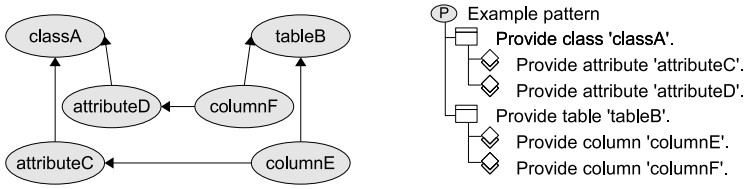
the mechanism with an example.

2 Transformation Mechanism

In this paper, a transformation specification consists of so called *(transformational) patterns* and *assembly rules*. A transformational pattern describes how a transformation rule, e.g. `Transform a UML Class into a Java class`, is implemented. The assembly rules describe how the individual patterns relate and which patterns are applied to which source model elements. We call such a collection of inter-related patterns a *(transformational) pattern system*. It is an implementation of an interactive transformation for a specific source model. These different components of a transformation are presented in Figure 1.

A transformational pattern describes a configuration of model elements, which must exist after the corresponding transformation rule has been applied. A pattern is given as a set of roles and constraints. Each role of a pattern instance is attached, i.e. *bound*, to a model element. The constraints restrict to which elements a role can be bound. A small pattern is depicted in Figure 2. The constraints state, for example, that (the attributes bound to) roles *attributeB* and *attributeC* belong to (the class bound to) role *classA*. They also require *columnE* to have the same name as *attributeC*. If the constraints permit, multiple roles can be bound to the same element.

Applying a pattern can also be viewed as a set of tasks; "bind *classA*", "bind *tableB*", etc. When all the tasks have been performed either by selecting an existing model element or by generating a new one (while observing the constraints), the pattern has been applied. To make performing tasks easier, each constraint concerning two or more roles is directed. That is, one role (binding) is considered to be "correct" and the other(s) must be bound to conforming element(s). This implies a partial ordering of tasks, which can be presented as a directed acyclic graph. Figure 3(a) depicts a task graph



(a) A task graph (b) A task hierarchy

Fig. 3. A task graph and the corresponding task hierarchy

for the pattern in Figure 2. A task graph resembles a function or a program that derives new bindings based on existing ones. For every task, a new role is bound to a model element. If there is only one option the task can be performed automatically, otherwise human interaction is required.

We use MADE [5] to apply patterns. For easier task selection, MADE presents a task graph as a hierarchy of roles/tasks. Figure 3(b) shows the task hierarchy for the task graph in Figure 3(a). The hierarchy criteria is currently fixed and is based on containment. For example, the task for *attributeC* is under *classA* because the constraints demand that the class bound to *classA* contains the attribute bound to *attributeC*. The user can browse this hierarchy by selecting a task. The tool will then show the list of tasks directly underneath the selected task. Tasks with unbound dependencies will be hidden. For example, the task for *columnE* will not appear before tasks for *tableB* and *attributeC* have been performed. MADE also offers some shorthand commands, for example to perform all automatic tasks in a task list.

Task graphs can be connected together in sequence and in parallel by merging some of their nodes. This is equivalent to merging the roles, where the new role has all the dependencies and constraints of the merged roles. Such a pattern system is a more complicated function, assembled from simpler ones, and fulfills a more complicated purpose. Since a pattern system is itself a pattern, MADE can be used to apply pattern systems, too.

The example in Figure 4 contains five task graphs (1.). A pattern system is assembled from the two top patterns by merging one node from each task graph. Likewise, the three patterns on the bottom are assembled into a second pattern system (2.). Two new pattern instances are created and joined with the old ones (3.) creating the task graph for the complete pattern system (4.).

The pattern assembly mechanism parses the source model and as a side-effect forms a pattern system by creating and joining pattern instances. The mechanism is essentially a graph rewrite system (GRS). However, each graph production p_i is associated with an action a_i. A production-action pair $\langle p_i, a_i \rangle$ is called an *assembly rule*. The productions are applied to a directed labeled graph representing the source model, where each node has a type and can have named values attached. Whenever a production is used, the associated action is triggered. The productions reduce the input graph step by step, while the actions construct the resulting pattern system. In other words, the GRS is

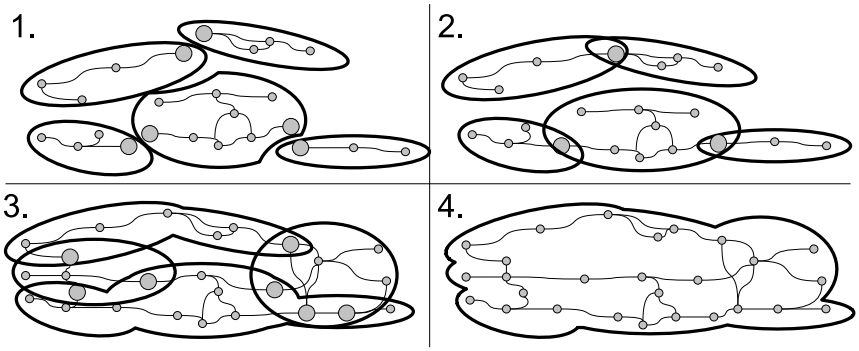


Fig. 4. An example of joining task graphs (patterns) together

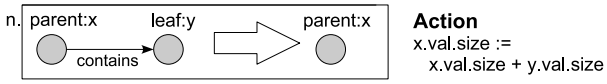


Fig. 5. An example of an assembly rule; a graph production and an action

used to recognize or parse the source graph.

The assembly rules are ordered and the first with an applicable production is always used. When no more productions apply, the mechanism stops regardless of how many nodes or edges remain in the graph.

In the beginning each node corresponds to one source model element and the node's name-values come from the element, e.g. the name and id of a UML class. Later on the values are usually roles or pattern instances created by actions. When a production triggers an action, it has access to the values of the nodes matching to the production's left hand side (LHS) and right hand side (RHS). A typical action fetches the patterns attached to two nodes in the LHS and joins them together. The concept is analoguous to the grammar rules (productions) and actions in the common textual parser generator yacc.

For a simple example, consider a graph consisting of directed trees, i.e. a directed forest, and that we want to know the amount of nodes in each of the trees. Let us assume that in the beginning the leaf nodes are of type $leaf$ and the other nodes are $parent$. Let us also assume that each node starts with a single named value; $size = 1$. The assembly rule in Figure 5 could be used as part of the solution. It is applicable whenever there are two nodes, x and y, such that x is of type $parent$, and y is of type $leaf$, and y is a child node of x. When the production is applied its action increments the value of $size$ in x by the value of $size$ in y. The leaf y is then removed from the graph. A few more assembly rules are needed to complete the example. One changes a $parent$ with no children into a $leaf$. Another collects the $size$ from a one-node tree into some global stack and removes the tree.

It is important to note that the assembly rules do not perform the actual model transformation. They only assemble the pattern system, which is then used to transform the model, guided by the user.

In the implementation, the productions are given using Object Constraint

Language [8] (OCL) expressions and Python code. For this reason, a graphical notation (Figure 5, 7) is used in this paper for presenting productions. The notation is used solely for visualization, and is not formally defined. The production rule implementation is currently not automatically derived from the description. Actions are expressed in Python.

3 Example of Constructing a Transformation

As an example, consider the seemingly simple transformation from a structure model (a UML Class Diagram) into a relational database schema. It seems quite straight forward, but there are details, options and exceptions that add complexity. For example, there are different ways to interpret and transform composition, inheritance and other relations between classes, and there is not always enough information in the source model to make the decision. It is in managing these details and variations where the real challenge for a transformation mechanism lies. With transformational patterns, their inherent flexibility and interactive nature helps overcome some of these difficulties.

A rough natural language description of the transformation might be:

(i) Each class inheritance hierarchy is transformed into a single table. The table is named after the root class.

(ii) At least one column in each table belongs to its primary key.

(iii) Foreign key should reflect the primary key selected for the target table.

(iv) Each attribute is transformed into a column in the table corresponding to the attribute's class. The column is named after the attribute.

(v) Each association is transformed into a table reference. The designer decides which table holds the foreign keys. The foreign keys are named after the primary keys and the association role chosen.

The (task graphs for) transformational patterns in Figure 6, one for each informal rule, describe how the rules are implemented. The constraints have been omitted for clarity. The patterns could be read as `"a table is created based on some class"` (pattern i), `"some columns are chosen from some table"` (pattern ii), and so on. When an instance of such pattern is partially bound, it gets a more precise meaning, e.g. `"a table is created based on class` *Show*`"`. The flexibility in patterns and the choices the user will make eventually decide how *exactly* the rule is applied.

The `+` in pattern (ii) and the `XOR` in pattern (v) are details of the notation for MADE, the tool used for applying patterns. The markings mean that the user decides at runtime how many *pk* roles pattern (ii) has and which of the alternative structures is used for pattern (v).

There are five assembly rules and their productions are in Figure 7. It is a coincidence, that there are as many rules as there are patterns. The first rule is used for initializing a *Class* node. Production (1) marks an initialized

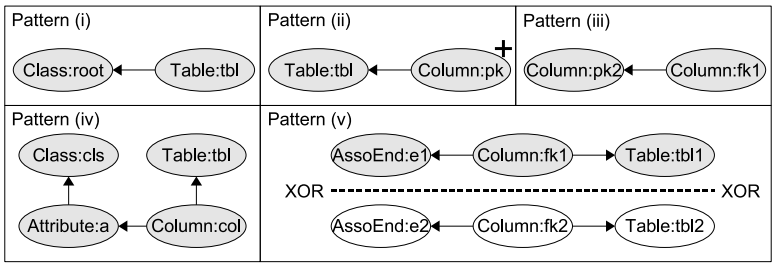


Fig. 6. Transformational patterns corresponding to the informal rules

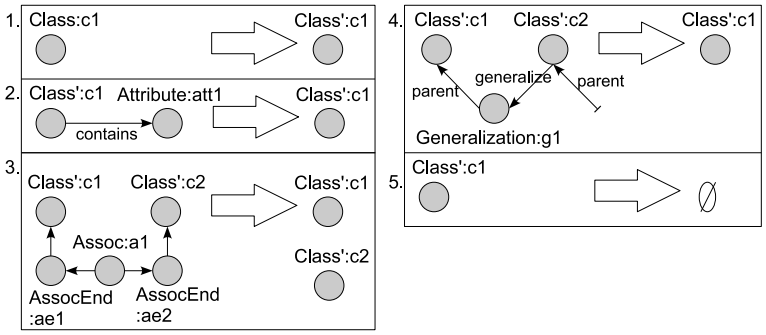


Fig. 7. GRS productions for the pattern assembly rules

node by changing its type to *Class′*. The rest of the productions parse the source graph. Productions (2) and (3) remove attributes and associations. Production (4) removes a leaf *Class′* in an inheritance hierarchy. When the hierarchy has been reduced to a single node, production (5) removes that node. The actions for productions (1) and (2) (in pseudo-code) are:

```
1 patt = new Pattern_ii #For the patterns, see Figure 6
  c1.val = {class:  new ClassRole, table:  patt.tbl, pk:  patt.pk}
  bind_role(c1.val.class, c1.id)
```

```
2 patt = new Pattern_iv
  merge_role(c1.val.class, patt.cls)
  merge_role(c1.val.table, patt.tbl)
  bind_role(patt.a, att1.id)
```

When the pattern assembly rules are used on, e.g. the diagram in Figure 9, the first production applies and is used. The action (1) is triggered and variable `c1` points to one of the graph nodes representing a class. The action attaches three roles as named values to the node; *class*, *table*, and *pk*. In addition, it binds the role *class* to the source model class the node corresponds to. The production changes the type of the node from *Class* to *Class′*, so that the first production will not be used on the node again. This is repeated on each node of type *Class*. So, the first assembly rule does not change the structure of the graph, it merely initializes the class nodes' values.

The second production is used when the first no longer applies. It finds attribute nodes and removes them. The action (2) creates a new instance

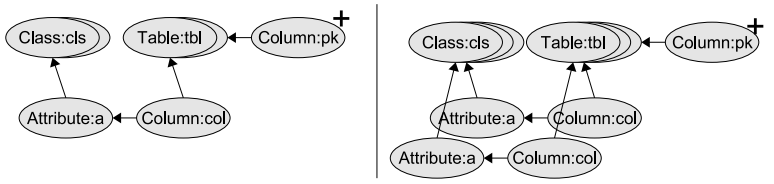


Fig. 8. Rule (iv) pattern joined once (left) and twice (right)

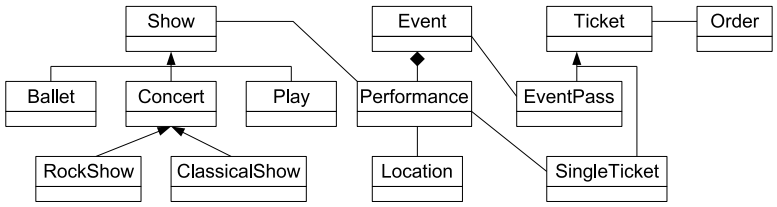


Fig. 9. Structure model used in the example

of pattern (iv), binds the attribute role to the source model attribute the attribute node corresponds to, and finally merges the pattern's *cls* and *tbl* roles with the roles *class* and *table* associated with the class node. The left side of Figure 8 shows the pattern associated with a class node after one of its attributes has been removed. The right side shows the pattern after another attribute has been removed. The stacked tasks represent merged tasks. In reality, it is not possible to tell after the fact, whether a task has been merged.

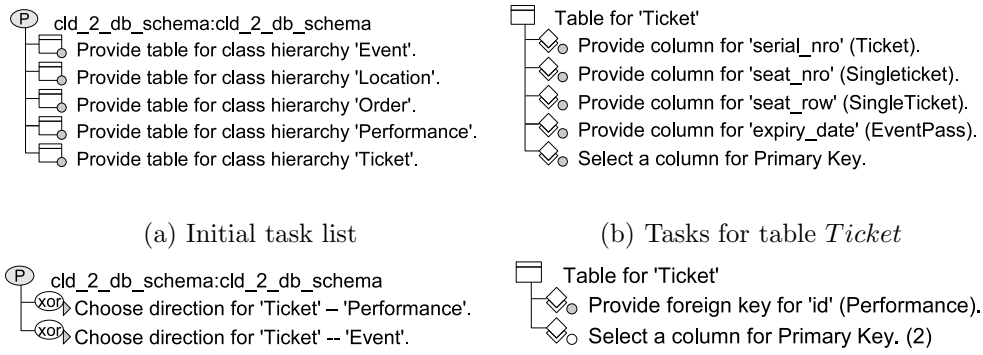
After the second production rule no longer applies, the third one is used, then the fourth, and so on, until no production rules apply. At that point, there is a complete transformational pattern system created by the actions.

4 Example of Applying a Transformation

To demonstrate applying a transformational pattern system, a possible user session is presented step by step. The transformation itself is the structure model to database schema presented in Section 3 and it will be applied to a ticket service structure model (Figure 9). Bob is assigned with the task of creating the database schema. A CASE-tool is used for visualizing the structure model and the schema (both as UML Class Diagrams) and MADE is used for applying the pattern system.

Bob starts the CASE-tool and loads the source model. He executes the assembly rules from the command line, starts MADE and imports the pattern system. A list of tasks appears, one `Provide table for class hierarchy <name>` task for each class hierarchy (Figure 10(a) [3]). Bob selects the task for *Performance* and tells MADE to generate a new table. A new class representing the table is generated and appears in the CASE-tool. New tasks become

[3] For better image scaling, bitmap screen captures in Figure 10(a)–10(d) have been manually redrawn in a vector format.



(a) Initial task list

(b) Tasks for table *Ticket*

(c) Choosing reference directions

(d) Deriving foreing keys

Fig. 10. Binding a pattern and the representation as a task graph

available and are listed under the old, now inactive, task; one for selecting primary keys and one for transforming attributes to columns. Bob ignores them for now, and instead instructs MADE to create tables *Show*, *Event*, and *Ticket*.

Bob looks at the tasks (Figure 10(b)) listed under the *Ticket* table; creating columns and selecting primary keys. He selects `Perform all automatic tasks` and a column (represented by an attribute) is created and appears in the CASE-tool for each attribute in the classes *Ticket*, *EventPass*, and *SingleTicket*. Primary keys are not selected, because that task is not automatic. Bob performs the task manually by selecting the column *serial_no* which he knows uniquely identifies a ticket. The selection is visualized by stereotyping the column as ≪`PrimaryKey`≫. There are no more mandatory tasks for this class, but he could select more primary keys if he wanted.

There is nothing special about the attributes of *Event* and *Performance*, so Bob tells MADE to generate columns for those, too. When looking for primary key candidates, he realizes none of the columns will work. He switches to the CASE-tool and manually adds a column *id* in both tables. Then, in MADE, he selects them as primary keys for their tables.

When primary keys have been selected for some tables, choosing the directions for table references between those tables is enabled. The task list now includes tasks for the associations from *Ticket* to *Event* and *Performance* (Figure 10(c)). Bob is able to reason that there will be many *SingleTickets* for each performance, and that one ticket can be used for one show only. He therefore selects the task for the association between *SingleTicket* and *Performance* and chooses *SingleTicket* to hold the foreign keys. When the choice is made, tasks for deriving the actual foreign key columns from the primary key of *Performance* appear (Figure 10(d)). Bob tells MADE to generate the foreign keys, and the column *performance_id* stereotyped ≪`ForeignKey`≫ appears in *Ticket*. Bob applies the same reasoning for the other table reference and generates the column *event_id* under *Ticket*, too.

The user can always choose the next tasks freely, as long as the tasks it

depends on have been performed first. Bob utilizes this freedom fully, when he, in this order, generates the tables for *Show* and *Order*, manually adds a column in *Show*, generates the derived columns for *Order*, selects one primary key column for *Show*, one for *Order*, then another for *show*, generates the table *Location*, and chooses a direction for the reference between *Show* and *Event*. This order may seem random from the outside, but Bob is working according to some personal internal logic, probably inexplicable even to himself. When he, not the tool, chooses what to do an in what order, he keeps better track of the context and is therefore more capable of making the right design decisions when the tool needs the human plug-in.

When looking at the relationship between *Location* and *Performance*, Bob concludes it is more complex than the previous ones. He decides there needs to be a third table to map the other two. There is no task for it, because such a possibility was not taken into account when designing the transformation. Still, Bob can manually create the mapping table and all required columns in the three tables. There is currently no way of marking a task obsolete, so he has to remember to ignore the task for choosing the direction for the *Location* - *Performance* reference. Although the purpose of the new table is not "understood" by the transformation, that does not affect the rest of the model and the rest of the transformation.

Bob started working on the transformation so late in the day, that he is not able to finish it before leaving work. So, he saves his work in the CASE-tool and MADE, knowing he can load the structure model, database schema and the transformation the next day and continue right where he left off.

5 Related work

There are many model transformation approaches, but few attempt interaction or manual editing of models beyond pre-determined choices or parameters.

Triple graph grammars [9] are grammars spanning three related graphs; one for the source model, one for the target model, and one for the relationships between the models. Each production alters all the graphs (models) at the same time, keeping them always synchronized and confirmant with their schemas (metamodels). A transformational pattern system contains elements for the source and target models and their relations. In that sense, a pattern system is an abstract triple graph. Due to the flexibility in binding, it represents a group of triple graphs.

With triple graph grammars, additions to the source or target models can be dealt with simply by applying further productions. We have not yet addressed the problem of incrementality for pattern systems. Triple graph grammars are also bidirectional. Although a transformational pattern itself is not directed, a derived task graph always is. The assembly rules, too, create a bias towards a direction.

Some graph transformation tools provide interaction, e.g. AGG [11], and

AToM^3 [4]. The user can perform stepwise transformations and to choose the next production to apply. In AGG the user can even choose on which graph elements the production is applied, which resembles binding a pattern. Allowing the user to choose productions is powerful and enables ambiguous rules. But in order to make a decision, the user has to thoroughly understand the grammar in addition to understanding the transformation semantics, e.g. classes to tables, attributes to columns. We try to put the decisions more in terms of the semantics by placing the interaction in the pattern system. The user still has to work with a tool's process, but we believe it to be more similar to the user's view of the transformation process. Perhaps the power of interactive grammars can somehow be combined with the intuitiveness of pattern systems.

GREaT [2] is a graph transformation tool, which produces a Java program that can be run to perform the model transformation. We use assembly rules to produce a pattern system, which is then applied with MADE. However, the motivation with GREaT seems to be integration into Java applications and possibly efficiency. User interaction does not seem to be considered.

A transformational pattern system, once all the roles are bound, is also a mapping between the source and target models. So, model mapping techniques [6] are in some way similar. However, they are typically bidirectional, whereas transformational patterns are not.

ATL [1], among others, approaches the problem of too strict transformation definitions by enabling specialization of transformations. This, in effect, allows vague or general rules, which are then refined for a more specific situation.

6 Conclusions and future work

Transformational patterns (and thus also pattern systems) are rather flexible in describing structures. They can be viewed as task graphs, which are executable and give an implementation for applying the patterns. Tasks also have a natural interpretation as user choices, making task graphs interactive. Adding assembly rules gives the approach some of the benefits of the fully automatic approaches without removing the built-in user interaction.

Although incrementality was not considered in this work, it is very important for open transformations. As it is now, any significant change to the source model demands a reassembly of the pattern system, effectively forgetting the previous user decisions. Supporting incremental transformations needs to be researched. The pattern assembly mechanism also has to be better integrated with the pattern tool, to improve the user experience. For the same reason, the production rules need a well-defined and intuitive notation.

We also intend to strengthen the theoretical foundation of our approach with, e.g. graph grammars. For example, it has been pointed out to us that transformation pattern systems might bear resemblance to graph processes [3]. This is an interesting connection we intend to explore further.

References

[1] Bézivin, J. and F. Jouault, *Using ATL for checking models*, in: *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT)* (2003).

[2] Christoph, A., *Graph rewrite systems for software design transformations*, in: *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World* (2003), pp. 76–86.

[3] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamenta Informaticae **26** (1996), pp. 241–265.

[4] de Lara, J. and H. Vangheluwe, *AToM3: A tool for multi-formalism and meta-modelling*, in: *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering* (2002), pp. 174–188.

[5] Hammouda, I., J. Koskinen, M. Pussinen, M. Katara and T. Mikkonen, *Adaptable concern-based framework specialization in UML*, in: *Proceedings of ASE 2004* (2004), pp. 78–87.

[6] Hausmann, J. H. and S. Kent, *Visualizing model mappings in UML*, in: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization* (2003), pp. 169–178.

[7] OMG, *Model driven architecture (MDA)* (2001).
URL http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf

[8] OMG, *UML 2 OCL available specification* (2005).
URL http://www.omg.org/cgi-bin/doc?ptc/2005-06-06

[9] Schürr, A., *Specification of graph translators with triple graph grammars.*, in: E. W. Mayr, G. Schmidt and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, Lecture Notes in Computer Science **903** (1995), pp. 151–163.

[10] Siikarla, M., K. Koskimies and T. Systä, *Open MDA using transformational patterns*, in: U. Aßmann, M. Aksit and A. Rensink, editors, *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, Lecture Notes in Computer Science **3599** (2005), pp. 108–122.

[11] Taentzer, G., *AGG: A graph transformation environment for modeling and validation of software.*, in: J. L. Pfaltz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, Lecture Notes in Computer Science **3062** (2004), pp. 446–453.

Towards Testing the Implementation of Graph Transformations

Andrea Darabos [1] and András Pataricza [2] and Dániel Varró [3]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary

Abstract

We present a method for testing the implementation of graph transformation specifications focusing on test case generation for graph pattern matching. We propose an extensible fault model for the implementation of transformations based on common programmer faults and the technicalities of graph transformations. We integrate traditional hardware testing (combinational circuits) and software testing techniques (mutant generation) for generating test cases.

Key words: graph transformation, testing, test generation, pattern matching

1 Introduction

Due to the growing importance of transformations, a standardized Model Driven Architecture (MDA) based model transformation method has been requested in the *OMG Request for Proposal MOF 2.0 Query / View / Transformations* [17]. Graph transformation, which provides a rule and pattern-based manipulation of graphs, is a promising technology for model transformations as evaluated by a taxonomy presented in [15].

The separation of the design and execution time of model transformations is a recent tendency today (see GreaT, Fujaba, Viatra), by providing both an interpreted engine and compiled transformation plug-ins as platform specific implementations (Figure 1). In case of graph transformations, the implementation can be derived by hand or generated automatically as described in [3].

[1] E-mail:andrea_darabos@t-online.hu
[2] E-mail:pataric@mit.bme.hu
[3] E-mail:varro@mit.bme.hu

[4] This work was partially supported by the SENSORIA European project (IST-3-016004). The third author was also supported by the Bolyai Scholarship.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

However, even if these plug-ins are generated automatically, these implementations can be erroneous.

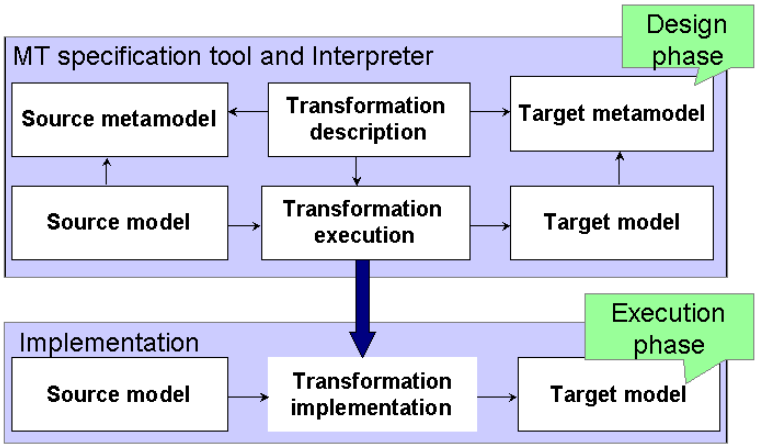


Fig. 1. The meaning of model transformation (MT) implementation

In order to detect conceptual flaws in transformations, typically, either verification (termination, confluence, semantic correctness, etc.) and/or testing techniques are applied. In general, verification is mainly used in the design phase of transformations, while testing is appropriate in the implementation phase, when a stand-alone transformation plug-in has been created for the corresponding specification. Testing has typically two main advantages: *(i)* it can be used for large models without combinatorial explosion, *(ii)* tests are executed directly on the implementation, which in case of model checking often cannot be guaranteed.

Our aim is to test stand-alone graph transformation implementations by generating test cases from graph transformation specifications. In this paper, we focus on the graph pattern matching phase, which is considered to be the most problematic phase of graph transformations.

We propose a fault model to incorporate potential flaws in the implementation. Test generation is performed by using a combinational circuit representation derived from the preconditions of graph transformation (further: GT) rules. Possible faults are mapped to stuck-at-faults (a signal lines is assumed to be stuck at a fixed logic value, regardless of the inputs), as there are various hardware testing methods for the combinational circuit and this fault model. With the help of systematic fault injection, single binary (stuck-at-faults) faults are inserted into the circuit and test vectors are calculated. The exact test cases are generated by mutation rules in the form of test graphs.

2 Graph transformations in Modeling Languages

2.1 Metamodels and models

The abstract syntax of a modeling language is defined by a metamodel (MM). It can be represented formally as a type graph. The instance model or instance graph (M) is a well-formed instance of the metamodel and describes concrete systems defined in the modeling language. The finite automaton will serve as a running example throughout the paper. To demonstrate our steps, the very simple domain of the finite automaton and belonging instance models are depicted in Figure 2.

Example 2.1 According to the metamodel, a well-formed instance of a finite automaton is composed of *states* and *transitions*. A transition is leading between its *from* state and *to* state. The initial states of the automaton are marked with *init*, the active states are marked with *current* edges. Special, e.g. colored states, are definable by inheritance.

A sample automaton `a1` consisting of three states (`s1`, `s2`, `s3`) and three transitions between them t1 (leading between `s1` and `s2`), `t2` and `t3` is depicted as an instance model. We can notice that the initial state of `a1` is `s1`.

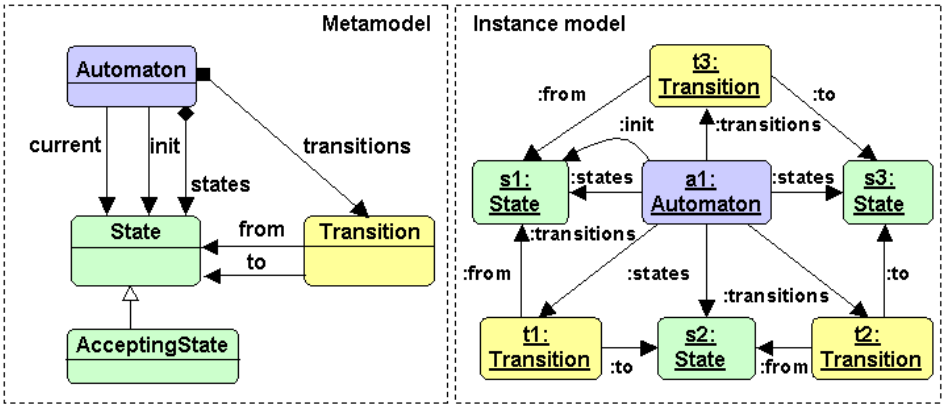


Fig. 2. Metamodel and instance model of finite automata

2.2 Graph transformations

Graph transformation [20] is a pattern and rule based formalism for the manipulation of graph models. On rule application, a graph is transformed by replacing a part of it by another graph. With the definition of a metamodel and a set of rules over that metamodel the dynamic changes of an initial model can be described. On rule application, a graph is transformed by replacing a part of it by another graph.

A graph transformation rule R contains a left-hand side graph LHS, a right-hand side graph RHS, and negative application condition graphs NACs. The LHS and the NAC graphs are together called as the precondition of the rule R.

The application of a rule to a instance model M (which is instance model of the metamodel) replaces a matching of the LHS in M by an image of the RHS (formally there is a graph morphism between the LHS and the instance model M). This is performed by *(i)* finding a matching of LHS in M, *(ii)* checking the negative application conditions NACs (which prohibit the presence of certain objects and links) *(iii)* removing a part of the instance model (that can be mapped to LHS but not to RHS) yielding the context model, and *(iv)* gluing the context model with an image of the RHS together by adding new objects and links (that can be mapped to the RHS but not to the LHS) and obtaining the derived model M'. A graph transformation is a sequence of rule applications from an initial model M_i.

Typically, the most critical phase of a graph transformation step is graph pattern matching, i.e. to find a single (or all) occurrence(s) of a given graph in a instance model M.

Example 2.2 The dynamic semantics of finite automatons can be described with the help of graph transformation rules. The example rule depicted in Figure 3 shows the firing of a transition. If the `S1` state of the `A1` automaton is active (there exists a `C1` edge between them), and there exists a transition, which leads from `S1` to `S2`, then the rule is applicable, and the current state of the automaton will be `S2`.

The process of pattern matching can also be illustrated. If we regard the instance model in Figure 2 as an instance model, and we assume, that there is an additional `current` edge from `a1` to `s1` in it, then the example GT rule can be applied onto this instance graph: with variable instantiation `A1-a1`, `S1-s1`, `T1-t1`, `S1-s2`, `C1-c1`, `St1-st1`, `St2-st2`, etc. The rule can be applied here, and as a result, the `current` edge will be leading from `a1` to `s2` in the instance graph.

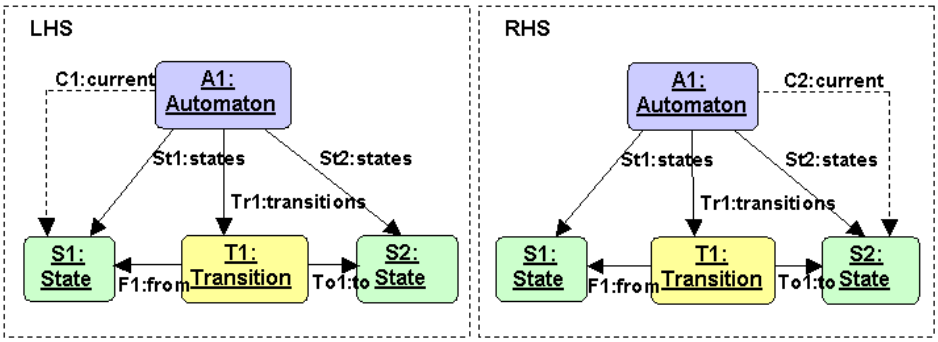


Fig. 3. The fire GT rule of a finite automaton

3 Fault Model

For the testing of graph transformation implementations, (in fact, for any kind of testing), a formal fault model has to be defined for the possible fault

types in the implementations. This was inspired by object-oriented testing and hardware-based testing techniques, assuming similarities between traditional software developers and transformation developers and these were adopted for graph transformations. Thus the following fault types were declared for the pattern matching phase (the fault model is extensible, additional faults can be defined via the same method):
General implementation faults (based on programmer's experience):

- *Omission fault.* In a graph transformation implementation, the omission fault means, that some elements are missing from the implementation of the pattern matching criteria, described originally in the specification. This can lead to situations, when graph transformation rules can be matched to a smaller subset of graph elements than it was specified (e.g. node `s1` or `states` edge is missing from the implementation of pattern matching).
- *Interchange fault.* An interchange fault means, that the criteria was implemented with incorrect type definitions (e.g. too specific type used, `AcceptingState` instead of `State`). Most commonly, the programmer makes such a fault in the generalization hierarchy.
- *Side effect fault.* This fault means unnecessary, redundant elements in the implementation, having more criteria defined for pattern matching than those specified (e.g. additional nodes or edges were implemented in the criteria).

GT specific faults of the pattern matching phase:

- *Dangling edge production fault.* The production of dangling edges is not allowed in the *DPO Double-Pushout* approach [5], therefore this criterion must be investigated.
- *Violation of injectivity fault.* If only injective matchings are allowed, the non-injective matching of elements (different nodes in a GT rule have the same image in the match) is a violation of injectivity fault.

  In the future we also plan to consider non-injective matchings, where the violation of identification condition needs to be investigated.

The pattern matching criteria for each rule are defined by the LHS of rules in the specification. It is assumed, that the graph pattern is well-typed (syntactically correct), therefore only implementation (semantic) errors are aimed to be detected.

4 Test Case Generation for Graph Pattern Matching

4.1 High level Overview

Figure 4 provides a brief overview of our test case generation approach. There are three primary components of the envisaged framework: test case constructor, testing engine and test analyzer. For the current paper, we only focus

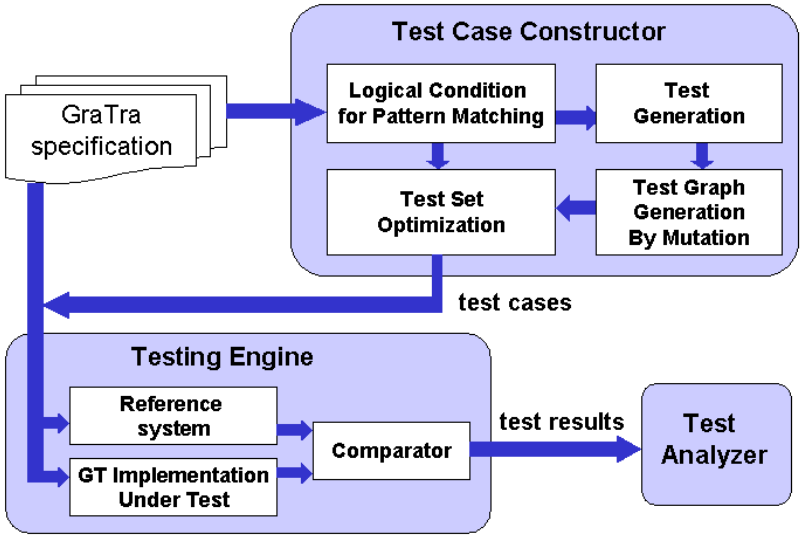


Fig. 4. Testing workflow

on pattern matching, but the conceptual elements of the framework are more general, like the test case constructor and the testing engine. The input of the test case constructor is a set of GT rules as specification, and the output is a set of test cases in the form of test graphs for the testing of the implementation of the transformation.

The main steps of the test case generation are the following:

- *Pattern Matching Criteria* The logical criteria for the successful matching of each rule is extracted from the transformation specification in the form of a Boolean expression. The formula is satisfied, when a successful matching is found for the belonging GT rule. The idea is to reuse existing techniques for hardware testing, therefore the Boolean formula is depicted in form of a combinational circuit, for which traditional test generation algorithms can be applied.
- *Test Generation* With systematic fault injection, single faults are injected into the inputs of the circuit. For its simplicity, the method of Boolean differences [16,21] is applied here which generates binary test vectors for stuck-at-faults in the combinational circuit representing the pattern matching. The method of Boolean differences guarantees that with the generated test vectors the fault is observable on the output of the circuit. If a variable in the generated test vector is one, then the corresponding condition is satisfied, else it is not satisfied. For further details see Section 4.2.
- *Test Graph Generation* After the test vectors are calculated with binary values, the corresponding test graphs have to be produced. The LHS copy of the tested GT rule is created, and with mutation rules the specified faults are injected into the LHS copy graph. The calculated test vectors control the process of mutation rule application. The resulting test graphs are the possible realizations of the calculated logical test vectors, created according

to the fault model introduced in Section 3. For instance, a binary test vector expressing that some node has a wrong type can have multiple realizations, e.g. a more general type can be one case (e.g. `AutomatonElement` instead of `State`), or a more specific type (`AcceptingState` instead of `State`) in the generalization hierarchy can be implemented. Thus, for each test vector, multiple test graphs can be created. For further details see Section 4.3.

- *Test Set Optimization* The set of produced test graphs should be examined for test optimization, in order to create a more compact set of test cases, it should be optimized. Naturally, it has to be decided, whether the aim is only fault detection or diagnosis as well. In the latter case, test compaction can only be carefully applied, not to loose information for diagnosis.

After the test set is created, it is passed to the *Testing Engine*. The testing engine is responsible for executing the transformation specification on the given test graph both in the reference system (a GT Interpreter tool) and the transformation implementation. The comparator compares the results of pattern matching of both components, and collects the results for each test graph. Here an important restriction has to be made: only those GT rules are suitable for testing, for which the difference of RHS and LHS is nonzero. It means, that in order to being able to compare the results of pattern matching, rule application must make visible changes on the test graph. The test analyzer collects and visualizes the results of the test engine.

Due to space restrictions, we discuss in further details only the test generation and test graph generation phases. The interested reader can find more about this topic in [6].

4.2 Details of representation and test generation

The general, formal criteria for a match are presented in [23]. The idea is to describe these criteria for each GT rule in the rule set of the specification, and to create a combinational circuit representation of this Boolean formula, which will supply us with the usability of traditional testing methods. The formula evaluates to 1, if a match fulfills the defined criteria meaning that the pattern matching was successful.

The construction of the formula follows the upcoming scheme:
Existence of images of the LHS elements in the instance graph $\wedge$
$\wedge$ Correct type of elements in the match $\wedge$
$\wedge$ Attribute conditions satisfied by the matched attributes $\wedge$
$\wedge$ Isomorphism/homomorphism condition $\wedge$
$\wedge$ Fulfillment of dangling edge conditions $\wedge$
$\wedge$ No violation of NACs

For the example rule showed in Figure 3, the criteria is the following:
Automaton(a1) $\wedge$ State(s1) $\wedge$ Transition(t1) $\wedge$ State(s2) $\wedge$
$\wedge$ current(c1, a1, s1) $\wedge$ states(st1, a1, s1) $\wedge$ states(st2, a1, s2) $\wedge$

$\wedge$ transitions(tr1, a1, t1) $\wedge$ from(f1, t1, s1) $\wedge$ to(to1, t1, s2) $\wedge$
$\wedge$ s1 $\neq$ s2

For the presented example, no dangling edge or NAC condition was present, therefore we briefly summarize the construction of these criteria.

- *The dangling edge condition*: All nodes and edges in LHS of a GT rule R but not in RHS are deleted when the rule is applied. When applying this rule R on a instance graph, all the edges *to* and *from* these nodes which are not part of the match are the dangling edges. The dangling edge condition is fulfilled, if no dangling edges will be produced on rule application.
- *The NAC condition*: If only single, non-hierarchical NAC graphs are used, the NAC condition is satisfied, if its elements cannot be found in the match. The Boolean formula can be written for the NAC graph as above, and it is inverted before connecting it to the pattern matching criteria. In case of hierarchical NAC conditions, the Boolean formula of a lower level NAC is inverted before connecting it to the higher level condition. More on this topic can be found in [19].
- *The injectivity condition* is formalized as follows: $\forall$ X, Y nodes $\in$ LHS, and p, q images of X and Y in the instance graph: X $\neq$ Y $\Rightarrow$ p $\neq$ q which means, that two different elements of the LHS of a given GT rule cannot be mapped to the same element in the match. In our example, this was the `s1` $\neq$ `s2` condition for states `S1` and `S2`.

The combinational circuit generated from the criteria is depicted in Figure 5.

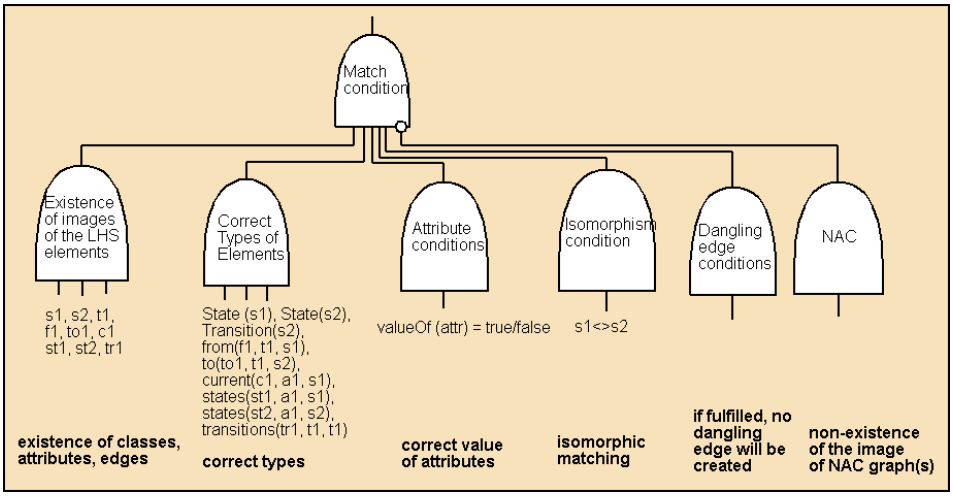


Fig. 5. Combinational circuit representation with details of the example

The test vector generation is performed on the combinational circuit with systematic fault injection, with the help of the method of Boole differences resulting in binary test vectors for each GT rule. In our example, a test vector for the omission fault of the `Automaton(a1)` element from the criteria is the following: (0,1,1,1,1,1,1,1,1,1,1) for (a1,s1,t1,s2,c1,st1,st2,tr1,f1,to1,s1$\neq$s2).

Some details of the processing of these test vectors and the test graph generation are discussed in the next section.

4.3 Details of test graph generation

The generation of test graphs is based on a set of mutation GT rules, all of which define the injection of faults of fault types defined in the fault model (Section 3). These rules describe the possible realizations of a given fault type: e.g. in case of the interchange fault type (where we supposed faults inside the generalization hierarchy), the fault can originate from a too specific or a too general type realized by the implementation. The mutation rules are metatransformation rules, which are applied on the LHS copy of a GT rule from the specification as instance graph.

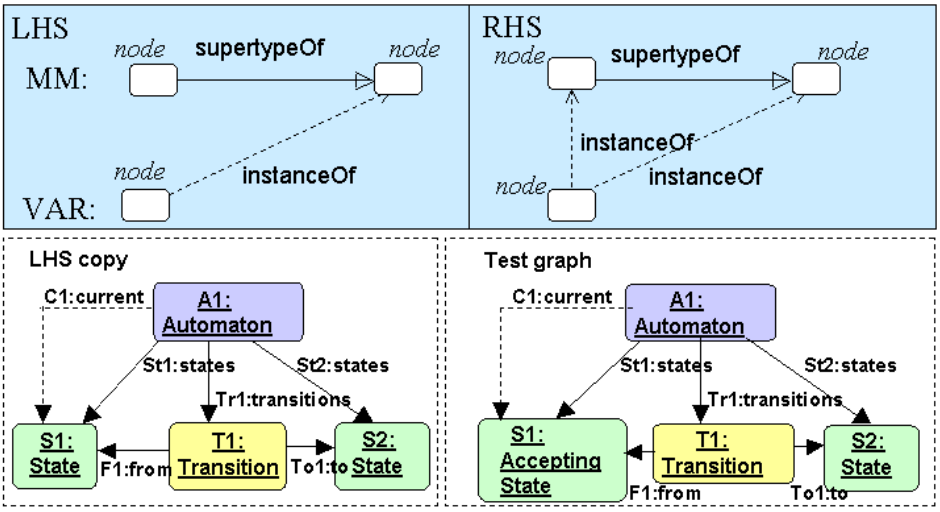


Fig. 6. Mutation rule for one realization of the interchange fault and the generated test graph from the LHS copy of the example

Returning to our example of the finite automaton, after applying the mutation rule depicted in Figure 6 on the `State` entity of the GT rule LHS copy (Fig. 3), we would gain an `AcceptingState` entity, which is of a more specific type. A test graph including this fault would test, whether the implementation regards the correct type of element when pattern matching or not. The original GT rule (Fig. 3) is applied onto this test graph, and the success or failure of pattern matching indicates the correctness of the implementation.

The test vectors calculated on the combinational circuit can be regarded as a control structure for the mutation rules; they define, which test graphs have to be produced with the help of mutation rules. The instance graph is the LHS copy of the GT rule under test, and the result graph is the test graph. For each test vector, a new LHS copy is created, and the according to the possible mutation GT rules, as many test graphs are created as the number of different possible mutation rules were defined for this fault type. Thus, a *test graph set* is generated for each test vector.

5 Related Work

The formal correctness analysis of model transformations has been already investigated in the literature.

Syntactic correctness and completeness was examined in [9] and sufficient conditions guaranteeing the termination and uniqueness of transformations were set up in [12] based on the critical pair analysis [10] technique.

An automated formal verification technique is presented in [18,7,24] based on various model-checking techniques to prove semantic correctness criteria in graph transformation systems starting from a concrete initial graph. A static analysis technique is proposed in [1] to investigate the correctness of graph transformation systems by using a Petri net abstraction. A tool for checking inductive invariants has been presented recently in [4]. To guarantee the preservation of constraints during model transformations, aspect-oriented techniques are proposed in [13].

However, much less results are available for the testing of graph transformations. Jeff Gray underlined the importance of testing model transformations and presented a model transformation testing framework in [14]. This framework focuses on automating test execution, i.e. to automatically compare test results with the expected behavior, while we focus on automatic test generation.

The testing of code generators specified by graph transformation rules has been addressed in the literature by adapting well-known test strategies such as test case generation by model checking [2] or the classification tree method [22]. In [11] tests are generated for black-box implementations of web services based upon domain partitioning. While the overall goal i.e. to derive test cases directly from GT rules is similar, we assume that implementation is strongly linked to the GT specification, furthermore we use systematic fault injection and combinational circuit testing techniques in the background.

On the tool level, one pioneer is FUJABA which generates JUnit test cases [8] from graph transformations specified by graphical story diagrams. This approach focuses on the correctness of model manipulation steps (based on the right-hand side), which nicely complements the results of our current paper.

6 Conclusion and Future Work

In this paper we presented a method for the test generation for the pattern matching of graph transformation implementations, and a test execution method as well. Our primary goal was to use well-known hardware/software testing techniques and the extendibility of the fault model, therefore we elaborated a method which can be used for any graph transformation implementation and extended on demand with e.g. more fault types or with the injection of multiple faults.

The complexity issues of this problem will be part of measurements of the implementation, but it can be said that the pattern matching criteria - from which the test generation is performed - is comparable, proportional with the LHS size of the GT rules of the specification. Therefore, as the size of GT rules is generally much smaller than the size of instance models, it seems to be surmountable. Another important question is the size of test graphs, which, in the presented version is equal to the size of GT rule LHS graphs, as test graphs are generated with the slight modification of corresponding GT rule LHS graphs.

Our aim is to extend our testing method with the consideration of rule application, the RHS or postcondition of GT rules as well. Secondly, we plan to improve the fault model with control flow faults and design methods for testing the control structure of graph transformations as well. Furthermore, more work has to be done in the area of test set optimization. It is a future goal to examine the usability of our method for not only fault detection, but also for diagnosis as well, and to try out our method on graph transformation implementations.

References

[1] Baldan, P., A. Corradini and B. Koenig, *A static analysis technique for graph transformation systems*, LNCS, **2154** (2001), pp. 381–395.

[2] Baldan, P., B. König and I. Stürmer, *Generating test cases for code generators by unfolding graph transformation systems*, in: *ICGT'04*, LNCS, **3256**, (2004), pp. 194-209.

[3] Balogh, A., D. Varró, A. Pataricza and G. Varró, *Generation of platform-specific transformation plugins for EJB 3.0*, to appear for SAC'06.

[4] Becker, B., H. Giese and D. Schilling, *A plugin for checking inductive invariants when modeling with class diagrams and story patterns*, in: *Proc. of the 3rd International Fujaba Days*, 2005, pp. 1–4.

[5] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Loewe, "in [20] Basic Concepts and Double Pushout Approach," World Scientific, 1997 pp. 163-245.

[6] Darabos, A., A. Pataricza, D. Varró, *Testing the implementation of graph transformations, Technical Report*, in: http://www.inf.mit.bme.hu/~varro/publication/TR-2006-01.pdf.

[7] Dotti, F. L., L. Foss, L. Ribeiro and O. M. dos Santos *Verification of Distributed Object-Based Systems*, in: *FMOODS*, 2003, pp. 261-275.

[8] Geiger, L., C. Schneider and C. Reckord, *Template- and modelbased code generation for MDA-Tools*, in: *Proc. of the 3rd International Fujaba Days*, 2005.

[9] Hausmann, J. H., R. Heckel and S. Sauer, *Extended model relations with graphical consistency conditions.*, in: *UML Workshop on Consistency Problems in UML-based Software Development*, 2002, pp. 61-74.

[10] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, in: *Proc. 1st Intl Conf. Graph Transformation*, LNCS, **2505** (2002), pp. 161-176.

[11] Heckel, R., L. Mariani, *Automatic Conformance Testing of Web Services*, in: *Proc. Fundamental Approaches to Software Engineering* (2005).

[12] Küster, J. M., R. Heckel and G. Engels, *Defining and validating transformations of UML models*, in: *Proc. IEEE Symposium on Human Centric Computing Languages and Environments (HCC)*, 2003, pp. 145–152.

[13] Lengyel, L., H. C., T. Levendovszky, *Eliminating crosscutting constraints from visual model transformation steps*, in *7th International Workshop on Aspect-Oriented Modeling*, 2005.

[14] Lin, Y., J. Zhang and J. Gray, *Model comparison: A key challenge for transformation testing and version control in model driven software development*, in: *OOPSLA*, 2004.

[15] Mens, T. and P. V. Gorp, *A taxonomy of model transformation and its application to graph transformation*, in *GraMoT*, 2005.

[16] Miron, A., M. A. Breuer and A. D. Friedman, "Digital systems testing and testable design," Computer Sci. Pr., 1990.

[17] OMG, *MOF 2.0 query / views / transformations rfp ad/2002-04-10* (2002), URL: http://www.omg.org/cgi-bin/apps/doc?ad/02-04-10.pdf.

[18] Rensink, A., *The GROOVE simulator: A tool for state space generation*, In M. Nagl and J. Pfalz, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, LNCS, **3062** (2004), pp. 479-485.

[19] Rensink, A., *Representing First-Order Logic Using Graphs*, In *Graph Transformations: Second International Conference (ICGT)*, LNCS, **3256** (2004), pp. 319.

[20] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformations: Volume 1: Foundations," World Scientific, 1997.

[21] Sallay, B., A. Petri, K. Tilly and A. Pataricza, *High level test pattern generation for VHDL circuits*, in: *IEEE European Test Workshop*, 1996, pp. 202-206.

[22] Stürmer, I., M. Conrad, *Test suite design for code generation tools*, in: *ASE'03*, (2003), pp. 286.

[23] Varró, D., "Automated Model Transformations for the Analysis of IT Systems," Phd thesis, Budapest University of Technology and Economics (2003).

[24] Varró, D. and A. Pataricza, *Generic and meta-transformations for model transformation engineering*, in: *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, LNCS **3273** (2004), pp. 290-304.

Maintaining coherence between models with distributed rules: from theory to Eclipse

Paolo Bottoni [a] Francesco Parisi-Presicce [a,c] Simone Pulcini [a]
Gabriele Taentzer [b]

[a] *Università di Roma "La Sapienza - Italy*
[b] *Technische Universität Berlin - Germany*
[c] *George Mason University - USA*

Abstract

Integrated Development Environments supporting software and model evolution have to deal with the problem of maintaining coherence between code and model despite changes which may occur on both sides. Rather than going through model reingeneering or code regeneration, it would be better to build a full correspondence between the starting models and keep it updated in an incremental way after each evolutionary step. In a series of previous papers, it was shown how distributed graph rewriting could support such updates. Here, we show how to construct a distributed graph from individual models, through the use of synchronized rules. In particular, we discuss the case of Java code and UML models, and propose an Eclipse implementation of the approach.

Key words: Distributed graphs, model morphism, software evolution.

1 Introduction

Integrated Development Environments (IDEs) are increasingly devoted to enable their users to move through the different processes of design and implementation, providing tools to keep some form of coherence between the design models and the produced code. In particular, several tools support refactoring, usually providing the possibility of combining simple refactorings

⋆ Work supported in part by the European Community's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis and by the European Network of Excellence Interop

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

into complex ones, managing aspects such as assessment of preconditions and modifications of model components, typically class diagrams.

In previous papers we have made the case for keeping into account other views of the design model, such as sequence and state diagrams, and have proposed the use of distributed graph rewriting [3,4] for an integrated management of modifications in the code and in the global UML model underlying a software artifact. The approach is based on identifying mappings between software elements, represented by an Abstract Syntax Tree (AST) derivable from the code, and model elements, expressed in UML terms. Both AST and UML models are seen as instances of their respective metamodels, interpreted as graph types. In this context, the construction of the correspondence between them amounts to that of their (typed) interface graph. In such a graph, each node corresponds to some abstract concept common to the two models. At the instance level, morphisms between nodes in the interface graph and the corresponding nodes are constructed.

In this paper, we show how to construct the interface graph and the associated morphisms, based on the assumption that the two models (AST and UML) already exist and are coherent in the sense that elements with the same (qualified) name refer to the same concept. The approach can be easily extended to the case of two incoherent models, so that reasons for failure can be identified. On the other hand, by assuming one of the two models as correct, repair actions can performed on the other one.

In particular, we express the sequences of actions performing the morphism construction as transformation units [11,2], which are specializations of a general transformation pattern and illustrate how such specializations can be generated. We also present guidelines for implementing the rules using the Eclipse API system [6]. The discussion is illustrated by presenting transformation units for the construction of mappings between some particular types.

The rest of the paper develops as follows. After a brief recall of Theory in Section 2, we present the general pattern of transformation some of its specific instantiations in Section 3. Section 4 presents the Eclipse implementation and conclusions are given in Section 5.

2 Theory and models overview

For correspondence construction, we rely on the DPO approach [5], and in particular, to the theory of *distributed graphs and graph transformation* [13], allowing the concurrent construction of the interface graph, of the morphisms between it and individual graphs, and of morphisms between corresponding nodes in the different graphs, so that diagrams such as the one of Figure 1 commute. Figure 1 also illustrates the convention adopted in the rest of the paper: corresponding nodes are identified by the same name, primed in the code graph and doubly primed in the UML graph. This allows us to deal with the existence of morphisms and of a node with corresponding name in the

interface graph implicitly.

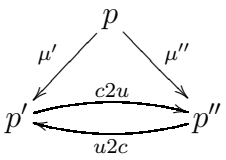


Fig. 1. The general form of morphisms.

In the examples of the paper, we show pairs of local rules working in a synchronized manner. Rules are defined on the metamodels specifying the type graphs for the two models.

A *rule* $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$ is given by two morphisms l and r. Given an object G and a rule $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$, a *match* of p to G is a morphism $m : L \rightarrow G$. A *direct derivation* d from G to H by p and match m, $d : G \Rightarrow_{p,m} H$, is given by a double pushout (see Figure 2). Rules may have application conditions, both positive and negative (NACs), as well as attribute evaluation actions associated. In Figure 2, the NAC is an object N and an injective total morphism n; a rule is applicable only if match m cannot be extended to m' such that $n \circ m' = m$. Several objects N_i, and the associated morphisms n_i, can be associated with one L, indicating that no extension of m should exist for any i. The *derived rule* from a direct derivation $d : G \Rightarrow_{p,m} H$ is $p_d : G \stackrel{g}{\leftarrow} D \stackrel{h}{\rightarrow} H$.

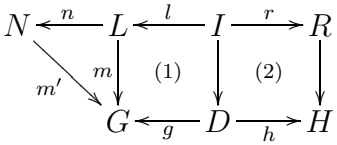


Fig. 2. Double Pushout rule with a negative application condition.

A transformation unit controls rule application through control conditions specified by expressions over a set $Names$ of rule names. The class $\mathcal{C}$ of *control expressions* is recursively defined by

(i) $Names \subseteq \mathcal{C}$,

(ii) **forall** n **end** $\in \mathcal{C}$, if $n \in Names$,

(iii) $C_1; C_2 \in \mathcal{C}$, if $C_1, C_2 \in \mathcal{C}$,

(iv) **asLongAsPossible** C **end** $\in \mathcal{C}$, if $C \in \mathcal{C}$,

(v) **if** B **then** C **end** $\in \mathcal{C}$, if $C \in \mathcal{C}$,

where B is a logical expression constructed using the logical operators OR and AND on atoms of the form `applicable(`r`)`, with r a named rule and `applicable` a predicate which evaluates to true only if r is applicable in the current graph. If an expression consists of a name $r \in Names$ only, the rule with name r is applied to the current host graph. The operator in (ii) applies the rule with name n at all different matches in parallel to the same host graph. The operator ; is left associative and applies first the expression C_1 and then

the expression C_2. The operator in (iv) sequentially applies expression C as long as its application is possible. The operator in (v) prescribes the execution of the expression C conditioned on the success of B (typically this will contain names of rules to be applied first). Transformation units have a transactional interpretation, i.e. they either succeed or fail completely.

In this paper we exploit the metamodels resulting from the definition of the abstract syntax of the Java language, as per the JavaML DTD [1], and the UML Metamodel [9].

3 Correspondence Construction

In this section, we illustrate the approach to the construction of the correspondence, by showing a general template for the used transformation units and illustrating it by an example. The complete construction is described in [12]. While the identification of corresponding elements is based on type and name identities, the main problem lies in the identification of the context, i.e. the *namespace*, in which to check identities. A general search template has therefore been specifically devised to address this problem.

In general, we consider the Java AST as the basis for the construction process, so as to exploit the facilities for tree visit provided by Eclipse. For the sake of simplicity, a slightly abstract form of Java and UML model elements are used in the rules. Where necessary, adaptations of the rules to the real metamodels are discussed.

Templates for Correspondence Construction

In several situations, establishing a correspondence between elements requires recognizing the correspondence of the embedding contexts. In particular, we rely on the notion of parenthood as provided by the tree model. As the number of sibling elements is arbitrary, we adopt transformation units to force an exhaustive search of such elements.

In particular, we observe that a common structure exists for transformation units to build correspondences between elements in a well defined pattern. A correspondence can be established between elements so that the element p' in the AST is the root of some subtree, and children of p' correspond to elements which are linked according to some suitable association with p''.

We can therefore define a template for transformation units to be properly instantiated with a suitable set of rules to resolve the correspondence for a specific pattern. The transformation unit is constructed from 4 basic steps.

Step 1 : Identify the corresponding parent elements to ensure the presence of a context for the rest of the transformation unit.

Step 2 : The construction of the correspondence for children of a mapped element requires a mapping for each corresponding pair of children. Hence, the rule establishing the correspondence has to be applied in the context of

the parent and to each different pair of corresponding children.

The template for transformation units is expressed as

CorrespondenceConstruction()
forall `mapParent();`
forall `mapChild()` **end**

This template can be compared to amalgamated graph transformation as presented in [13].

Sample Correspondence Construction

Now we illustrate the specialization of the template presented above to study the case in which the context is a Java class declaration; as stated in JavaML DTD, together with zero or more field declarations in its scope. The construction of the mapping between a Java *class* and a UML *Class* is realized by the rule `mapClass()` in Figure 3, an example of instantiation of `mapParent()`, while the construction of mappings between Java *field*s and UML *Attribute*s requires the instantiations of `mapChild()` in the form of `mapField2Attribute()`, as shown in Figures 4.

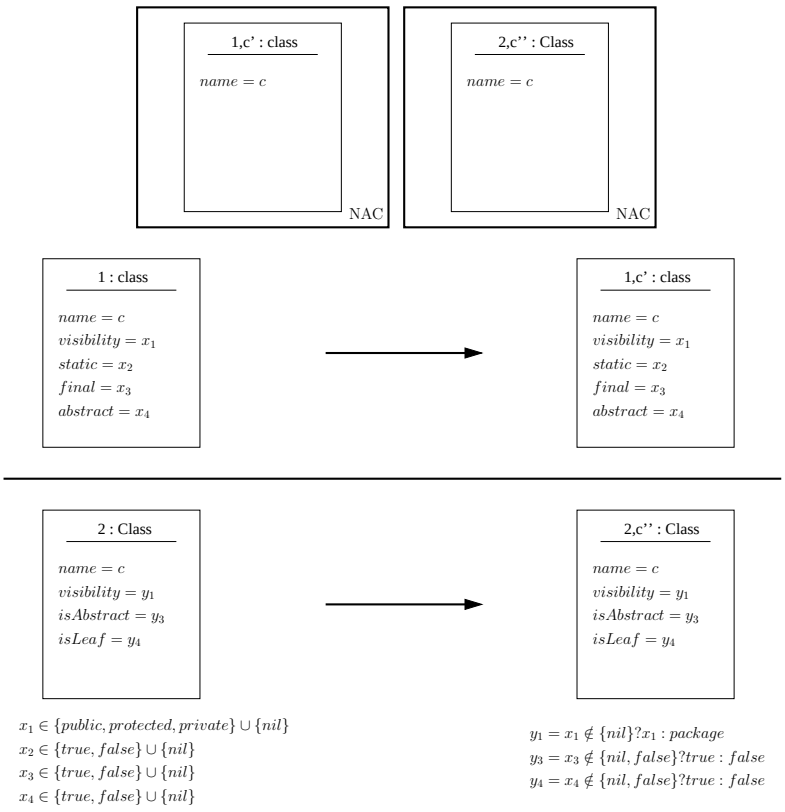


Fig. 3. Rule `mapClass()`.

The rules in Figure 3 show several application conditions on the class properties:

- If visibility in the AST is undefined, then the UML side assumes default, *package*, visibility. Otherwise, *visibility* is the same for both elements;
- The UML counterparts of the `abstract` and `final` JavaML attributes are `isAbstract` and `isLeaf` respectively;
- No counterpart for the JavaML *static* attribute is available from the UML metamodel for outer classes.

In the rules of Figure 4 specific issues of concern are [1]:

- `targetScope` is specified with the `instance` value according to the metamodel semantic. By doing so, `Attribute` is not used to store meta-information but behaves as a normal model attribute;
- `changeability` represents the UML 1.5 way to specify a Java `final` attribute modifier.

The transformation unit which establishes the correspondence between classes, fields, and attributes results from the specialization of the template given above and is expressed as follows:

Field2Attribute()
forall `mapClass());`
forall `mapField2Attribute()` **end**

4 Correspondence Construction between Java and UML in Eclipse

This section discusses the implementation of template instances in an Eclipse plugin, `com.spulci.C2MCM` (**Code to Model Consistency Maintainer**). C2MCM is based on the Eclipse AST framework, residing in the `org.eclipse.jdt.core.dom` package tree, and on the UML2 Eclipse tool project in package `org.eclipse.uml2` [9]. C2MCM manipulates structures generated by these APIs to search for semantic equivalent nodes inside them. C2MCM also creates a representation of the interface graph within an XML file. A brief introduction to the Eclipse platform, *AST* framework and the *UML2* plugin is given as needed.

4.1 The Eclipse Platform

Eclipse is a platform centric IDE which offers tools to develop and maintain software taking into account various project aspects. The whole Eclipse architecture is extensible and open. Indeed, tools belonging to the platform are structured as plug-ins. Each plug-in can define one or more **extension-points**, places where another plug-in can attach itself to provide new capabilities and offer an interface to the existing ones.

[1] The field mapping rules shown in this article are a simplified version; some attributes are omitted and a more complex pattern on the UML side is not shown in order to keep the presentation of the Eclipse implementation simpler.

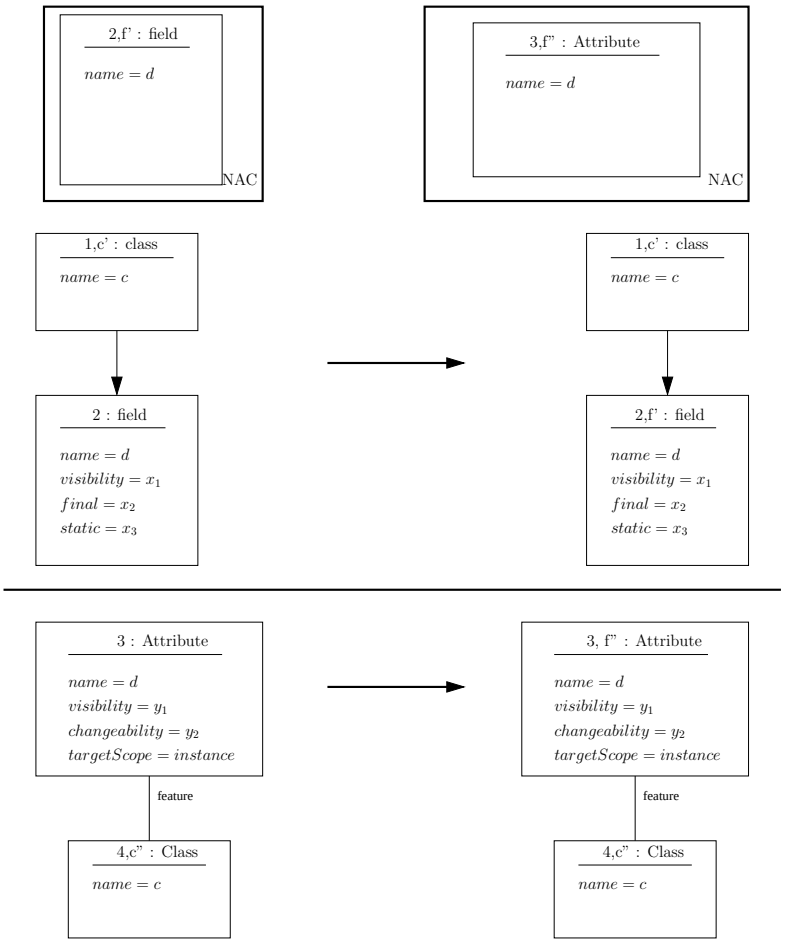


Fig. 4. Rule `mapField2Attribute()`.

4.2 Java Abstract Syntax and UML2 in Eclipse

We rely here on the definitions of the Java Abstract Syntax and of UML2 as provided by the Eclipse core, in which the instances of these metamodels are stored as separate files without reference between them. The basic assumption is that matching names refer to corresponding elements.

Classes from `org.eclipse.jdt.core.dom` and `org.eclipse.uml2` are imported to manage the Java *AST* and *UML2* models. The AST of some Java file is taken as input, allowing the search for semantically equivalent nodes in the UML2 model during the AST visit.

Correspondence construction in C2MCM is started by a call to the method *startEngine(ICompilationUnit icu)*, where the actual value for *icu* is an instance implementing the `ICompilationUnit` interface, specified by the user

through the plug-in GUI. This is the root of an AST built from a `.java` file. Besides loading the AST, this method evaluates the URI of the UML2 model on which to construct the mapping and passes it to the *loadModel(URI uri)* method which actually loads it.

The realization of the approach takes advantage of the implementation of the `Visitor` pattern supported by Eclipse which can be advantageously used to implement the template as developed in the previous section.

Actually, visiting the tree according to the node types allows the interleaving of rules from different transformation units. However, this does not alter the final result with respect to the normal execution of these transitions. Indeed, each transformation unit resulting from the instantiation of the template produces, as its net effect, the construction of a node in the interface graph and of the mappings to UML2 and AST models, without eliminating any existing node or edge. As a result, no derived rule for each such instantiation may disrupt the positive context for the application of another (i.e. to consume something in the left-hand side of a rule). Hence, building a correspondence between some elements cannot prevent the construction of other correspondences between elements in their context. We can thus conclude that any interleaving of rules from different transformation units produces the same result, provided that any partial order between rules in the same transformation unit is respected.

loadModel() returns a **Package** model class instance with the same name as the **Package** Java class. To avoid namespace conflicts, we adopt the convention of always using the fully qualified name **org.eclipse.uml2.Package**. The model is loaded through a call to an EMF method, as the UML2 plug-in is an extension of the *Eclipse Modelling Framework*.

4.3 Code Skeleton

The first step to the Eclipse implementation of a transformation unit is to identify the nodes that should be visited in the AST. The visit is started on the nodes for which a transformation unit is defined. This results in the mappings prescribed by instantiations of `mapChild`, and possibly in those prescribed in the instantiations of `mapParent`, which are optionally applied. According to the AST Eclipse API, it is necessary to override the appropriate *visit()* method for each node type that has to be visited by the framework [2] The steps below analyze the template core notions and show the skeleton followed to build the Eclipse implementation:

Context Identification and Applicability: The identification of the context (schematised in the template as parent) for the node under examination is done by navigating the tree starting from the current node and looking for

[2] The abstract syntax node type is passed as parameter to *visit()* Hence, a *visit(A x)* method codes a visit for a node **x** of Java type **A**. To grant children visit for the current node, the value *true* must be returned by each implementation.

the pattern described in the `mapParent()` rule, also checking the applicability conditions. In most cases, this is simply done by navigating upwards until a node of a specific type is found. As node visits proceed from the root downwards, a mapping for the found parent may have been constructed in the visit of some other type with the same context (e.g. fields and methods in a class).

Node Mapping: The visiting policy adopted in the Java AST Framework provides an implementation of the `forall mapChild() end` construct, invoking a visit each time it finds a node of a certain type. This assures that a node of a certain type is visited at most once for each visit. Actually, it proceeds in a sequence in which the leftmost child of a node is always the first to be visited, and the subsequent siblings are visited in the order of declaration.

Name checking: As the mapping relies on name identification, the method `getFullyQualifiedName()` is used on AST nodes. On the UML side, the obtained name is used to construct an argument for *findNamedElements()*, which returns a *Collection* of nodes (typically at most two elements, if a variable and a method in the same class have the same name). The node of the correct kind is then extracted from the collection.

Application Conditions: An application condition in a rule is directly coded as a Boolean clause which performs checks on the attribute values specified in the rule.

Mapping construction: If the check is passed, the mapping is represented by adding an XML node to three different documents, one representing the Interface Graph, one for the *Java to UML* correspondences, the last for the reverse UML to Java mappings.

4.4 AstDecorator class: AST visit to find equivalent nodes

The bulk of the work is realized within the `AstDecorator` class in Listing 1, by which AST nodes are visited to find semantic equivalences. The constructor initializes a reference to the UML2 model passed as argument and stores the UML2 model name, to be used to construct fully qualified UML2 names. For each AST node type a version of the *visit()* method is defined. The actual node parameter is passed at runtime by the framework while the returned boolean value is set to true to allow visits to children nodes. In particular, for each node of the AST, a reference to the corresponding element in the UML model is set, and vice versa. Moreover, a node of the interface graph is constructed with references to the nodes put in correspondence. This also provides the correct context for the visit to the children.

In particular, we show the code for a `TypeDeclaration` node in the Java Language Specification [3] in Figure 3, and for a `VariableDeclarationFragment`,

[3] We follow here JLS3, i.e. the version described in the third edition of [10]

a JLS grammar element containing JavaML Field node items, together with their parent `FieldDeclaration`(see Figure 4).

A **TypeDeclaration** can be specialized as either an Interface or a Class Declaration; we consider here only the latter. The corresponding `Class` element in the UML2 model is found using the Eclipse *findNamedElement()* method. Inside the **if** clause body, the concrete coding of the mapping is performed. (See Listing 2.)

Field declarations require some additional work; a field identifier can be found inside a **VariableDeclarationFragment**, child of a **FieldDeclaration**. As our matching technique is based on name searching, it is better to define a visit on the former instead of the later. As explained before, a check is needed to find the context of that node. This time the context will be a class declaration and is searched by the method in Listing 3.

The visit implementation is shown in listing 4. Its structure is quite similar to the **TypeDeclaration** visit, exploiting the Java context to find an UML Class that contains a semantic equivalent field.

4.5 XML Document for the Interface Graph

Correspondences built by **C2MCM** are maintained both as new elements of the XML files for AST and UML2 and in a specific XML Document representing the Interface Graph. Nodes in this document have the following structure:

- The name of the node is the name of the rule which built it.
- The attribute *JAVANAME* contains the fully qualified name of the Java Ast element mapped by the rule.
- The attribute *UMLNAME* contains the fully qualified name of the corresponding element in the loaded UML2 model

As an example, the following code snippet constructs the node for the `mapClass()` rule mapping, using the DOM4J open source API [7]:

```
Element igChild = igRoot.addElement("Class2Class"); //Node name
igChild.addAttribute("JAVANAME", packageName+"."+
   className.getFullyQualifiedName()); //Java name
igChild.addAttribute("UMLNAME",md.getQualifiedName()+"::"+
   className.getIdentifier()) //UML name
```

5 Conclusion

In conclusion, we have shown how synchronized rules defined on the meta levels of Java abstract syntax and UML2 can be used to establish correspondences between instance models. This can be used for several purposes, including navigation from code to model and viceversa, and is particularly suited to

allow consistency management between refactored code and model, without having to recur to reverse engineering or recompilation.

References

[1] Badros, G., *Javaml: A markup language for java source code*, 9th Int. World Wide Web Conference (2000).
URL http://www.badros.com/greg/JavaML/

[2] Bottoni, P., M. Koch, F. Parisi Presicce and G. Taentzer, *Automatic consistency checking and visualization of OCL constraints*, in: *UML 2000 - The Unified Modeling Language* (2000), pp. 294–308.

[3] Bottoni, P., F. Parisi Presicce and G.Taentzer, *Specifying Integrated Refactoring with Distributed Graph Transformation*, in: *Applications of Graph Transformations with Industrial Relevance*, LNCS **3062** (2004), pp. 220–235.

[4] Bottoni, P., F. Parisi Presicce and G. Taentzer, *Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations*, in: P. v. Bommel, editor, *Transformation of Knowledge, Information, and Data: Theory and Applications* (2004), pp. 95–125.
URL http://tfs.cs.tu-berlin.de/%7Egabi/gBPT04.pdf

[5] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*, , **I**, World Scientific, 1997 pp. 163–246.

[6] D'Anjou, J., S. Fairbrother, D. Kehn, J. Kellerman and P. McCarthy, "The Java Developer's Guide to Eclipse 2nd Edition," Addison Wesley, 2004.

[7] Dom4J Group, *Dom4J API Project*, http://www.dom4j.org/.

[8] Eclipse Organisation, *Eclipse 3.1.x Official Documentation*, http://help.eclipse.org/help31/index.jsp.

[9] Eclipse Organisation, *UML2 project*, http://www.eclipse.org/uml2/.

[10] Gosling, J., B. Joy, G. Steele and G. Bracha, "Java™Language Specification, Third Edition," The Java™series, Addison Wesley, 2005, 3rd edition.

[11] Kreowski, H.-J. and S. Kuske, *Graph transformation units with interleaving semantics*, Formal Aspects of Computing **11** (1999), pp. 690–723.

[12] Pulcini, S., "Evoluzione concorrente di Modelli Basata su Grafi Distribuiti," Master's thesis, University "La Sapienza" of Rome, Italy (2005).

[13] Taentzer, G., "Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems," Ph.D. thesis, TU Berlin (1996), Shaker Verlag.

A Listings

```
public class AstDecorator extends ASTVisitor {
  public Model md; public String modelName;; //UML2 Model and Model name
  public AstDecorator(Model md){super(); this.md = md; this.modelName = md.getName();}
  ...
  public boolean visit(TypeDeclaration node){ // see Listing 2
   return true;
  }
  public boolean visit(VariableDeclarationFragment node){ // see Listing 4
   return true;
  }
}
```

Listing 1: AstDecorator class

```
public boolean visit(TypeDeclaration node){
  //check for class declaration
  if (!node.isInterface()){ SimpleName className = node.getName(); //get node simple name
   Collection c = UML2Util.findNamedElements((Resource) md.eResource(),
       modelName+"::"+packageName+"::"+className.getFullyQualifiedName());
   Iterator it = c.iterator(); //iterate on found model elements if any
   while(it.hasNext()){
     if (it instanceof org.eclipse.uml2.Class){
       org.eclipse.uml2.Class cl = (org.eclipse.uml2.Class)it.next(); //cast to Class
       VisibilityKind visibilityKind = cl.getVisibility(); //get UML visibility
       int modifiers = node.getModifiers(); //get AST node modifiers bit mask
       Modifier.ModifierKeyword keyword = Modifier.ModifierKeyword.fromFlagValue(modifiers);
  //boolean clause for application conditions
       boolean test = (cl.isAbstract() & Modifier.isAbstract(modifiers)) |
             (cl.isLeaf() & Modifier.isFinal(modifiers)) |
             (keyword.toString().contains(visibilityKind.getName()));
       if (test){ // code for mapping construction}
     }
   }
  }
}
```

Listing 2: visit(TypeDeclaration node) body

```
private TypeDeclaration getClassDeclaration(ASTNode node){
  ASTNode tempNode = node.getParent();
  while(!(tempNode instanceof TypeDeclaration)){ tempNode = tempNode.getParent(); }
  return (TypeDeclaration) tempNode;
}
```

Listing 3: getClassDeclaration(ASTNode node) body

```
public boolean visit(VariableDeclarationFragment node){
 TypeDeclaration parent = getClassDeclaration(node); //get the ClassDeclaration context
 if (!parent.isInterface()) { //check for parent node to be a class
   String parentName = parent.getName().getFullyQualifiedName();
 //search for the class inside the UML model
   org.eclipse.uml2.Classifier classifier;
   Collection c = UML2Util.findNamedElements(md.eResource(),
     md.getName()+"::"+packageName+"::"+parentName);
   Iterator it = c.iterator();
   while(it.hasNext()){
     classifier = (org.eclipse.uml2.Classifier)it.next();
 //find UML Attribute with same Java Field name
     Property attribute = classifier.getAttribute(node.getName().getFullyQualifiedName());
 //find fragment modifiers
     FieldDeclaration parentNode = (FieldDeclaration) node.getParent();
     int modifiers = parentNode.getModifiers();
     Modifier.ModifierKeyword keyword = Modifier.ModifierKeyword.fromFlagValue(modifiers);
     if (attribute != null){
       VisibilityKind attributeVisibility = attribute.getVisibility();
 //boolean clause for the application conditions
     boolean test = (keyword.toString().contains(attributeVisibility.getName()) |
       (attribute.isReadOnly() & Modifier.isFinal(modifiers)) |
       (attribute.isLeaf() & Modifier.isStatic(modifiers)));
     if(test){ // code for mapping construction }
     }
   }
 }
 return true;
}
```

Listing 4: visit(VariableDeclarationFragment node) body

Visual Specification of Metrics for Domain Specific Visual Languages

Esther Guerra, Paloma Díaz [1,2]

Computer Science Department
Universidad Carlos III
Madrid, Spain

Juan de Lara [3]

Polytechnic School
Universidad Autónoma
Madrid, Spain

Abstract

We present a Domain Specific Visual Language (DSVL) for the definition of metrics for other DSVLs. The metrics language has been defined using meta-modelling, and includes some of the more used types of product metrics. The goal is to make the definition of metrics for a DSVL easy, reducing or eliminating the necessity of coding. For this purpose, we rely on the use of visual patterns for the specification of the properties that should be measured in each metric type.

These ideas have been implemented in the AToM^3 tool, which allows the definition of DSVLs by means of meta-modelling. In this way, with the new extension, the DSVL designer is able to define a metrics suite for a DSVL. Then, an environment is generated where a number of widgets allow taking actual measures of the defined metrics on the models. We present some illustrative examples using the hypermedia design language Labyrinth.

Key words: Domain Specific Visual Languages, Metrics, Meta-Modelling, Graph Patterns, Code Generation.

1 Introduction

Diagramatic notations are pervasive in many software development activities. They are used in the planning, analysis and design phases as a means to

[1] Email:eguerra@inf.uc3m.es
[2] Email:pdp@inf.uc3m.es
[3] Email:jdelara@uam.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

specify, understand and reason about the system to be built. DSVLs are constrained diagramatic notations, oriented to a particular application domain. They provide high-level, powerful primitives, having the potential to increase the user productivity for the specific modelling task.

Measurement plays a central role in many engineering disciplines, such as electrical, mechanical and civil engineering [6]. However, traditionally, it has received little attention in the area of Software Engineering. The kind of entities that can be measured in this area include processes, resources and products [6]. In this paper we concentrate in the latter. Product metrics measure features of software systems (e.g. complexity, cohesion, coupling or maintainability) in order to control and improve their quality. One of the factors that may improve the use of metrics in industrial practice is their support by tools. Moreover, integrating a metrics tool in the early phases of the development can help to detect defects prior to implementation, saving time and budget. However, there is a proliferation of notations and tools that the software engineers use, and adapting and implementing metrics for them is a costly and time-consuming activity. Our goal is to provide a means to reduce such cost, by making the customization of metrics for any kind of DSVL easy (not only in the Software Engineering domain).

In this work, we propose using a DSVL (called *Metrics*) for the specification of a metrics suite for other DSVLs. *Metrics* has been defined through a meta-model which contains the main types of metrics we have identified. These include metrics for global model properties (such as number of cycles and size), single element features (e.g. methods of a class in object oriented languages), features of groups of elements (e.g. their similarity or coupling) and paths (e.g. hierarchies in object oriented languages, navigation paths in web design languages). The DSVL designer is able to customize these metrics by providing a visual pattern with the property to be measured. Visual patterns are graphical, declarative, user-friendly and intuitive. They have the advantage of saving the user the necessity of coding the metrics procedure and learning neither the API of the used tool nor the programming language in which the tool was coded. In addition, since no coding is required, it can help to minimize errors and reduce the development time, mainly when dealing with complex metrics. Nonetheless, the *Metrics* language also allows creating new metrics (different from the ones we provide) in a procedural way by coding in Python. In addition, it is possible to specify threshold values for the metrics. Thresholds may have an associated action, described either in Python or using a graph transformation system [13]. In this way, when a metric reaches one of its threshold values, the user is asked whether he wants to execute the action. This is useful if the action executes known design patterns or redesigns that improve the quality of the final product. For space constraints we concentrate only in the metrics aspect of the DSVL and leave out the discussion on actions.

These ideas have been newly implemented in the AToM^3 tool, and are

illustrated with examples using the Labyrinth DSVL for hypermedia design [4]. The tool also gives support for the execution of metrics and report generation, without the necessity of coding.

2 A Taxonomy of Product Metrics

In this section we present a classification of the main types of product metrics. We have defined such a taxonomy by generalising metrics that are devoted to a specific language, notation or domain (like [2][6][11][15][16]) so that we provide an abstract metric language that is domain independent. We have distinguished four types of metrics:

- *Model-Oriented* metrics allow taking measures of the whole model. These include for example Mc Cabe's cyclomatic number (number of cycles), which is used in software engineering to measure the code complexity of a module [11].
- *Element-Oriented* metrics measure properties of individual elements in the model. In object oriented systems, these include the number of methods of a class.
- *Group-Oriented* metrics measure features of groups of elements in a system. For example, in object oriented notations they can provide an idea of the modularity (cohesion) of a system, by measuring the similarity between the different attributes and methods of classes [15].
- *Path-Oriented* metrics gather measurements involving paths between elements of the same type (or any of their subtypes). Thus, a path is made of instances of a certain type connected through some relation (which in fact can be a "complex" relation, made of several connected elements). For example, in web applications, a navigation path joins different pages by means of hyperlinks. Inheritance-related metrics can also be included here. In this group we find metrics for measuring the length of the path between two elements, or to detect start points of paths.

3 The *Metrics* Domain Specific Visual Language

We have created a DSVL for metrics specification using meta-modelling. The goal of the language is to be able to adapt to particular DSVLs some general predefined metrics (or create new ones) in an easy way. Fig. 1 shows its meta-model.

Abstract class *Metric* is the base class for all the metrics that the DSVL designer may create. It has a *name*, which must be unique. Attribute *variableTypes* indicates the domain of the metric (i.e. the types for which the metric is going to be calculated). For example, if the attribute contains the name of two types, the metric is calculated for each combination of one instance of the first type and another one of the second, resulting in a matrix.

If no type is specified, the domain of the metric is the whole model, and a scalar is obtained as a result. Attribute *subtypeMatching* specifies whether the objects in the domain must have exactly the type specified in the previous list (value *false*) or also its subtypes are allowed (value *true*). In addition, relation *dependency* allows a metric to use results calculated by others. A constraint in the meta-model forbids cycles of *dependency* relations.

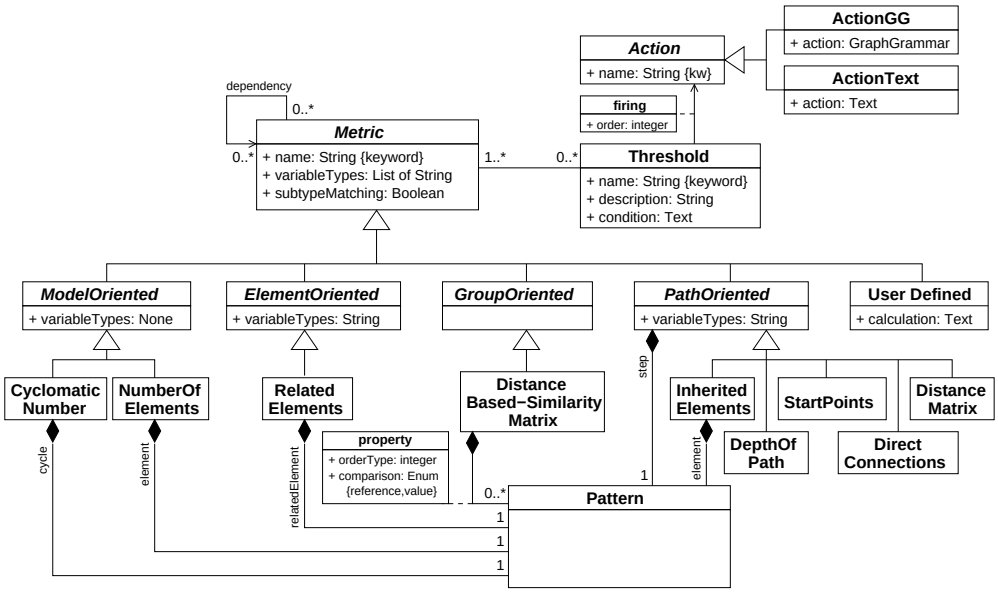


Fig. 1. Meta-Model for Metrics.

Thresholds can be associated to metrics, and contain a *name*, a *description* and a *condition*. The latter is a logical expression over metric values. Thresholds may have a number of associated actions that can be fired whenever the metric makes the threshold condition true. *Actions* can be described by means of procedural code (in Phyton), or by means of a graph transformation system. For space limitations, we leave out the discussion on actions and concentrate on metrics.

The four categories in our taxonomy of metrics are considered in the meta-model, all of them inherit from class *Metric*. Class *ModelOriented* and its children implement metrics of the first kind. The domain for the metric is not a single type of element, but the model itself. That is the reason why attribute *variableTypes* is empty. Our language contains two metrics of this kind. Metric *CyclomaticNumber* gives the number of cycles in a model. The user can customize what is considered a cycle by means of a pattern. This is made of a graph that should be found in the model, and additional graphs constraining the application of the pattern. We have used a similar approach to [5] for graph constraints. The structure of patterns is shown in Figure 2 and discussed in subsection 3.1. Metric *NumberOfElements* measures the number of elements of certain type in a model. The elements to measure are given as a pattern. In this way, we can constraint them (e.g. elements of some type that are not related to elements of some other type).

Class *ElementOriented* corresponds to the second subclassification in our taxonomy, that is, metrics for properties of single model elements. Therefore, only one type has to be specified in the domain. Subclass *RelatedElements* measures the number of elements of certain kind related to a given one. The way in which both are related is given as a pattern.

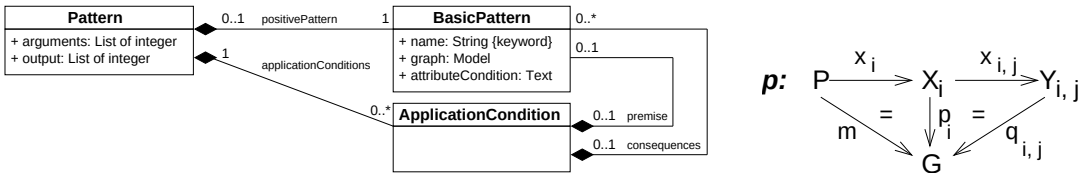
Class *GroupOriented* corresponds to the third subclassification in our taxonomy. We have included just one subclass, *DistanceBased-SimilarityMatrix*, which uses the formula for distance presented in [15]. In this way, if two objects x and y are to be compared, and assume that function $b(\cdot)$ returns the set of relevant properties for the comparison, function: $sim(x, y) = \frac{|b(x) \cap b(y)|}{|b(x) \cup b(y)|}$ gives the similarity between the two elements. The function returns a value in the $[0, 1]$ range. The lower the value, the less similar the two elements are. Then, $dist(x, y) = 1 - sim(x, y)$ gives the distance between the two elements. We have generalized this metric to an arbitrary number of elements of different or the same type. For each type, the set of properties to be measured (function $b(\cdot)$ in the previous formula) has to be specified. This is done with a pattern for each property and is modelled as a qualified relation between the subclass and the pattern. Attribute *orderType* in the relation specifies the type for which the pattern is given. In addition, the comparison can be made by reference (i.e. two objects are considered equal if they are the same), or by value (i.e. two objects are considered equal if all their fields have the same value).

Class *PathOriented* represents metrics of the fourth type in the taxonomy. Our DSVL allows customizing the type of the "node" in the path (attribute *variableTypes*), as well as the fundamental step (by means of a pattern). The result of metric *DistanceMatrix* is a matrix where each position (i, j) denotes the distance between element i and j (i.e. the number of steps to reach j starting from i). Metric *StartPoints* informs about the elements where a path begins (these are called base classes for the case of inheritance). Metric *DirectConnections* measures the number of elements than can be directly reached in one step (e.g. the number of direct children for the case of inheritance). Metric *DepthOfPath* obtains the minimum number of steps that are necessary in order to reach an element starting from a start point (for inheritance this is the depth of inheritance tree). Finally, metric *InheritedElements* is applicable only for inheritance. It measures the number of elements of certain type that are inherited through an inheritance hierarchy. For example, the number of methods that a class inherits from its parent classes. In this metric the relation between the element in the path (e.g. class) and the element that is propagated (e.g. method) has to be given as a pattern.

A fifth metric called *UserDefined* has been added, so that DSVL designers can also define other domain specific metrics, different from the previous ones. The class has a field named *calculation* that allows the designer to include Python code to calculate the metric for a value in the domain. This code is encapsulated in a method that receives as parameters an instance of each

of the types defined in the inherited field *variableTypes* and also the hosting model. The code should return a scalar value as a result of the calculation. In execution time, the method is consecutively invoked once for each value in the domain. Note that this is the only metric where the user has to code the metrics computing procedure, since in the previous ones, the use of patterns is enough for the customization (and subsequent execution) of the metrics.

3.1 Graph Patterns



(a) (b)

Fig. 2. (a) Meta-Model for Patterns and (b) Satisfaction of pattern p by Graph G.

Fig. 2 (a) shows the structure of a *pattern*. It is made of a positive graph condition, and a number of extra graph application conditions composed of a premise graph and a set of consequence graphs. In this way, in order for a pattern to be satisfied by a graph, an occurrence of the positive graph condition has to be found. Then, for each application condition, if the premise graph is found, some of the consequence graphs have to be found as well. The pattern can also be initialized with a partial match (whose elements are given by *arguments*) and produce some output (the elements in the positive graph condition identified by *output*). Note how a *BasicPattern* is made of a graph condition and an attribute condition, which is expressed in some textual language (Python in our case).

Formally, a pattern p is defined using a similar approach to [5] for application conditions, as $p = (P, \bigwedge_{i \in I}(x_i \Rightarrow \vee_{j \in J_i} x_{i,j}))$, where P is the main positive pattern (*positivePattern* in the meta-model) and $x_i : P \to X_i$ and $x_{i,j} : X_i \to Y_{i,j}$ are injective morphisms (X_i is the *premise* and $Y_{i,j}$ are the *consequences* in the meta-model). In this way, a graph G satisfies p (written $G \models p$), if a morphism $m : P \to G$ is found. In addition, if an x_i is specified and a morphism $p_i : X_i \to G$ is found, then some morphism $q_{i,j} : Y_{i,j} \to G$ must also be found, such that both triangles in Fig. 2 (b) commute. Technically, morphisms m, p_i and $q_{i,j}$ are clan-morphisms [1], as instances of abstract classes may appear in P, X_i and $Y_{i,j}$, which are mapped into instances of some class in their inheritance clan. We also require the typing of $Y_{i,j}$ be more concrete than the type of X_i, and this one more concrete than the type of P.

There are two special cases in the application conditions. If for some i no consequence graph is specified, then X_i is a negative application condition

(NAC). On the other hand, if for some i, $P \cong X_i$ and $x_i = id$, then $Y_{i,j}$ (for $j \in J_i$) are positive application conditions.

4 Implementation in AToM^3 and Example

AToM^3 [9] is a meta-modelling tool for the specification of multi-view DSVLs [7]. It has a generative approach, because starting from a meta-model, it generates an environment for the defined language. We have recently improved AToM^3 by adding a tool for the specification of metrics. In this way, the *Metrics* tool enrichs the generated environments for the DSVLs with the possibility to apply customized metrics to the models. This tool was created in AToM^3 itself, using the meta-model in Fig. 1. We completed the meta-model with some elements for the customization of the DSVL environments where the metrics are to be executed. In particular, we added an abstract class *UIButton* (with a single boolean attribute *button*) as the parent of classes *Metrics* and *Actions*. Attribute *button* is set to true if we want to generate a button to execute the metric or action in the DSVL environment. In addition, class *Metric* was provided with two additional attributes. The first one (*genReport*) is of type boolean and is selected in order to obtain a report in *pdf* format with the metric result. Finally, *report* is an enumerate type to select whether the report should show all the obtained values, or only the ones making some threshold condition true. From the *Metrics* meta-model, we used AToM^3's code-generating capabilities to obtain a tool for metrics specification. However, code had to be added by hand for metrics execution control and pattern matching.

Labyrinth [4] is a DSVL for the design of hypermedia and web applications. Hypermedia systems are described as a set of nodes where contents (text, images, etc.) are placed. Links establish the way in which users can navigate in the system. Besides, users can assume roles and belong to different teams from which they receive a set of permissions. Roles and teams can execute certain functions if a relation *permission* exists between them. Besides, roles and teams can be nested in hierarchical structures by means of relation *composition*.

The Ariadne Development Method [3] is based on Labyrinth to build hypermedia and web applications. It proposes a set of artefacts or diagrams that are views of the Labyrinth meta-model. We have used AToM^3 to develop an environment supporting the different Ariadne artefacts. The first step was defining the meta-model for Labyrinth, as it is shown in the background window in Fig. 3. More details regarding the definition of this multi-view DSVL can be found in [7]. Once the meta-model was created, nine different metrics were defined using the *Metrics* tool. This tool can be opened using the button labelled as "Metrics&Redesign" to the left of the window at the background. The *Metrics* tool is shown in the window labelled "2", and contains the specification of the metrics.

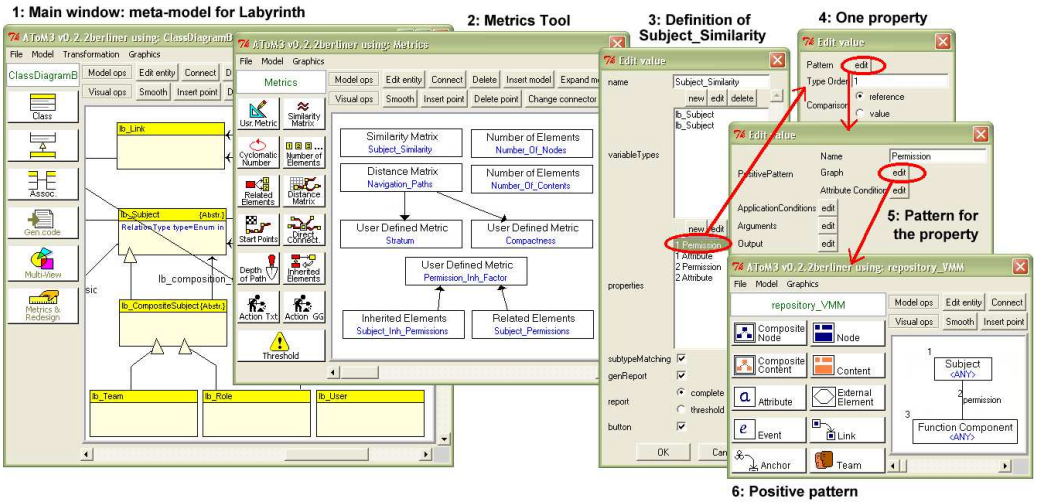


Fig. 3. Metrics Definition for the DSVL Labyrinth.

One of the defined metrics is called *Subject_Similarity*, and is of type *DistanceBased-SimilarityMatrix*. The dialog box to the right (window 3) corresponds to the customization of the attributes for this metric. Attribute *variableTypes* contains type *lb_Subject* twice, as we want to compare subjects to subjects. As it is shown in the meta-model of the main window, a subject is an abstract class that has two concrete subclasses: *lb_Role* and *lb_Team*. In fact, we really want to compare subclasses *lb_Role* and *lb_Team*, because subject is abstract and never appears in any diagram. Therefore, the attribute *subtypeMatching* is checked. The list of properties for evaluating the similarity has also to be customized. In this case we take into account *permissions* and *attributes* for the comparison. This is the reason why the list *"properties"* in window 3 contains items *Permission* and *Attribute* twice (once for each subject). Window 4 shows the specification of the property *Permission*. The visual pattern that describes such property is shown in windows 5 and 6. In particular, window 6 shows the positive graph condition of the pattern. It collects the permissions of a certain subject for function execution. In this way, the access policy can be validated at design time. The argument of this pattern is the element labelled "1" (the subject to be compared) and the output is element "3" (the function). That is, the subject to be compared is passed as a partial match to the pattern, and all connected functions are returned as the result. Nonetheless, these details are hidden to the DSVL designer, who only has to specify the properties as patterns.

Metrics *Number_Of_Nodes* and *Number_Of_Contents* in window "2" are customizations of the model-oriented metric *NumberOfElements*. They count the number of *nodes* and *contents* in our system, providing a measure of its size. In both cases the pattern *element* simply contains an element of the type to be counted.

Metric *Navigation_Paths* is used to calculate the length of navigation paths. It is a customization of metric *DistanceMatrix*. This metric gives a measure

of the minimum number of hyperlinks that a user has to navigate from a page to another one. It is very useful to detect isolated pages or pages of hard access. For this metric, *variableTypes* contains a node of information. Hyperlinks in Labyrinth are expressed with a class named *Link* connected to source and target classes *Anchor*, which in their turn are connected to the source and target *Nodes*. This navigation step has been easily expressed with a pattern, as it is shown in Fig. 4. The argument of the pattern is the element labelled "1", as well as the output. That is, the target node of a navigation step (output) will be the source of the following step (argument). As before, the implementation details are hidden to the DSVL designer, who only has to specify the navigation step as a visual pattern, without coding.

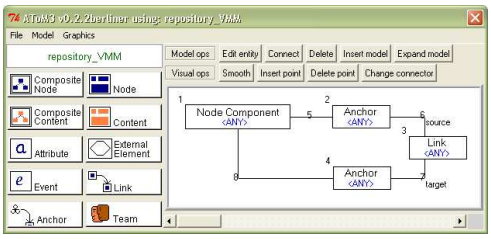
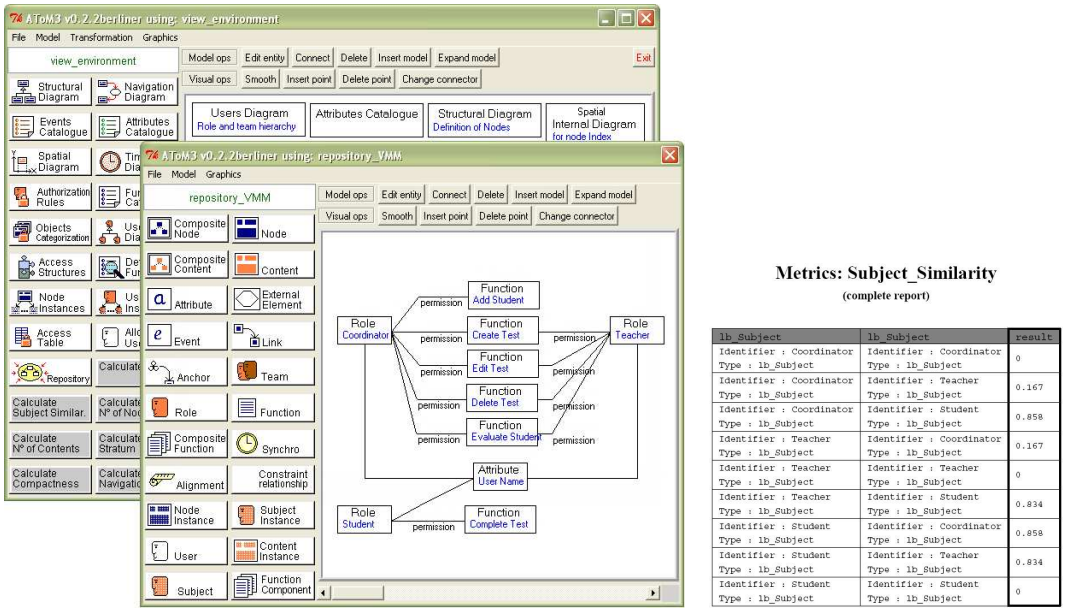


Fig. 4. Pattern for the Specificacion of a Step in Metric *Navigation_Paths*.

User defined metrics *Stratum* and *Compactness* [2] are oriented to the hypermedia domain. The first one is a measure of the linearity of the navigation path and may take values between 0 and 1. The lower the value, the less linear is the path. *Compactness* is a measure of the degree of connectivity of the navigation graph and also takes values between 0 and 1. The lower the value the less connected is the graph. Both metrics are based on the calculation of a distance matrix using the length of the paths between two nodes. This is the reason of the dependency relations between these two metrics and the distance matrix *Navigation_Paths*.

Finally, we have defined three metrics that are generalizations of existing metrics in the object oriented domain. Metric *Permission_Inh_Factor* (PIF) calculates the inherited permission ratio, being an indicator of the reuse. It is a particularization of metrics Method and Attribute Inheritance Factor (MIF and AIF respectively) in the object oriented domain [16]. It is the sum of all the permissions inherited by subjets (roles and teams) divided by the total number of defined permissions (locals and inherited). We have defined auxiliary metrics *Subject_Inh_Permissions* and *Subject_Permissions* to calculate the factors of this division. The first one is a customization of the path-oriented metric *InheritedElements*, and the second one of the element-oriented metric *RelatedElements*. Then, the PIF metric can be calculated from them using a couple of dependency relations. For the two auxiliary metrics no button is generated in the final environment (attribute *button* is set to false).

Fig. 5 (a) shows the environment generated for Labyrinth from the previous definition. In the window at the background, the buttons in white allow the creation of new instances of the Ariadne artefacts. A new artefact is

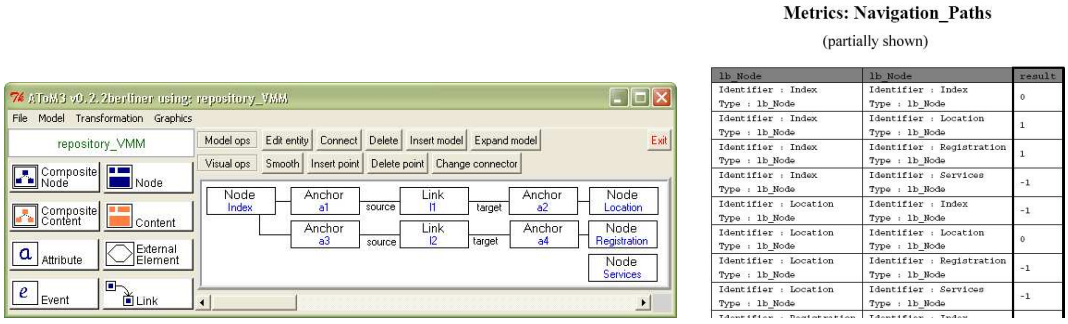


(a) Generated Environment (b) Generated Report

Fig. 5. Calculation of Metric *Subject_Similarity* on a Model.

represented as a box in the window canvas, which later can be edited to include the model. The buttons in grey allow executing the previously defined metrics on the current models. The execution of a metric on a model generates a *pdf* document where the result is shown as a table. For example, Fig. 5 (b) shows the document obtained after executing metric *Subject_Similarity* on the model shown in the foreground window to the left. It can be observed that roles *Coordinator* and *Teacher* are quite similar (distance of 0.167) because they share four functions and one attribute. On the other hand, these two roles are quite different from role *Student*. The application designer could use these results to improve the design by adding a parent role common to both *Coordinator* and *Teacher*, and pulling up the common properties. This could be done using the actions in our *Metrics* DSVL. Note also how sometimes (as in the present case), metrics are not taken on isolated diagrams (which may only contain partial information), but on a *repository* model (see [7]), which contains the union of all the diagrams created by the user.

Fig. 6 shows another sample of metrics execution. To the right it is partially shown the report generated by the execution of metric *Navigation_Paths* on the model to the left. The report shows the minimum number of steps to reach a node from another one. A number of steps equals to -1 indicates that the second node is not reachable from the first one. Node *Services* is isolated since it has a distance -1 from and to any other node.



(a) A Model (b) Generated Report

Fig. 6. Calculation of Metric *Navigation_Path* on a Model.

5 Conclusions and Future work

In this work, we have presented a taxonomy of product metrics, together with a DSVL (called *Metrics*) for specifying metrics for other DSVLs. Our language makes easy the customization of metrics by means of graph patterns. We have implemented these concepts in the meta-modelling tool AToM^3 and shown some examples in the hypermedia domain. The example showed how the use of metrics is especially interesting in the early phases of development in order to improve the design or detect quality defects prior to implementation.

There are a variety of tools which incorporate functionalities for obtaining metrics. Some of them are for the implementation phase [8], and some others for the analysis and design phases [14]. Nonetheless, the set of metrics they provide is usually hard-coded and the possibilities of extension are very limited. One exception is the *SDMetric* tool [14], which allows the definition of metrics for UML using a relational-like language based on XML. Our approach is more general, as we are not restricted to UML, but we can define metrics for any DSVL. In addition, our *Metrics* language is visual, allowing the customization of metrics in a graphical and declarative way. In the area of meta-CASE tools, our work is also original. There is a plethora of this kind of tools (such as GME [10] or MetaEdit+ [12]), but to our knowledge none of them support the definition of metrics.

The presented meta-model is complete in the sense that it is possible to define any metric different from the already generalized ones by using the *UserDefined* metric. Nonetheless, we are currently working in generalizing additional metrics, and on their application to the hypermedia domain. We are also working in general analysis techniques for multi-view DSVLs.

Acknowledgements: This work has been partially supported by the Spanish Ministry of Education and Science with projects MD2 (TIC200303654) and MOSAIC (TSI2005-08225-C07-06).

References

[1] Bardohl, R., Ehrig, H., de Lara, J. and Taentzer, G. 2004. *Integrating Meta-Modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation*. FASE. LNCS 2984, pp.: 214-228. Springer.

[2] Botafogo, R. A., Rivlin, E., and Shneiderman, B. 1992. *Structural analysis of hypertexts: identifying hierarchies and useful metrics*. In ACM Trans. Inf. Syst., 10(2):142–180.

[3] Díaz, P., Montero, S., Aedo, I. 2005. *Modeling hypermedia and web applications: the Ariadne Development Method*. Information Systems, Vol 30(8). pp.: 649-673.

[4] Díaz, P., Aedo, I., Panetsos, F. 2001. *Modeling the Dynamic Behavior of Hypermedia Applications*. IEEE Trans. on Soft. Eng., 27 (6). pp.: 550-572.

[5] Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H. 2004. *Constraints and Application Conditions: From Graphs to High-Level Structures*. ICGT'04, LNCS 3256, pp.: 287-303. Springer.

[6] Fenton, N. E., Pfleeger, S. L. 1998. *Software Metrics: A Rigorous and Practical Approach (2nd edition)*. PWS.

[7] Guerra, E., Díaz, P., de Lara, J. 2005. *A Formal Approach to the Generation of Visual Language Environments Supporting Multiple Views*. Proc. of IEEE VL/HCC'05, Dallas, USA, pp.: 284-286.

[8] Jmetric home page: `http://www.it.swin.edu.au/projects/jmetric`

[9] de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* FASE'02, LNCS 2306, pp.: 174-188. Springer.

[10] Lédczi, A., Bakay, A., Marói, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. 2001. *Composing Domain-Specific Design Environments*. IEEE Computer. pp.: 44-51.

[11] McCabe, T. J. 1976. *A complexity measure*, IEEE Transactions on Software Engineering SE-2, 308–319.

[12] Pohjonen, R., Tolvanen, J-P. 2002. *Automated Production of Family Members: Lessons Learned*. Proc. of PLEES'02, Seattle, USA. pp.: 49-57.

[13] Rozenberg, G. (ed). 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1*. World Scientific.

[14] SDMetric home page: `http://www.sdmetrics.com`

[15] Simon, F., Löffler, S., Lewerentz, C. 1999. *Distance Based Cohesion Measuring*. Proc. 2nd European Software Measurement Conference, pp.: 69-83.

[16] Vázquez, P.J., Moreno, M. N., García, F. J. 2001. *Métricas Orientadas a Objetos*. (In Spanish). Technical Report DPTOIA-IT-2001-02. University of Salamanca.

Semi-automatic generation of metamodels and models from grammars and programs

Andreas Kunert [1]

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany

Abstract

Most recent languages used in the field of computer science (programming languages, modelling languages, ...) are defined by using a grammar-based notation. Although the definition of a language by metamodels is more convenient in terms of understandibility, precision and the ability to reuse abstract concepts from other language definitions, most current textual languages are still missing a complete metamodel. Unfortunately this implies that modern model-based software development tools are not able to process programs written in those languages.

We propose a framework which generates a metamodel for each programming language defined by a grammar. Moreover the framework is able to create a compiler which reads programs of the given grammar and produces models which conform to the generated metamodel. The generation of the metamodel can be adjusted by a predefined set of annotations which can be written directly into the grammar, so the generated model is more appropriate for whichever application.

Key words: metamodels, grammars, programming languages, model transformation

1 Introduction

Model transformations are a key point in the ongoing research on model driven software engineering. Especially the ability to transform models from different modelling languages into each other is a crucial technology especially for the development and use of domain specific languages.

However the majority of today's programs was not created in a model-driven context but (more or less) directly written in a textual programming language. Since those legacy programming languages lack a proper metamodel, programs written in such programming languages can not be processed

[1] Email: kunert@informatik.hu-berlin.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

by model-driven software tools. This gap could be filled by automatically generated metamodels from grammars since most textual languages are defined by grammars. Once a metamodel for a textual language is defined, all programs written in this language can be treated as models (which are instances of the language's metamodel). The concrete transformation from programs to models is more complicated, but the main difficulty of the whole approach lies in the generation of a proper metamodel.

Various applications in the field of model-driven software development would benefit from the ability to treat grammar-based languages as metamodels and programs as models. For instance software reengineering and reconstruction would become significantly easier if the affected programs could be processed by visual modelling tools. The OMG (Object Management Group) defined the term *architecture driven modernization* (ADM [7]) for exactly this task. A similar but different application is the development and use of textual domain-specific languages, which would be the opposite approach to the one described in [6].

In this paper, we will present a framework which is able (i) to generate metamodels from grammars and (ii) to transform programs into models. Moreover, we present a qualitative characterization of the generated metamodels in order to facilitate further model transformation/refinement steps.

The rest of the paper is structured as follows. After this introduction we present some related works. In section 3, we define a measurement of quality which is needed to understand the proposed framework described in section 4. Section 5 describes our current implementation, while section 6 summarizes the paper.

2 Related Work

There are a couple of papers dealing with the connection between grammars and metamodels. A rather formal approach was taken in [1]. In this paper the authors define a relation between grammars and metamodels and describe a mechanism to convert instances from both concepts into each other. Unfortunately, the paper is restricted to the metamodel (M2) level. This implies that the transformation between concrete programs and models (instances of grammars and metamodels) is not handled. Moreover the generated metamodels are rather "flat" due to the fact that they are just different representations of the grammar rules (we will discuss this kind of "quality" of a metamodel in details in section 3).

The solution first described in [10] (and later in [2]) goes further. In this paper the author describes the generation of a metamodel for the ITU-T language SDL [3]. His approach was to generate the metamodel in two steps. A very simple metamodel was generated fully automatically from the grammar and then transformed in a number of manual steps until the metamodel had become a metamodel that was considered sufficient. However, this approach

has two drawbacks: (i) these model transformations are not generic as they are only able to generate the metamodel of SDL and (ii) the model level (M1) is still not handled.

A relation between grammars and metamodels on metamodel *and* model level (M2 and M1) is discussed in [12]. The authors of the paper propose a framework which also works in two steps: model generation and model refinement. Moreover they propose the automatic generation of a model generator which produces models from programs. The proposed framework looks very similar to the one proposed by us but there are differences in the details. They also propose the automatic generation of simple metamodels which will be improved in later model transformations. However we have different opinions about when to solve which specific task and where to annotate the additional information needed to improve the metamodel. While they rely only on model transformation we start to improve our first metamodel before it is even generated. Moreover they propose to add the additional information into the metamodel created in the first step while we do not want to change intermediate models since they will be overwritten if we start the whole generation process again (e. g. if there is a slightly change in the underlying grammar). We propose to add all additional information into the grammar instead.

3 A characterization of metamodel quality

3.1 A measurement of quality

Before we start describing our proposed framework we want to introduce a measurement of quality of metamodels. This is necessary to understand various decisions made in the development of our framework.

As presented in [1], [2] and [12] it is easy to define generation rules, which produce a valid metamodel for a given grammar. Most of those ad hoc algorithms:

(i) generate classes for each grammar symbol (metasymbol and terminal)

(ii) introduce additional classes for occuring sequences, alternatives, optional parts and recurrences

(iii) connect all generated classes according to the grammar rules (by using aggregations, associations and/or generalizations)

In figure 1 a sample grammar for a simple language is shown. Figure 2 shows a metamodel which is created by an ad hoc algorithm.

Metamodels generated by such simple algorithms are not in principle bad or useless. It mainly depends on what the metamodels shall be used for.

Whenever you can define an application for your metamodel, you instantly get a measurement of quality. This measurement is defined rather pragmatically: the more appropriately the metamodel satisfies your needs, the higher quality it has. This implies that there is no global measurement of quality but

program:	(vardefinition | assignment)∗
vardefinition:	type IDENT !SEMICOLON
type:	(INT | FLOAT)
assignment:	IDENT !BECOMES expression !SEMICOLON
expression:	(IDENT | NUMBER) (PLUS expression)?

Fig. 1. A sample grammar describing a very simple programming language (the exclamation marks will be explained in section 4.1 as they are used by our framework)

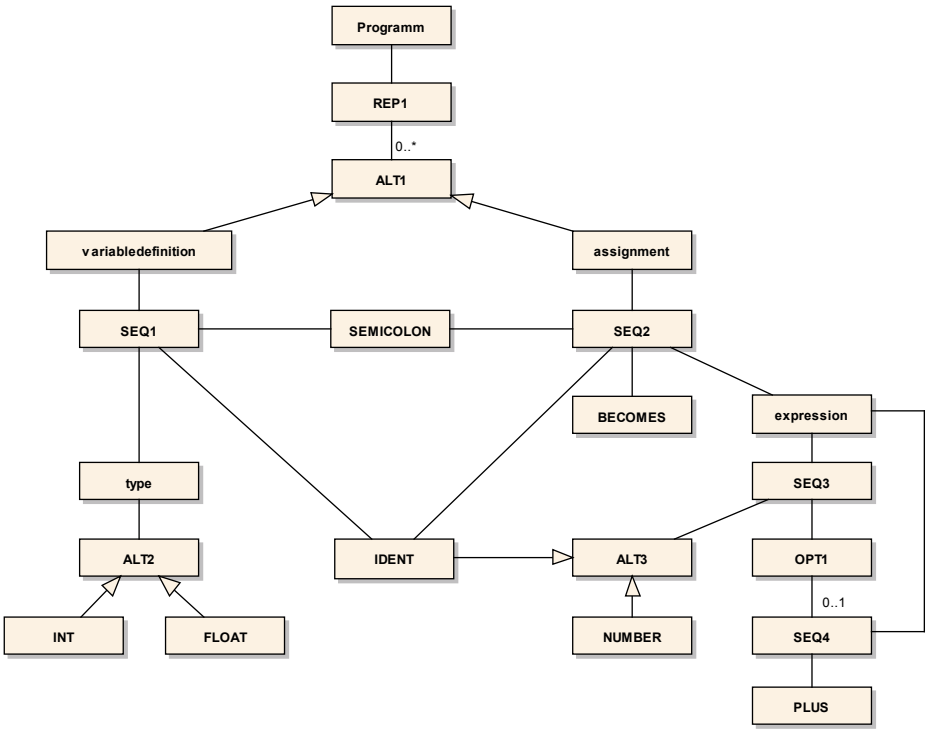


Fig. 2. Metamodel for the example grammar generated by an ad hoc algorithm

many local ones.

When we talk about higher or lower quality metamodels in this paper we consider the application mentioned in the introduction. We are especially interested in metamodels whose instances (i. e., models) can be easily transformed into other models which are instances of other metamodels.

3.2 A high quality metamodel

A metamodel suitable for our needs has to fulfill different requirements. First of all it has to be as abstract as possible (without losing any semantic detail of course).

This implies that it has to represent the semantics of the language and not the syntax. Therefore it is a good idea to start with an abstract grammar and not with a concrete one (as done in [10]). When no abstract grammar

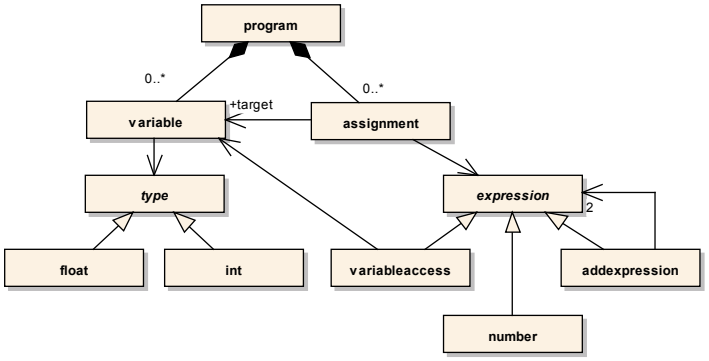


Fig. 3. A higher quality metamodel

is given (as in most textual languages) it is useful to create one by stripping all terminals from the concrete grammar that are only needed for the concrete syntax (e. g. semicolons as statement separators are only needed in the concrete syntax and can therefore be deleted when generating the abstract grammar).

Moreover all constructs which only appear in the metamodel because of the concrete syntax of the grammar as defined by EBNF should be deleted. This means all helper classes for options, repetitions, sequences and alternatives.

Another concept from grammars no longer needed in metamodels are identifiers. Identifiers are only used in programs to reference other parts of the program, but in metamodelling we do not need this helper construct. Associations between referring and referred objects should be used instead. A by-product of this approach is that the models become independent from a concrete identifier notation meaning that if we transform between models of different programming languages, the compatibility of identifiers in the concrete syntax does not have to be checked.

The last and most complicated requirement on a good metamodel for the applications mentioned in the introduction is the efficient use of abstract concepts. Abstract concepts are concepts which appear in different languages (e. g. the concept *namespace* appears as *namespace* in C++, as implication of *package* in Java and so on). It is very convenient to use abstract concepts since subsequent model transformations between different languages become a lot easier if you can simply rely on the mapping between the related abstract concepts of each language.

Figure 3 shows a metamodel which has a much higher quality according to the mentioned requirements than the metamodel of figure 2.

4 Description of the proposed framework

We propose a framework which generates metamodels and models from given grammars and programs, respectively. This generation is performed in two steps. The first step consists of the production of rather simple metamodels

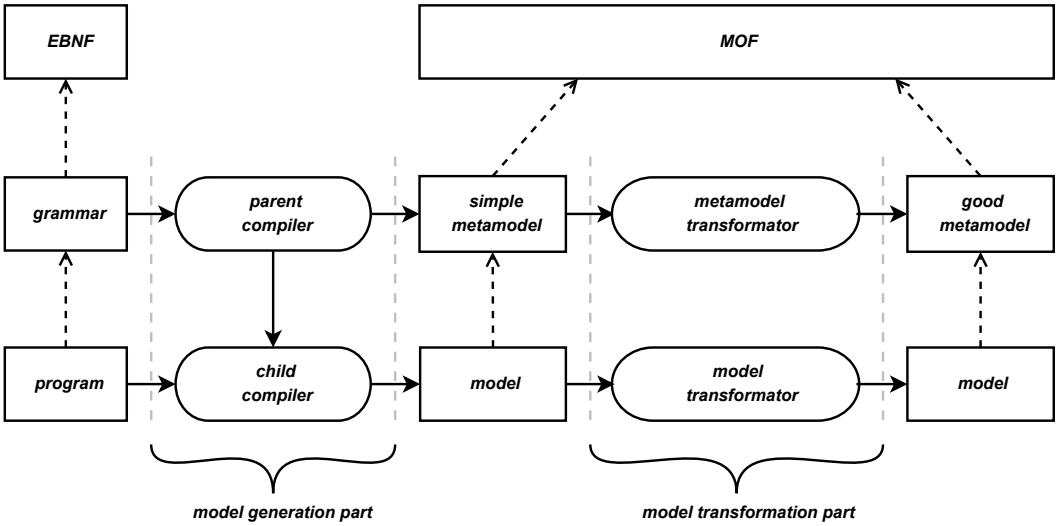


Fig. 4. Overview of the proposed framework

and conforming models using traditional compilers with a connection to a MOF repository. Since the generated models are low quality ones (as described in the previous section) there is a second step which transforms the low quality metamodel into a more appropriate one. Figure 4 gives an overview of the proposed toolchain.

4.1 The model generation part

The compiler part itself consists of two compilers called the *parent compiler* and the *child compiler*. The *parent compiler* reads a grammar written in EBNF and produces a metamodel by using a MOF repository. The EBNF grammar can contain some annotations influencing the metamodel generation. For instance you can mark all terminals which only belong to the concrete syntax but not to the abstract syntax (in our current implementation the exclamation mark is used as shown in figure 1), so they will not become an object in the generated metamodel.

The algorithm for the metamodel generation is a modified and corrected variant of [2]. It consists of three functions shown in figure 5 and 6 (written in pseudocode). The algorithm is a kind of improved ad hoc algorithm: classes are created for every symbol which is not marked for deletion. Then the classes are connected as defined by the grammar, but obviously unnecessary grammar-related constructs are not even created.

Figure 7 shows the metamodel which has been generated by the algorithm presented in figure 5. The difference between our algorithm and the ad hoc algorithms can be seen if figure 2 is used for comparison.

The *parent compiler* also generates the source code of the *child compiler* which is able to parse programs written in the language described by the given grammar. As a consequence a separate *child compiler* is generated for

```
function createMetamodel()
for each grammar symbol X
    if X is not marked for deletion
      create a class named X
for each grammar rule A → B
    if B consists of only one symbol
      connect(A, B, "association")
    else if B is an alternative (B = B_1|B_2|...|B_n)
      for each B_i
        connect(A, subexpression(B_i), "generalization")
    else if B is a sequence (B = B_1 B_2 ... B_n)
      for each B_i
        if B_i is a repetition (B_i = (B'_i)^+)
          connect(A, subexpression(B'_i), "associationMult")
        else if B_i is an option (B_i = (B'_i)?)
          connect(A, subexpression(B'_i), "associationOpt")
        else if B_i is an optional repetition (B_i = (B'_i)*)
          connect(A, subexpression(B'_i), "associationOptMult")
        else
          connect(A, subexpression(B_i), "association")
    else if B is a repetition (B = (B')*)
      connect(A, subexpression(B'), "associationMult")
end function

function subexpression(expression B)
if B consists of only one symbol
    return B
else if B is an alternative (B = B_1|B_2|...|B_n)
    create a class with a unique name C
    for each B_i
      connect(C, subexpression(B_i), "generalization")
    return C
else if B is a sequence (B = B_1 B_2 ... B_n)
    create a class with a unique name C
    for each B_i
      if B_i is a repetition (B_i = (B'_i)^+)
        connect(C, subexpression(B'_i), "associationMult")
      else if B_i is an option (B_i = (B'_i)?)
        connect(C, subexpression(B'_i), "associationOpt")
      else if B_i is an optional repetition (B_i = (B'_i)*)
        connect(C, subexpression(B'_i), "associationOptMult")
      else
        connect(C, subexpression(B_i), "association")
    return C
else if B is a repetition (B = (B')+)
    create a class with a unique name C
    connect(C, subexpression(B'), "associationMult")
    return C
else if B is an option (B = (B')?)
    create a class with a unique name C
    connect(C, subexpression(B'), "associationOpt")
    return C
else if B is a optional repetition (B = (B')*)
    create a class with a unique name C
    connect(C, subexpression(B'), "associationOptMult")
    return C
end function
```

Fig. 5. Our algorithm for the generation of the first (low quality) metamodel (Part 1)

each programming language. Moreover the *child compiler* produces models conforming to the metamodel produced by the *parent compiler*.

```
function connect(expression A, expression B, connectionType T)
if T is "association"
    create an association between A and B
else if T is "generalization"
    create a generalization between A and B
else if T is "associationMult"
    create a association between A and B with the multiplicity 1...n
else if T is "associationOpt"
    create a association between A and B with the multiplicity 0...1
else if T is "associationOptMult"
    create a association between A and B with the multiplicity 0...n
end function
```

Fig. 6. Our algorithm for the generation of the first (low quality) metamodel (Part 2)

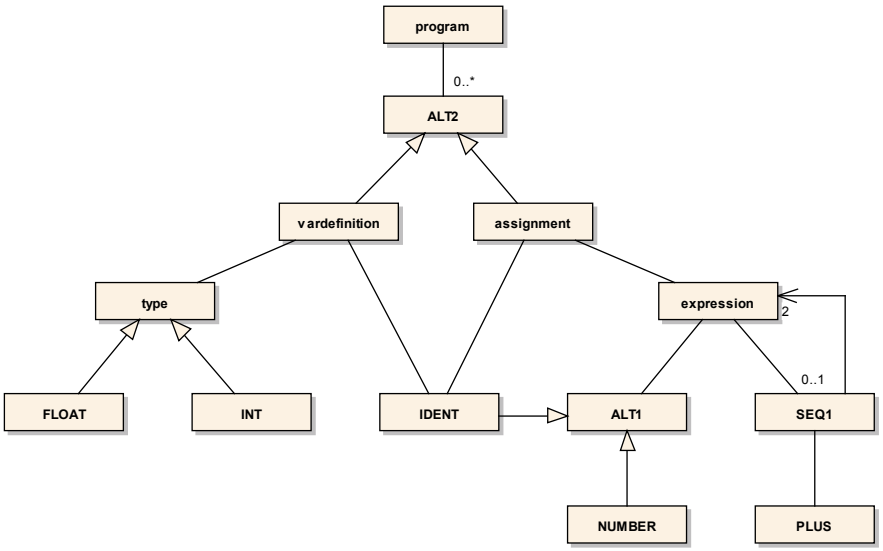


Fig. 7. Metamodel for the example grammar generated by our algorithm

4.2 The model transformation part

Once we have derived a simple metamodel by using the algorithm defined in the previous section, we can start improving it (to gain a metamodel with a higher quality for our purpose). Therefore we propose the introduction of additional annotations written in the grammar read by the *parent compiler*.

For instance identifiers in the metamodel can be deleted if we mark every definition and every use of an identifier with special annotations. Additional annotations can be used for the purpose of renaming classes in the metamodel.

We are still investigating how the introduction of abstract concepts can be expressed by grammar annotations which is by far the most complicate model transformation in our work.

The inclusion of annotations into the grammar file is convenient for the user since he has only to cope with one source file to control the whole model generation process. Moreover later changes in the grammar can be done quite easily.

5 Implementation

We are currently working on a prototype of the proposed toolchain. We already implemented a working *parent compiler* producing simple metamodels and corresponding *child compilers*.

One of our implementation goals was to rely as much on standards and standardized tools as possible, so that our project results can be easily adapted by others.

The *parent compiler* is written with the help of the widely-used compiler generator ANTLR [8], but the source code is easily portable to any other LL(1)-parser generator (e. g. JavaCC [4]). The *child compiler* is also defined by ANTLR grammar specifications generated by the *parent compiler*.

The repository used by the parent and child compilers for model generation and later for model transformation is a MOF 2 repository called *A MOF 2.0 for Java* [11,9]. Since the only interface used to communicate with the repository is JMI [5], the repository can be exchanged with any other JMI-conforming MOF 2 repository without any further changes needed in our implementation.

6 Conclusion

The proposed framework is able to generate metamodels for every given grammar. The generated metamodels are not just other representations of the grammar, but metamodels which only contain the semantic information of the programming language and are therefore a good starting point for further model transformations.

Moreover our framework is able to automatically produce compilers which read programs written in the given languages and produce models according to the generated metamodels.

Once our implementation is finished we have a good base to migrate programs written in many textual languages into the field of (meta-)modelling. This implies that we can use all available model-based tools for software development, reengineering, modernization, etc. on programs written in legacy languages, which will make the mentioned applications much more understandable, easier and less error-prone.

References

[1] Alanen, M. and I. Porres, *A relation between context-free grammars and meta object facility metamodels*, Tucs technical report no 606, Turku Centre for Computer Science (2003).

[2] Fischer, J., M. Piefel and M. Scheidgen, *A metamodel for SDL-2000 in the context of metamodelling ULF*, in: D. Amyot and A. W. Williams, editors, *System Analysis and Modeling: 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1–4, 2004, Revised Selected Papers*, Lecture Notes in Computer Science **3319 / 2005** (2005), p. 208.

[3] ITU-T, "ITU-T Recommendation Z.100: Specification and Description Language (SDL)," International Telecommunication Union, 2002.

[4] java.net, "JavaCC – Java Compiler Compiler," Last checked: February 2006.
URL http://javacc.dev.java.net/

[5] JMI, "The Java Metadata Interface (JMI) Specification (Final Release)," Java Community Process, 2002, JSR-000040.

[6] Muller, P.-A. and M. Hassenforder, *HUTN as a bridge between modelware and grammarware*, in: *WISME Workshop, MODELS/UML 2005*, Montego Bay, Jamaica, 2005.

[7] Object Management Group, "Architecture Driven Modernization (ADM)," Last checked: February 2006.
URL http://adm.omg.org

[8] Parr, T., "ANTLR – Another tool for language recognition," Last checked: February 2006.
URL http://www.antlr.org

[9] Scheidgen, M., "A MOF 2.0 for Java," Last checked: February 2006.
URL
http://www.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava

[10] Scheidgen, M., "Metamodelle für Sprachen mit formaler Syntaxdefinition, am Beispiel von SDL-2000," Diploma thesis, Humboldt-Universität zu Berlin (2004).

[11] Scheidgen, M., *CMOF-model semantics and language mapping for MOF 2.0 implementation*, in: *Joint Meeting of the 4th Workshop on Model-Based Development of Computer Based Systems (MBD) and 3rd International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES), 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, 2006.

[12] Wimmer, M. and G. Kramler, *Bridging grammarware and modelware*, in: *MoDELS Satellite Events*, 2005, pp. 159–168.

Implementing an EJB3-Specific Graph Transformation Plugin by Using Database Independent Queries

Gergely Varró [1,2]

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

Abstract

The current paper presents a novel approach to implement a graph transformation engine as an EJB3-specific plugin by using EJB QL queries for pattern matching. The essence of the approach is to create an EJB QL query for the precondition of each graph transformation rule. Pattern matching and updating phases of a rule application are implemented in a public method of a stateless session bean by executing the prepared EJB QL query and by manipulating persistent objects, respectively.

Key words: graph transformation, EJB 3.0, EJB QL queries.

1 Introduction

Nowadays, the immense role of model transformation concepts and tools is unquestionable for the success of model-driven systems development. Model transformation approaches should support cost and time efficient specification, design, execution, validation and maintenance of manipulations within and between modeling languages. As different phases of transformation design have conflicting requirements, their optimal solution also necessitates different approaches.

In a recent paper [3], we proposed to separate the design of model transformations from their *execution* by generating stand-alone plugins for the EJB 3.0 platform from platform-independent specifications of transformations given by a combination of graph transformation and abstract state machine rules.

Based on the observations of several studies [10,3,14], it may be stated that (i) graph pattern matching is the critical part in graph transformation, and (ii)

[1] This work was partially supported by the SENSORIA European project (IST-3-016004).
[2] Email: gervarro@cs.bme.hu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

very large models can only be handled by EJB3-based plugins with an underlying database as pure Java solutions run out of memory. In the paper, we examine the generation of EJB3-based graph transformation plugins.

In addition to handle very large models, EJB3-based graph transformation plugins have further advantages including (i) the transparent access to system models via a traditional Java interface by hiding the underlying relational database where these models are physically stored, (ii) the atomic execution of graph transformation rules by using the transaction handling mechanism of the application server, (iii) the integration of graph transformation into existing business applications via a standard business logic interface defined by EJB3.

The current implementation of EJB3-based graph transformation plugins (also reported in [3]) performs the computation intensive graph pattern matching on business-level objects. This solution suffers from unnecessary memory handling operations as all objects being traversed must be loaded into the application server at least once, even if the pattern has only a couple of successful matchings.

Our current aim is to improve the performance of pattern matching in graph transformation plugins by using the query support of EJB3. In this case, queries are executed in the underlying relational database, and only those business-level objects are loaded into the application server, which effectively participate in at least one successful matching.

EJB3 provides two declarative languages (the Standard Query Language (SQL) [12] and the EJB Query Language (EJB QL) [11]) for specifying queries. Since the underlying relational databases typically use different dialects of SQL, an approach that uses database dependent SQL queries for describing graph transformation like the one presented in [13] would not be portable.

In order to provide a portable solution, the current paper proposes a novel approach to implement an EJB3-specific graph transformation plugin by using EJB QL queries for pattern matching. The essence of the approach is to create an EJB QL query for the precondition of each graph transformation rule by using *search plans* [17], which have been calculated by some sophisticated algorithms [17,6,16] for the LHS and NAC patterns of the rule in a preprocessing phase. Pattern matching and updating phases of a rule application are implemented in a public method of a stateless session bean by executing the prepared query and by manipulating persistent objects, respectively.

In contrast to [3], the main novelty of this paper is *the usage of database independent EJB QL queries for pattern matching*. Consequently, in the current paper, we only focus on graph pattern matching techniques (as in Sec. 4.2) and completely omit the handling of the updating phase.

The main advantages of the proposed approach include (i) the ability to handle large models in contrast to pure in-memory solutions; (ii) portability and database independence contrary to pure SQL-based approaches; and (iii) reduced memory consumption in the application server compared to other EJB3-based solutions.

The rest of the paper is structured as follows. Section 2 provides a brief introduction to models and metamodels, graph transformation and the main concepts of

search plans. Sec. 3 gives an overview on the EJB3 platform and on the syntax of its query language. In Sec. 4, which is the main part of the paper, we sketch how to encode preconditions of graph transformation rules into EJB QL queries. Finally, some related work is reviewed in Sec. 5, while Sec. 6 concludes our paper.

2 Model manipulation by graph transformation

We first briefly introduce the main notions of metamodels and models, and then show how these models can be manipulated by using graph transformation.

2.1 Metamodels and models

In order to present the concepts of models, metamodels and transformations, a standard object-relational mapping (see e.g. [12]) will be used throughout this paper as a running example, which generates a relational database schema from a UML class diagram.

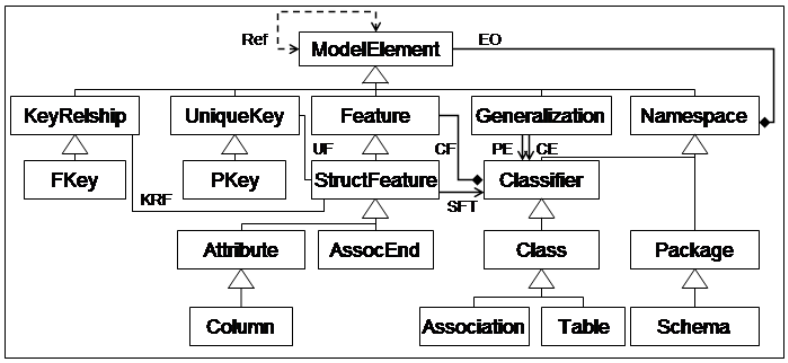


Fig. 1. An extended metamodel for the object-relational mapping

The *metamodel* describes the abstract syntax of a modeling language, which can be formally represented by a type graph. The metamodels of UML class diagrams and relational database schemas (following the CWM standard [9]) are depicted in Fig. 1. Nodes (e.g. Schema, Table) of the type graph are called *classes*. *Associations* like EO, CF, SFT, KRF and UF define connections between classes. Both ends of an association may have a *multiplicity* constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. We consider the most typical multiplicity constraints, which are (i) the at most one (denoted by arrows or diamonds), and (ii) the arbitrary (denoted by line ends without arrows and diamonds). Furthermore, we use one-to-one reference edges (denoted by bidirectional dashed lines in instance models) connecting source and target model nodes. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may specify further associations. Note that the CWM standard derives database notions like tables, columns, etc. from UML notions by inheritance (see Fig. 1). Finally, we assume

without the loss of generality that multiple inheritance is not allowed and both ends of associations are navigable.

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required.

Example 2.1 A well-formed instance model of this domain (going to be shown in Fig. 2(a)) has a single package p that contains two classes (c1 and c2) and the association a. Association a connects class c1 to c2 via association ends ae1 and ae2, respectively. Package p is mapped to a corresponding schema s in the database. Additionally, a table with a single primary key column has already been added to schema s for each content (i.e., c1, c2, and a) of package p.

2.2 Graph transformation

Graph transformation [4] provides a pattern and rule based manipulation of graph models. Each rule application transforms a graph by replacing a part of it by another graph.

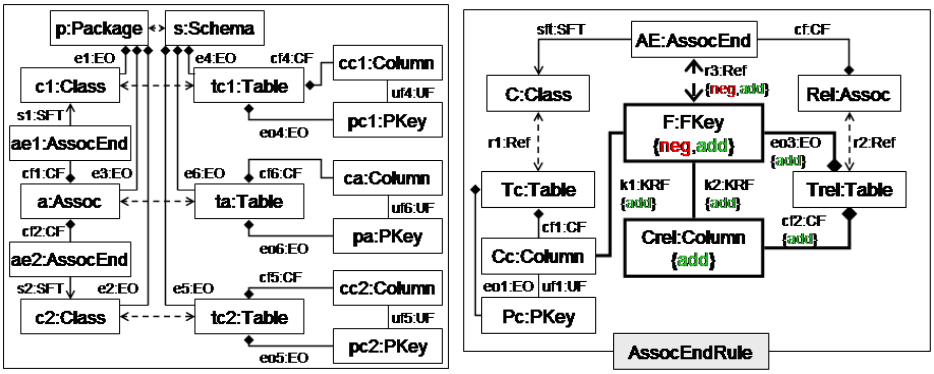
A *graph transformation rule* $\mathsf{r} = (\mathsf{LHS}, \mathsf{RHS}, \mathsf{NAC})$ contains a left–hand side graph pattern LHS, a right–hand side graph pattern RHS, and negative application condition graph pattern NAC [7]. The LHS and the NAC patterns are together called the precondition PRE.

In the paper, we use the graphical representation initially introduced in [5] where the union of these graphs is presented. Elements to be deleted are marked by the del keyword, elements to be created are labelled by the new, while elements in the NAC graph are denoted by the neg keyword.

The *application* of r to an *instance model* M replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M'. A *graph transformation* is a sequence of rule applications from an initial model M_I.

Example 2.2 A single graph transformation rule (AssocEndRule in Fig. 2(b)) is selected as an example for the paper, which handles association ends.

The rule is applicable, if a table Tc with a primary key column Cc already exists for the class C representing the type of the association end AE, and moreover, there is a database table Trel that corresponds to the association Rel whose end is currently processed. The application of the rule creates a new column, which



(a) A sample instance model (b) A sample graph transformation rule

Fig. 2. A sample instance model and graph transformation rule

will refer to the already matched column Cc as a foreign key constraint. Graph transformation rules of the entire object relational mapping are presented in [15].

2.3 Search plans

Informally, a search plan defines a sequence of pattern nodes, which can be used at run-time during pattern matching to control the order of traversal for the objects of the instance model. At first, a search graph is constructed by using the LHS and NAC patterns of the rule. This step is followed by the execution of a sophisticated algorithm (e.g. [17,16]) that generates an optimal search plan on the search graph.

A *search graph* is a directed graph with the following structure. (i) Each node of the pattern is mapped to a *pattern node* (denoted by a solid circle) in the search graph. (ii) A *center node* (denoted by a hollow circle) is also added to the graph. (iii) *Iteration edges* are directed edges connecting the center node to every pattern nodes. The selection of one such edge means an iteration over all objects having the same type as the pattern node being located at the target end of the edge. (iv) Each navigable direction of each pattern edge is mapped to a *navigation edge* in the search graph.[3] The selection of one such edge corresponds to a navigation along the pattern edge in the given direction. If the navigation target of the pattern edge has an at-most-one (arbitrary) multiplicity constraint, then the corresponding navigation edge is referred to a *to-one (to-many) navigation edge*, and it is denoted by an arrow with single (double) arrowhead(s).

Starting nodes (denoted by dashed boxes) mark the center node and the set of pattern nodes that are already matched when the pattern matching starts. The remaining (initially unmatched) pattern nodes are called *traversed nodes* as they are processed during pattern matching, when appropriate objects are to be matched.

A *search plan* is a traversal of such spanning trees of the search graph that are rooted at some starting nodes. A traversal defines a sequence in which edges are

[3] Note that for each pattern edge, a pair of navigation edges having their end nodes connected in both directions is created as the pattern edge is navigable in both directions.

traversed. The position of a given edge in this sequence is marked by increasing integers written *on* the thick edges of spanning trees as in the left part of Fig. 3. In the following, we suppose that a search plan is available for each LHS and NAC.

Example 2.3 Search plans for LHS and NAC patterns of the AssocEndRule are shown in the upper and lower left part of Fig. 3, respectively. As matchings for NAC are searched after pattern matching for LHS is completed, shared nodes (i.e., AE) of LHS and NAC can be considered starting nodes in the search graph of NAC.

3 Enterprise Java Beans 3.0

The Java 2 Enterprise Edition (J2EE) platform defines a layered architecture for scalable, distributed application development including several Java standards and APIs. An enterprise application being developed on the J2EE platform consists of Enterprise Java Beans (EJBs) as its most fundamental building blocks representing business data and functionality. An enterprise application is deployed to and executed by an application server, which provides many high-level services (such as transactions, security, persistence, etc.) beyond the execution of applications.

The two types of EJBs used in the current paper are the following.

- *Entity beans* are persistent objects representing business data, which are kept synchronized with an underlying relational database by means of an object-relational mapping. Entity beans are uniquely identified by their primary key and they can be in relationship with other entity beans referring to each other by direct references (many-to-one or one-to-one relationships) or typed collections (many-to-many or one-to-many relationships).
- *Session beans* implement the business functionality of the application. They can be considered as simple collections of business methods. As our approach does not require any transformation related information to be stored, we use stateless session beans.

EJB Query Language.

An application server has an entity manager unit, which provides operations (i) for creating and removing persistent entity instances, (ii) for finding entities by their primary key, and (iii) for querying over entities.

Queries can be specified in the declarative, object-oriented EJB Query Language (EJB QL) [11]. Due to space limitations, only the structure of the SELECT statement is presented in the current paper, which has the following structure.

```
SELECT select_clause
FROM from_clause
WHERE where_clause
```

The SELECT *clause* denotes the result of the query by a comma separated list of identification variables. An *identification variable* is a variable that can refer to a single instance of a particular entity bean class.

The FROM *clause* designates the domain of the whole SELECT statement by a comma separated list of *identification variable declarations* of the form $type$ AS new_var. The $type$ of an identification variable new_var can be defined explicitly by using the name of an entity bean class, or implicitly by navigating along links of type assoc from an already declared variable old_var. In the latter case, the target class of assoc defines the type of identification variable new_var. Navigation is defined by path expressions old_var`.assoc` and `IN(`old_var`.assoc)`, if the navigation returns a single value and a collection, respectively.

The optional WHERE *clause* is a Boolean expression, and it filters out those results of the query that do not satisfy this expression. A *Boolean expression* is the conjunction (logical AND) of Boolean valued factors, which may test (i) the non-existence of results for a well-formed subquery (`NOT EXISTS (`$subquery$`)`), (ii) the equality of simple factors (sf_1=sf_2), and the (iii) inequality of simple factors (sf_1<>sf_2). A *simple factor* can be a constant, or a navigation operation (denoted by var`.id`) to access the identifier id of an identification variable var.

4 Graph transformation on EJB3 platform

Now we discuss how to generate an EJB3-specific graph transformation plugin, which follows the single pushout [10] approach with injective matchings.

4.1 Mapping metamodels and models to EJB3 entity bean classes and instances

Based on the metamodel, we generate entity bean classes by using the standard object-relational mapping of [11], which can be summarized as follows. (i) A class of the metamodel is mapped to an entity bean class. (ii) The inheritance relations between classes are maintained accordingly. (iii) Each association end with an at most one (arbitrary) multiplicity constraint is mapped to a Java attribute (collection) and two corresponding property accessor (i.e., a getter and a setter) methods in the entity bean class that represents the metamodel class being located at the opposite end of the association. (iv) A Java attribute `id` representing the unique identifier and its two corresponding property accessor methods are added to each entity bean class that does not have a superclass.

Example 4.1 The skeleton of the entity bean class representing a StructuralFeature is as follows.

```
@Entity
public class StructuralFeature extends Feature {
    private Classifier sft;
    private Collection<UniqueKey> uf = new ArrayList<UniqueKey>();
    private Collection<KeyRelationship> krf = new ArrayList<KeyRelationship>();

    @ManyToOne
    public Classifier getSFT() { return sft; }
    public void setSFT(Classifier sft) { this.sft = sft; }

    @ManyToMany(mappedBy="uf")
    public Collection<UniqueKey> getUF() { return uf; }
    public void setUF(Collection<UniqueKey> uf) { this.uf = uf; }
```

```
    @ManyToMany(mappedBy="krf")
    public Collection<KeyRelationship> getKRF() { return krf; }
    public void setKRF(Collection<KeyRelationship> krf) { this.krf = krf; }
}
```

As StructuralFeature is a subclass of Feature, the identifier attribute `id` has not been created. According to the metamodel of Fig. 1, the StructuralFeature class has three incident edges. Consequently, the generated code has three attributes and six accessor methods.

Instance models representing the system under design are stored in an underlying database of the application server. By using entity beans, objects of the instance model can be created, accessed and manipulated exactly the same way as traditional (plain old) Java objects with the single exception that these objects have to be explicitly persisted by calling the `persist()` method of the entity manager.

4.2 Graph pattern matching on EJB platform

By using search plans of LHS and embedded NAC patterns, we construct and execute a single SELECT EJB QL query that calculates and retrieves all the successful matchings of the precondition of a rule.

The general form of the query is as follows:

```
SELECT node_1, ..., node_N
FROM traversed_nodes
WHERE type_checking_constraints AND check_edge_constraints
  AND injectivity_constraints AND NAC_constraints
```

A *traversed node* is an identification variable being declared in the FROM clause of the EJB QL query, which represents a pattern node being processed during the traversal of the search plan.

If a traversed node is reached by navigation in the FROM clause of an EJB QL query, then the type of this traversed node may be an ancestor of the type prescribed by the pattern node itself. This yields a situation where the traversed node possibly has a larger set of matching objects than it is allowed by the type restriction set up by the pattern node. In order to resolve this situation, an additional traversed node is declared for representing the same pattern node and a *type checking constraint* is defined to narrow the set of matching objects for this pattern node.

Traversed nodes declarations and type checking constraints are generated during search plan traversal, which processes search plan edges in increasing order.

Processing iteration edges. If an iteration edge with a target node trg is being processed, then an expression $type_{trg}$ `AS` trg is added to the end of the `FROM` clause where $type_{trg}$ is the type of the pattern node trg.

Processing to-one navigation edges. If a to-one navigation edge of type `assoc` connecting node src to trg is being processed, then expressions src`.assoc AS` trg`_sup` and $type_{trg}$ `AS` trg are appended to the end of the `FROM` clause, and a subformula trg`_sup.id =` trg`.id` is also added as a type checking constraint.

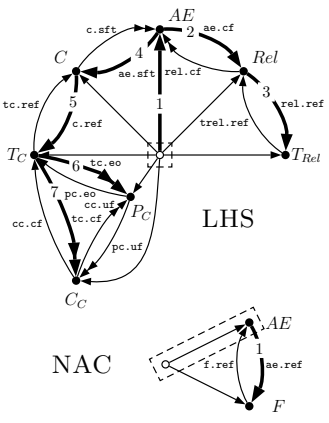
Processing to-many navigation edges. If a to-many navigation edge of type assoc connecting node src to trg is being processed, then terms `IN(`src`.assoc)` `AS` trg`_sup`, and $type_{trg}$ `AS` trg are appended to the `FROM` clause, and a subformula trg`_sup.id =` trg`.id` is also added as a type checking constraint.

An *edge checking constraint* expresses a restriction, which is caused by a pattern edge that has not been processed at all during the traversal of the search plan. For each pair of unnumbered navigation edges connecting nodes src and trg in both directions, we append a subformula src`.assoc.id=`trg`.id` or trg `MEMBER OF` src`.assoc` to the `WHERE` condition by using a logical AND operator for affixing, if src.`assoc` represents a to-one or a to-many navigation edge, respectively.

Injectivity constraints are defined for such pairs of pattern nodes where one member has a type that conforms to a supertype of the other. For each such pair $node_i$ and $node_j$, we add a subformula of the form $node_i$`.id <>` $node_j$`.id`.

NAC constraints express restrictions formulated by NAC patterns that are embedded into the pattern being processed. For each embedded NAC pattern, we add a constraint of the form `NOT EXISTS (`$subquery$`)`, where $subquery$ is the EJB QL query that is going to be generated for the embedded NAC pattern. Note that the `NOT EXISTS` constraint will be evaluated to true if and only if the subquery, which would list the successful matchings of the NAC pattern has no rows.

Example 4.2 To continue our running example, we present the SELECT statement (right part of Fig. 3) that is generated for the search plans of the LHS and NAC pattern of the AssocEndRule (as depicted in the upper and lower left corner of Fig. 3, respectively).




```
1  SELECT ae,rel,trel,c,tc,pc,cc
2  FROM AssocEnd AS ae,                      -- 1 (iter)
3     ae.cf AS rel_sup, Association AS rel,-- 2 (one)
4     rel.ref AS trel_sup, Table AS trel,  -- 3 (one)
5     ae.sft AS c_sup, Class AS c,         -- 4 (one)
6     c.ref AS tc_sup, Table AS tc,        -- 5 (one)
7     IN(tc.eo) AS pc_sup, PKey AS pc,     -- 6 (many)
8     IN(tc.cf) AS cc_sup, Column AS cc    -- 7 (many)
9  WHERE -- type checking constraints
10    rel_sup.id=rel.id AND trel_sup.id=trel.id
11    AND c_sup.id=c.id AND tc_sup.id=tc.id
12    AND pc_sup.id=pc.id AND cc_sup.id=cc.id
13    -- edge checking constraint
14    -- (unprocessed edges between cc and pc)
15    AND cc MEMBER OF pc.uf
16    -- injectivity constraints
17    AND c.id<>rel.id AND c.id<>tc.id
18    AND c.id<>trel.id AND tc.id<>trel.id
19    -- NAC constraint
20    AND NOT EXISTS (
21       SELECT ae,f
22       FROM ae.ref AS f_sup, FKey AS f   -- 1 (one)
23       WHERE f_sup.id=f.id
24    )
```

Fig. 3. Search plans generated for the LHS and the NAC of AssocEndRule and the corresponding EJB QL query

Lines 1–12 of the query are generated during the traversal of the search plan of LHS, when its edges are processed in increasing order as shown by the comments at the ends of lines. (Expressions in parentheses denote the search plan edge processing method being used.) As neither edges between Cc and Pc are processed by the traversal, a corresponding edge checking constraint (lines 13–15) is added to the query. Metamodel classes Association and Table are subclasses of class Class, so C cannot be mapped to the same object as association Rel and tables Tc and TRel, and moreover, matchings for tables Tc and TRel must also differ as expressed by lines 16–18. The query for the NAC pattern (lines 19–24) is processed similarly with the single exception that AE now counts as a starting node as a matching for node AE has already been found.

On the implementation level, we map each graph transformation rule to a public method of the stateless session bean representing the whole graph transformation system. One such method first executes the prepared EJB QL query, then retrieves objects and links needed in the updating phase from the result list, and finally, it manipulates persistent objects. The handling of the updating phase is not mentioned in the current paper as we use the technique presented in [3].

Due to the similarity of the syntax and semantics of SQL and EJB QL queries, the proof for the correctness of the code generation algorithm would be similar to the one presented in [13]. The termination of the algorithm is guaranteed by the finiteness of nodes and edges in the precondition of graph transformation rules.

5 Related Work

Search plans are a widely used technique to *control the order of traversal for the objects of instance models* in algorithms that perform local search for pattern matching meaning that a partial matching is extended step-by-step by neighbouring objects and links. Here we shortly review the four most advanced approaches using *local search with search plans*.

- Fujaba [8] has a token graph based search plan definition [6], which uses a static model for defining the costs of basic operations (i.e., tokens). The optimization of search plans is guided by several well-established rules of thumb.
- PROGRES [17] uses a very sophisticated cost model for defining costs of basic operations of operation graphs, which are similar to search graphs in the current paper. The compiled version of PROGRES generates search plan by a greedy algorithm performed on the operation graph.
- The pattern matching engine of GReAT [1] employs a breadth-first traversal strategy starting from a set of nodes that are initially matched. GReAT also uses simple rules of thumb like Fujaba for search plan generation.
- The compiled version of VIATRA2 [2] employs model-sensitive search plans [16], which are calculated by greedy algorithms performed on search graphs containing statistical data collected from typical instance models.

In contrast to the above-mentioned methods, our approach uses search plans on a syntactic level for the generation of EJB QL queries. As search plans have been optimized in a preprocessing phase, the generated queries give optimal solution for pattern matching on a database independent level. Depending on the features and configuration possibilities of the underlying database, the user may either enforce the same execution order on the database level, or allow its alteration to exploit further database-specific optimization techniques.

6 Conclusion and Future Work

In the current paper, we proposed an EJB3-based graph transformation plugin, which uses queries specified in the declarative EJB QL language for pattern matching. This approach additionally provides a promising, object-oriented and database independent alternative of pure SQL based pattern matching solutions [13].

The essence of the technique is to formulate an EJB QL query and also to generate explicit Java code from search plans for the precondition of each graph transformation rule. The execution of the prepared query and the manipulation of persistent objects implement the pattern matching and the updating phases of graph transformation rule application on the EJB3 platform, respectively.

Our previous experiments [3,13] show that due to the same technology and the underlying relational database, this approach (just like previous EJB3-based graph transformation plugins) is able to handle models having more than 1 million elements for a performance penalty of an order of magnitude (compared to a pure Java solution) in case of smaller models. Based on these experiments, our expectation for the current approach is a slightly better run-time performance, and noticeably reduced memory consumption in the application server compared to solutions, which use pure SQL for specifying queries. As a natural limitation of the approach, it is worth to emphasize the trade-off between portability and run-time performance when database-specific query optimizations are switched on and off. In the future, we plan to carry out experiments to confirm our expectations on both the run-time performance and memory consumption aspects of our approach.

Acknowledgements. The author is very grateful to Dániel Varró for reading initial versions of the paper and giving valuable feedback.

References

[1] Agrawal, A., G. Karsai and F. Shi, *Graph transformations on domain-specific models*, Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University (2003).

[2] Balogh, A. and D. Varró, *Advanced model transformation language constructs in the VIATRA2 framework*, in: *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, 2006, in press.

[3] Balogh, A., G. Varró, D. Varró and A. Pataricza, *Generation of platform-specific model transformation plugins for EJB 3.0*, accepted to SAC 2006.

[4] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, "Handbook on Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools," World Scientific, 1999.

[5] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph rewrite language based on the Unified Modeling Language*, in: G. R. G. Engels, editor, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, LNCS **1764** (1998), pp. 296–309.

[6] Geiger, L., C. Schneider and C. Reckord, *Template- and modelbased code generation for MDA-tools*, in: *Proc. of the 3rd International Fujaba Days*, 2005, pp. 57–62.

[7] Habel, A., R. Heckel and G. Taentzer, *Graph grammars with negative application conditions*, Fundamenta Informaticae **26** (1996), pp. 287–313.

[8] Nickel, U., J. Niere and A. Zündorf, *The FUJABA environment*, in: *The 22nd International Conference on Software Engineering (ICSE)* (2000), pp. 742–745.

[9] Poole, J., D. Chang, D. Tolbert and D. Mellor, "Common Warehouse Metamodel," John Wiley & Sons, Inc., 2002.

[10] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations," World Scientific, 1997.

[11] Sun Microsystems, "JSR 220: Enterprise JavaBeans, Version 3.0," Early draft 2 edition (2005), http://java.sun.com/products/ejb/docs.html.

[12] Ullman, J. D., J. Widom and H. Garcia-Molina, "Database Systems: The Complete Book," Prentice Hall, 2001.

[13] Varró, G., K. Friedl and D. Varró, *Implementing a graph transformation engine in relational databases*, Journal on Software and Systems Modeling (2005), in press.

[14] Varró, G., A. Schürr and D. Varró, *Benchmarking for graph transformation*, in: *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, USA, 2005, pp. 79–88.

[15] Varró, G., A. Schürr and D. Varró, *Benchmarking for graph transformation*, Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics (2005), http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf.

[16] Varró, G., D. Varró and K. Friedl, *Adaptive graph pattern matching for model transformations using model-sensitive search plans*, Tallinn, Estonia, 2005, accepted to GraMoT'05.

[17] Zündorf, A., *Graph pattern-matching in PROGRES*, in: *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, LNCS **1073** (1996), pp. 454–468.

Copying Subgraphs within Model Repositories

Pieter Van Gorp, Hans Schippers, Dirk Janssens

Formal Techniques in Software Engineering
University of Antwerp
{pieter.vangorp,hans.schippers,dirk.janssens}@ua.ac.be

Abstract

The set of operations in state-of-the-art graph transformation tools allows one to conditionally create and remove nodes and edges from input graphs. Node attributes can be initialized or updated with information from other attributes, parameters or constants. These operations appear to be too restricted for expressing model refinements in a concise manner. More specifically, graph transformation lacks an operation for copying subgraphs (multiple connected nodes, including their attributes) to a new location in the host graph. This paper presents a case study that illustrates the need, a syntax and an informal semantics for such an operation. It also discusses how the operation was integrated in an existing graph transformation language. Finally, it indicates how our ongoing effort towards the implementation of a model transformation language based on graph transformation makes optimal reuse of evaluation code for existing language constructs.

Introduction

A *model* can be defined as a simplified representation of a part of the world, named the system [17]. Model *repositories* are databases with specialized support for storing and retrieving models. Their main functionality consists of serializing their data into standard model exchange formats (like XMI [14]), and exposing a query and transformation API (like OCL [12] and JMI [5]). Any program with the purpose of creating or changing models can be called a *model transformation*. The purpose of this paper is to extend graph transformation such that model transformations can be programmed at a high level of abstraction while the low-level APIs of mainstream model repositories are interfaced by means of compilers.

The data definition languages (like MOF [11] and ECORE [9]) for modern model repositories (like MDR [7] and EMF [9]) are object-oriented. Consequently, model repositories can be perceived as object-oriented databases. The data instances in a repository can be perceived as graphs with objects taking the role of attributed nodes. Association, containment, inheritance and other relationships take the role of edges. Transforming data in repositories can thus be perceived as a graph transformation activity.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

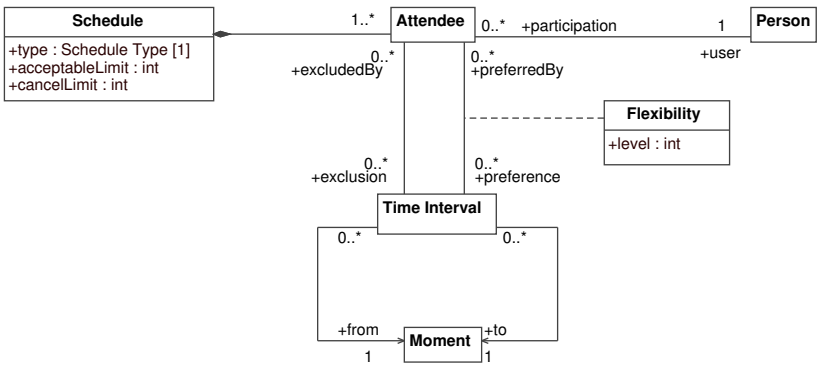


Figure 1. Conceptual Model of a Meeting Scheduler application.

This paper is structured as follows: Section 1 presents two models of a meeting scheduler system. These models are expressed in two different UML profiles and a part of one model should be *generated* from the other one. When using graph transformation to formalize the model transformation that defines this generation process in Section 2, the need for a copy operator becomes obvious. Section 3 presents the syntax and semantics of the proposed copy operator as an extension to Story Diagrams [6]. Additionally, the section briefly compares two approaches for extending an existing Story Diagram engine. The next section refers the reader to related work while the paper concludes with a summary of the contributions and lessons learnt.

1 Motivating Example Models

Figure 1 shows a conceptual model (CM) of a Meeting Scheduler application, specified in UML syntax [13]. At the conceptual level, analysts are free to use constructs such as association classes, views, and other language features. Such features may not be supported directly by the implementation language but they allow one to represent the problem domain as one perceives it in reality as good as possible.

A complete conceptual model contains all relevant nouns and verbs from a problem domain as classes and operations. In order to localize changes to the problem domain, many architectures hide the conceptual model by means of layers. To design such architectures, Rosenberg and Scott *[15]* propose to model user interface screens as *interfaces* and user interface flow as *services*. Only services are allowed to access *entities*, which are based on the classes in a conceptual model. Figure 2 shows a robustness model (RM *[15]*) of the application under study. Note that the entity *Schedule* corresponds to the class *Schedule* from Figure 1.

Figure 3 clarifies how the elements from the conceptual modeling diagram shown in Figure 1 relate to a typed and attributed graph in the underlying model repository. The tree on the left represents the "containment hierarchy view" from the Meeting Scheduler sample in the MagicDraw UML tool. Node *n1* is of type *Model* and represents the UML model that contains both the application examples

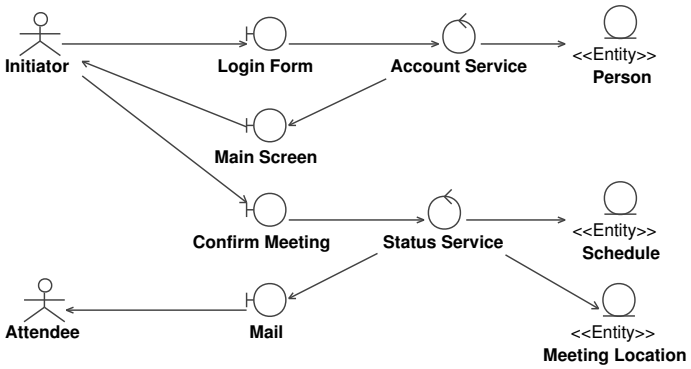


Figure 2. Robustness Model of a Meeting Scheduler application.

and the definitions of the profiles used within these examples. All examples reside in node *n2* of type *UmlPackage* and with name "Examples".

Node *n3* represents the actual Meeting Scheduler sample. This *UmlPackage* contains node *n4* which represents the conceptual model of the Meeting Scheduler. All its contained classes (like *Attendee*, *Flexibility*, ...) map directly to concepts in the problem domain. The containment relationship between *n1*, *n2*, *n3* and *n4* is realized by means of links *l1*, *l2* and *l3* with label "ownedElement". These links can be traversed in the other direction (from contained element to container) as well by means of the "namespace" label. Therefore, the underlying graph is not a directed graph. Moreover, it contains cycles: node n4 (*CM*, the conceptual model of the Meeting Scheduler) is decorated with the "Conceptual Model" stereotype by means of link *l4*. This link can be edited by means of the context-sensitive menu shown in the bottom right corner of Figure 3. The "Conceptual Model" stereotype is defined by node *n7* which is contained in node *n6*, representing the package defining the robustness modeling profile. Due to space limitations, *n7* is not shown in Figure 3. However, the figure does show a node defining another stereotype: node *n5* represents the definition of the "Foreign Key" stereotype from the profile for physical data modeling.

In the following, it will be shown how the entities in the robustness model can be created automatically from the classes in the conceptual model by means of the subgraph copy operator. The idea is to integrate the approach into model editors such that software engineers can focus on design decisions in the model refinement process rather than performing low-level copy operations manually.

2 The *CM2RM* Transformation

This section discusses the nature of the "Conceptual Model to Robustness Model" (*CM2RM*) transformation by presenting a structural and a behavioral model. The structural model will illustrate how the transformation is related to data from the input and output repositories. The subsection discussing the behavioral model will focus on the application of the subgraph copy operator.

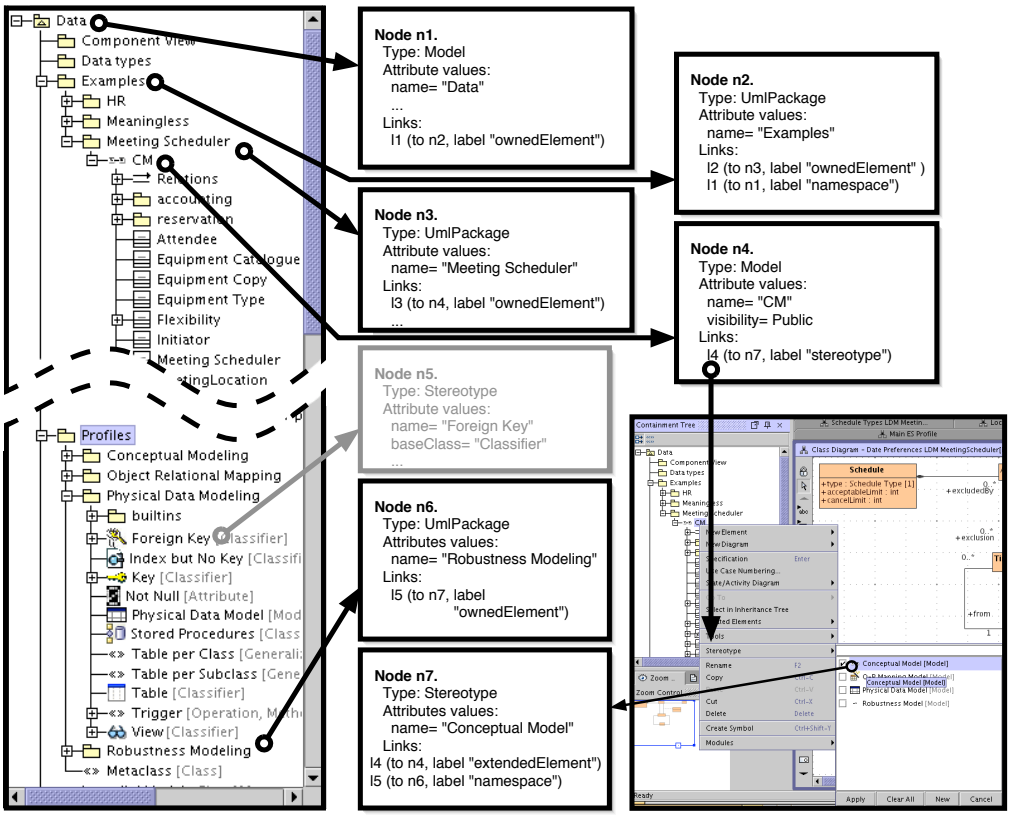


Figure 3. Relation between the UML editor and the underlying model graph.

2.1 *Structural model of the transformation*

As stated in the introduction, the structure of a modern model repository is defined by an object-oriented model. More specifically, such a "*metamodel*" represents the language of the models that can be stored in the repository. Since such metamodels define the input and output types of model transformations, they are discussed for the modeling languages used in the running example. Both the conceptual and the robustness models are expressed in the UML. Since the UML profiles that decorate the standard diagrams with a domain specific syntax are defined as UML models as well, the tranformation under discussion only needs to interact with UML repositories.

Figure 4 shows a structural model of the *CM2RM* transformation. The interesting fact about this diagram is that one does not have to reason about the distinction between transformations, models, metamodels and metametamodels (as defined in [11]) to understand its meaning. It is a traditional class diagram that happens to be used in the context of model transformations but that does not presume any knowledge about the platform-specific repository code that is generated from it.

The *CM2RM* transformation contains a reference to one *Model* (defined in package *org.omg.uml.modelmanagement*) while such a *Model* can be transformed by many *CM2RM* transformations. The *Model* class, its association to the con-

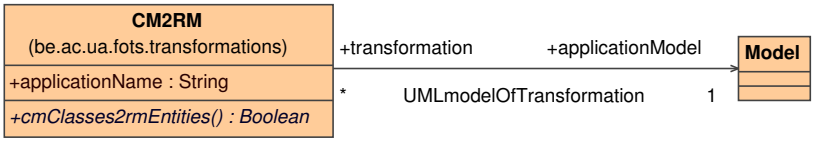


Figure 4. Metamodel defining a repository that allows one to store the CM2RM transformation with a direct reference to the UML model it transforms. The class "Model" is imported from the UML metamodel.

tained UML *Model Elements* (*UmlClass*, *UmlPackage*, *State*, ...) and other concepts from the UML are defined in the UML specification [13]. Since the repositories from popular UML tools are derived from (often even *generated from*) this specification, the class implementing the *Model* concept in MagicDraw does not define a collection of *CM2RM*s. Therefore, the *UMLmodelOfTransformation* association can only be traversed from *CM2RM* to *Model*. In order to apply the *CM2RM* transfomation to the example from Section 1, the *applicationModel* reference needs to be initialized with the "Data" model (node *n1* from Figure 3).

The *CM2RM* transformation can be parameterized with its *applicationName* attribute. This attribute determines what package inside the UML model will be looked up in order to transform the classes in its contained conceptual model to entities in its contained robustness model. When the *CM2RM* transfomation would be applied to the example from Section 1 then *applicationModel* would be set to node *n1* from Figure 3 while *applicationName* would be set to "Meeting Scheduler". This would configure *CM2RM* for execution on *n3*.

While the *CM2RM* transformation could contain more methods for more complete case studies, this paper requires only one which is called "cmClasses2rmEntities". The method does not take any arguments and returns *true* or *false* based on the success of the transformation. The complete behavior of *cmClasses2rmEntities* has already been discussed before [3] but this paper provides a more comprehensive discussion of the *copy operation* used there.

2.2 Behavioral model of the transformation

The behavior of the *cmClasses2rmEntities* method can be modeled in two phases. Firstly, the transformation needs to look up some meta-information for robustness modeling in the UML. Secondly, the classes are copied from the conceptual model to the robustness model and they are marked as entities by decorating them with the proper meta-information. Each of these steps can be implemented as a primitive graph transformation while the order between the primitives needs to be enforced by a controlled graph transformation rule. When using the story diagram syntax, such controlled graph transformation rules are specified as activity diagrams [6].

Figure 5 shows the primitive graph transformation rule for phase two. The rule is written in the UML profile for Story Driven Modeling (SDM) [16], into which the new copy operator is integrated. Unlike the Story Diagram syntax in Fujaba,

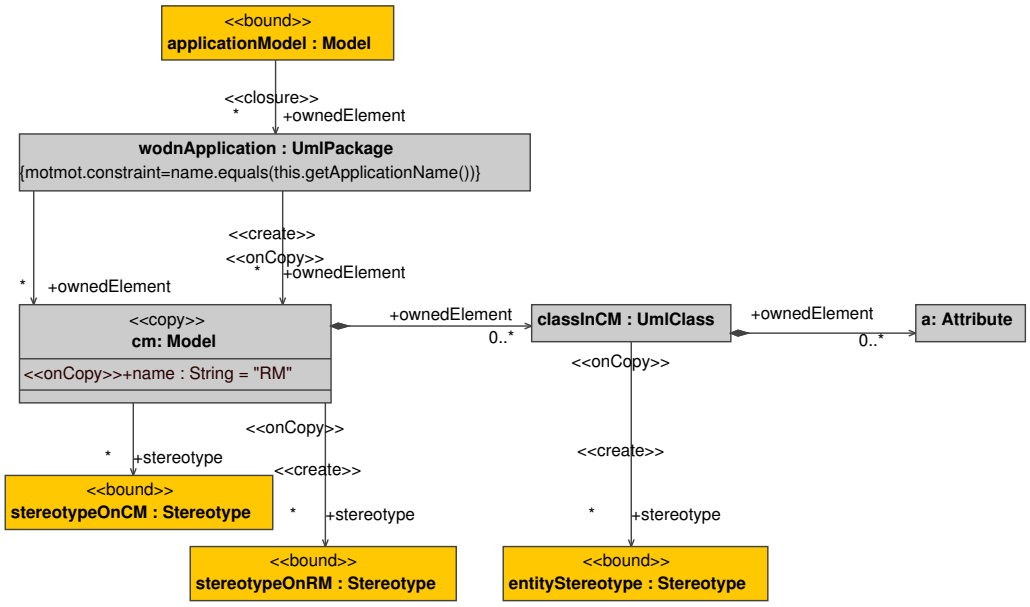


Figure 5. Primitive graph transformation rule applying the new copy operator.

the UML profile for SDM is based on class diagrams instead of object diagrams. This is primarily motivated by syntactical support for the visualization of attribute assignments. Moreover, using class diagrams to model rewrite rules allows one to show the cardinalities of link ends. This assists one to identify sources of multiple matches without looking at the type graph. The following subsections discuss the meaning of the rule in three steps.

2.2.1 Finding a Match

The nodes and edges that do not have a < <create> > stereotype in a primitive story specify a pattern that needs to be found in the input model. The pattern on Figure 5 starts from a node representing *CM2RM*'s *applicationModel* property. As stated, this property represents a handle to the input and output UML model of the discussed transformation (like node *n1* from Figure 4). Just like the *stereotypeOnCM*, *stereotypeOnRM* and *entityStereotype* nodes, the *applicationModel* node is already *bound:* in fact, attributes of transformation classes are bound during the construction of the transformation object while the stereotype nodes are bound by the first primitive graph transformation rule of *cmClasses2rmEntities*.

From the *applicationModel* node, the rule searches for each recursively contained package with its name equal to the *applicationName* property of *CM2RM*. Such a *UmlPackage* is called *wodnApplication* and it represents the application containing the model that needs to be copied. Variable *wodnApplication* would be bound to node *n3* from Figure 4. Note that all nodes and edges are typed and map directly to the class diagrams defining the UML metamodel. The *UmlPackage* nodes that can be reached from the *applicationModel* by recursively traversing outgoing links with association end name *ownedElements* are only bound to the

wodnApplication node if they in turn contain a specific *cm* node in their outgoing *ownedElement* links. A node is bound to *cm* if it is of type *Model* and contains the already bound *stereotypeOnCM* node in its outgoing stereotype links. By specifying that *cm* contains zero or more nodes of type *UmlClass* with zero or more nodes of type *Attribute*, one does not constrain the search for *cm* any further.

2.2.2 Copying the Subgraph

The *cm* node needs to be copied since it is decorated with the <<copy>> stereotype. Apart from the *cm* node, all nodes and links on its outgoing composition path need to be copied as well. Note that all matches on this path are handled since the controlled graph transformation rule that executes this primitive rule marks it as <<loop>>. Without this directive, the primitive rule shown on Figure 5 would copy only one matched class and attribute.

Implicitly, all attributes from a copied node are copied along. For example, since it is of type *Model*, the *name*, *isSpecification*, *isRoot*, *isLeaf* and *isAbstract* attributes of node *classInCM* are copied implicitly. For the definition of the *Model* class, its attributes and superclasses, please refer to the metamodel in the UML 1.5 specification [13].

2.2.3 Using the Copy

When copying a subgraph, one should always store a reference to the copy. Otherwise, it wouldn't become a subgraph of the host graph but just a standalone graph which may be inaccessible in subsequent graph transformations. The undesirable result would be an output model that would not contain the copy.

Creating a link is a standard graph transformation operation. In the UML profile for SDM one needs to specify a link between the nodes that need to be connected and label it with the <<create>> stereotype. Obviously the name of the link and the name and cardinality of the association ends need to conform to an association between the types of the node. Otherwise, the resulting graph would not conform to the output metamodel. In order to create a link from the *wodnApplication* node to the copy of the *cm* node, one needs an explicit notion of node copies in the graph transformation language.

Instead of representing the copy as a node in the transformation rule, the UML profile for SDM is extended with an <<onCopy>> stereotype. By specifying it on the *ownedElement* association end of the <<create>> link that connects *wodnApplication* with *cm*, one expresses that the link should be created to the copy of *cm* instead of to *cm* itself. When the <<onCopy>> stereotype would not be specified on *ownedElement* end, one would erroneously specify that the conceptual model needs to be added to the package it already resides in. The robustness model would be missing from the output model.

The <<onCopy>> instruction is also defined in the context of attribute assignments. This allows one to specify that the name of the robustness model, that is a copy of the *cm* node, needs to be changed to "RM": the attribute assignment on the *cm* node is decorated with the <<onCopy>> stereotype. Without this stereotype

one would change the *name* attribute of the conceptual model.

The <<onCopy>> instruction for <<create>> links is also applied to decorate all classes in the robustness model with the <<entity>> stereotype: the association end at the *classInCM* side of the stereotypes link is decorated with the <<onCopy>> stereotype while the association end at the *entityStereotype* side is left undecorated. The class in the robustness model is indeed a part of the copied subgraph while *entityStereotype* is a node in the original host graph.

The outgoing *type* link of node *a* (of type UML *Attribute*) needs to be copied to the target subgraph as well. A detailed discussion thereof is outside the scope of this paper. In summary, the rule on Figure 5 needs to be extended and an additional loop story is needed. By using multi-objects in combination with the <<onCopy>> instruction one can first create and then query the required traceability data.

3 Subgraph Copy operator

This section presents a syntax and an informal semantics for the proposed copy operator as an extension to the UML profile for Story Diagrams. It also compares two implementation approaches to motivate the direction of the ongoing effort.

3.1 What

The proposed copy operator consists of the following syntactical constructs:

copy The <<copy>> construct allows one to specify what node represents the entry point to the subgraph that needs to be copied.

composition Starting from the <<copy>> node one can specify that a particular match path has composition semantics. Each node and link on this path will be copied.

onCopy The <<onCopy>> construct can be used to indicate that a particular instruction needs to be executed on the copy of an element instead of on the element itself. The construct is defined on (1) association ends of <<create>> links and (2) attribute assignments.

(i) By specifying <<onCopy>> on the source (or target) end of a <<create>> link, one specifies that the link needs to be created from (or to) the copy of the node at that association end.

(ii) An assignment on an attribute from a node on the composition path, that is marked as <<onCopy>>, is executed on the attribute from the copy of this node instead of on that from the node itself.

Not every application of these directives results in a valid use of the copy operator. Therefore, the following new well-formedness rules (WFRs) are defined for the UML profile for SDM:

- At least one link should be created from the host graph to a node from the copied subgraph. More specifically, at least one link should be created to the <<copy>> node or a node on its outgoing composition path.

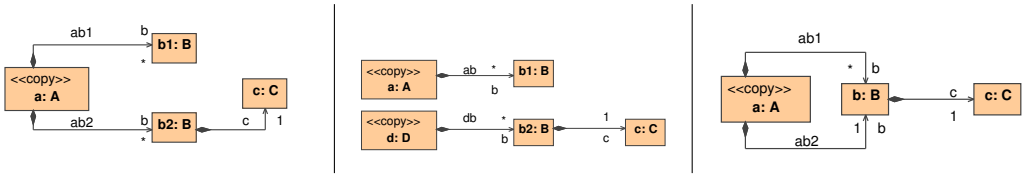


Figure 6. From left to right: two valid rewrite rules and an invalid one. In the rightmost rule, it is unclear whether the *c*'s contained by the *b*'s from *ab1* should be copied, or those contained by the *b*'s from *ab2*, or both. The leftmost rewrite rule illustrates how one can unambiguously specify that for the *b*'s from *ab2* the contained *c*'s should be copied while this is not the case for those from *ab1*. The middle rewrite rule illustrates that within one rewrite rule one can use multiple <<copy>> nodes as long as their composition paths do not overlap.

- The <<onCopy>> instruction should only be applied (1) on attributes inside a copied node, or (2) on association ends connected to a copied node.
- A node should be part of at most one composition. Otherwise, it would be ambiguous what should be the container of such nodes's copy. (see Figure 6).

Appendix A formalizes the first WFR in OCL. The specification is defined within the context of the *Class* class from the *Foundation::Core* package of the UML metamodel. Every instance of that metaclass needs to respect the invariant defined from line 56 onwards. One can use the OCLE tool [8] to confirm that the "cm: Model" node from the transformation rule in Figure 5 respects this invariant. The constraint makes use of three OCL helper attributes defined on line 43 to 49 and 50. The *trfoPkgNodes* attribute represents all nodes from the copy transformation rule under study. The *copiedNodes* and *nonCopiedNodes* attributes divide this set of nodes into the nodes that will or will not be copied respectively. These attributes are defined using the helper operations specified on line 10 and 30. The OCL specification of the latter two WFRs is left out due to space considerations but can be obtained from the authors.

3.2 How

Two implementation approaches have been investigated: a direct model-to-code transformation approach and a model-to-model transformation approach. All related artifacts are publically available in the MoTMoT project [10]. MoTMoT (Model driven, Template based Model Transformer) is a "model transformation" code generator based on the AndroMDA 3.x framework. It uses Freemarker templates to translate UML models (conforming to the profile for SDM [16]) into Java code conforming to the JMI standard.

The straightforward approach for adding support for the copy operator is to extend the Freemarker templates that handle the code generation for existing SDM constructs. At a very high level of abstraction, the generated code should implement the following algorithm:

(i) collect all nodes matching the composition path specified in a copy rule,

(ii) in the case of a complete match: (a) copy these nodes, including all their attributes, and (b) execute < <onCopy> > attribute assignments,

(iii) maintain a map of traceability links between nodes and their copies,

(iv) use the traceability map to create the composition links between the copies as soon as all of the copy nodes have been created,

(v) create < <onCopy> > links using the same approach.

In practice, the complexity of the Freemarker templates reached an unacceptable level after implementing step (iv).

Therefore, current development is focussed on a model-to-model transformation approach that leaves the code templates unchanged. Story Diagrams are used to transform models conforming to the profile discussed in Section 3.1 into models conforming to the SDM profile without the copy operator. The generated Story Diagrams realize the behavior of the copy operator by means of a traceability metamodel and the introduction of additional stories and control structures. The complete transformation is still complex but thanks to the use of an intermediate layer and the modularity mechanisms of Story Diagrams, the complexity can be decomposed into manageable parts. Apart from the facilities for manageing the transformation complexity, the model-to-model transformation approach is promising due to portability opportunities:

- It does not involve a further investment into code specific to the MDR/JMI platform. Migrating the Freemarker templates to platforms such as EMF does not become harder than before.
- With reasonable effort, it should be possible to deploy the story diagrams that are generated by the model transformation on other SDM platforms such as Fujaba.

An upcoming article will discuss this model transformation in more detail.

Related Work

Subgraph copying was first investigated in the context of *hierarchical* graph transformation. This work assumes that one can decompose the transformed graphs into "frames" where edges are not allowed to cross frame boundaries. Drewes, Hoffmann and Plump acknowledge that nested visual languages like the UML require a more flexible decomposition mechanism but require the assumption for proving that rewrite rules do not violate grammatical constraints [2].

Although the hierarchical approach presents the interesting idea of automatically copying all edges between the nodes in a frame, it should be extended for performing copy operations in a more general sense. An < <onCopy> > instruction such as the one presented in this paper could be defined to specify that, for example, the copy of a subgraph should not contain particular edges while including others that do not originate from the source subgraph. Another limitation of the hierarchical approach is that frames are not proposed to be defined on a rule by rule basis. Hoffmann et al. tackled this issue by allowing "shape grammars" to define

the structure of a frame variable in the scope of a rewrite rule instead of in the scope of the complete rewriting system [1].

This paper presents a very specific model *refinement* case study. However, the copy operator can be used for transforming any typed graph with edge labels and attributed nodes. More specifically, it can be used for implementing *refactorings*. Van Eetvelde et al. have proposed the use of graph variables and cloning for raising the abstraction level of graph transformation rules in this context [18]. Applying the copy operator on the *Push Down Method* refactoring defined on a metamodel for Java appears to be promising but the validation of this work is still in progress. This work builds upon the case study from Hoffmann [4] by considering the attributes and links from syntax nodes within method bodies in more detail. We are evaluating whether or not the use of control structures such as a Story Diagram <<loop>> leads to more complex rules than those making use of graph variables.

Conclusion

This paper introduces a graph transformation operator for subgraph copying. The operator allows one to define refinements on models conforming to UML profiles in a concise manner. More specifically: copying model elements from one domain specific model to another one, changing attribute values of copied elements and attaching links to the copied elements can be done in one rewrite rule. The operator has been integrated in Story Diagrams, a controlled graph transformation language with a wide user base. The extension has been implemented in the UML profile for SDM such that any UML 1.5 compliant editor can be used to *model* model transformations. The implementation effort for the transformation engine is focussed on an SDM model transformation from the extended SDM profile to the profile version without the operator. The operator appears to be applicable in the context of model refactoring as well but more validation is required to get a better understanding of its applicability and limitations.

References

[1] Berthold Hoffmann. Abstraction and Control for Shapely Nested Graph Transformation. *Fundamenta Informaticae*, 58(1):39–65, 2003.

[2] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64:249–283, 2002.

[3] Pieter Van Gorp and Dirk Janssens. CAViT: a consistency maintenance framework based on visual model transformation and transformation contracts. In J. Cordy, R. Lämmel, and A. Winter, editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[4] Berthold Hoffmann, Dirk Janssens, and Niels Van Eetvelde. Cloning and expanding graph transformation rules for refactoring. In *International Workshop on Graph and Model Transformation*, Tallinn, Estonia, 2005. A satellite event of GPCE'05.

[5] Java Community Process. Java metadata interface (JMI) specification – JSR 000040, June 2002.

[6] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering. Springer-Verlag, 2005.

[7] Sun Microsystems. NetBeans Metadata Repository, 2002. http://mdr.netbeans.org/.

[8] D. Chiorean A. Carcu M. Pasca C. Botiza H. Chiorean S. Moldovan. *Studia Informatica*, volume XLVII, chapter UML Model Checking, pages 71–88. Babes-Bolyai University, 400084 Cluj-Napoca, Romania, 2002.

[9] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.

[10] Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net/, 2005.

[11] Object Management Group. *Meta Object Facility (MOF) specification*. Object Management Group, 2002. Version 1.4. Available for download at url http://cgi.omg.org/cgi-bin/doc?formal/2002-04-03.

[12] Object Management Group. UML 2.0 OCL Final Adopted specification. ptc/03-10-14, 2003.

[13] Object Management Group. Unified Modeling Language (UML), March 2003. version 1.5. document ID formal/03-03-01.

[14] Object Management Group. XML Metadata Interchange (XMI), v2.0. formal/03-05-02, 2003.

[15] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[16] Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation*, 127(3):5–16, 2004.

[17] E Seidewitz. What models mean. *IEEE Software*, 20, Sept.-Oct. 2003.

[18] N. van Eetvelde and D. Janssens. Extending graph rewriting for refactoring. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *2nd Int'l Conference on Graph Transformation*, volume 3256 of *Lecture Notes in Computer Science*, pages 399–415. Springer-Verlag, 2004.

Appendices

A OCL for Well-Formedness Rule

```
context Class
 -- Return transitive closure of the "ownedElement" links starting from s
 def: let ownedElementTC(s: Set(ModelElement)): Set(ModelElement)=
 if s->includesAll(
   s->select(me1|
     me1.oclIsKindOf(Namespace)
   )->collect(me2|
     me2.oclAsType(Namespace)
   ).ownedElement->asSet()
 ) then s
 else ownedElementTC(
   s->union(
     s->select(me1|
       me1.oclIsKindOf(Namespace)
     )->collect(me2|
       me2.oclAsType(Namespace)
     ).ownedElement->asSet()
   )
 )
 endif

 -- Return from a primitive story all nodes that will be copied
 def: let allCopiedNodes(s: Set(Classifier)): Set(Classifier)=
  s->select(c| -- Return all classes
    hasStereotype(c, "copy") or -- that have a <<copy>> stereotype
    c.association->exists(end| -- or connected to
      end.association.connection->exists(end2| -- an association
        end2<>end and -- of which the other end
        end2.aggregation=AggregationKind::composite -- is of type composite.
      )
    )
  )
 -- Actual WFR: as soon as the <<copy>> instruction is issued, the copied sub-
 -- graph needs to be connected to the host graph by means of a <<create>> link
inv:
 let trfoPkgNodes: Set(Classifier) =
  ownedElementTC(Set{self.namespace})->select(element |
    element.oclIsKindOf(Classifier)
  )->collect(class |
    class.oclAsType(Classifier)
  )->asSet in
 let copiedNodes: Set(Classifier) = allCopiedNodes(trfoPkgNodes) in
 let nonCopiedNodes: Set(Classifier) = -- trfoPkgMEs minus copiedNodes
  trfoPkgNodes->reject(el|
    copiedNodes->exists(copiedNode|
      el=copiedNode -- Reject elements that are copied (set 'minus').
    )
  ) in
  hasStereotype(self, "copy") implies -- When applying the copy instruction,
  nonCopiedNodes.association->exists(end| -- the non-copied nodes should be
    hasStereotype(
        end.association,  -- connected to an association
                "create") and -- representing a <<create>> link
      end.association.connection->select(end2| -- and containing
        end<>end2 -- another end that
      ).participant->exists(copiedNode| -- is connected to a node
        copiedNodes->includes(copiedNode) -- that *is* copied.
      )
  )
```

Towards a Graphical Tool for Refining User to System Requirements

Marco Autili [1]

Dipartimento di Informatica, Università dell'Aquila
I-67010 L'Aquila, Italy

Patrizio Pelliccione [2]

Software Engineering Competence Center, University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg

Abstract

Informal and abstract user requirement specifications are usually complemented by formal and detailed system requirement specifications. While user requirements provide a high level description of what services the system is expected to provide, system requirements provide a more technical specification of how that services should be provided by the system. One of the relevant problems that arise during the Requirement Engineering process is the result of failing to make a clear transition between different levels of requirements description.

Goal of this paper is to introduce a graphical tool for requirements refinement which guides software architects while moving from user requirements to (architectural-level) system requirements. The tool makes use of a previous work that gives a simple but expressive graphical formalism, based on UML2.0 Sequence Diagrams, for specifying temporal properties.

Key words: Graphical Formalisms; Requirements Definition; Software Architectures.

1 Introduction

Formal methods can have an important role in developing reliable, effective computer systems. Verification techniques have been introduced to understand if a system satisfies certain expected properties. These properties are often informally specified as part of user requirements and tools have been

[1] Email: marco.autili@di.univaq.it
[2] Email: patrizio.pelliccione@uni.lu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

proposed to make specification and analysis rigorous and to help software engineers in their work [2]. Even if much work has been done on this direction, the application of such techniques and tools into industrial world can be still very difficult due to some extra requirements and constraints imposed by industrial needs.

For instance, model checker tools allow for automated checking of system model compliance to given temporal properties. These properties are typically specified as linear-time formulae in suitable temporal logics. However, it is a difficult task to accurately and correctly express properties in these formalisms. Properties that are simply captured within the context of interest and that are described in intuitive way by natural language may result very hard to specify in temporal logics. In other words, there is a substantial gap between natural language and temporal logic syntax. As a matter of fact, industries are not willing to use the above mentioned techniques and tools and this slows down the transition from "*research theory*" to "*industry practice*".

What is really needed is a semi-formal and easy to learn methodology for specifying such properties. In addition, the methodology should be time reducing, tool supported (automated tool support is fundamental for strongly reducing human effort and costs) and based on graphical notations that are widespread adopted in industrial contexts. For instance, UML [12] (as standard de-facto for software systems modelling) is one of the most attractive notations. In our context, scenario based formalisms (such as UML2.0 Sequence Diagrams) have been advocated as a means of improving requirements engineering but yet few methods or software assistant tools exist to support Sequence Diagrams based *Requirements Engineering* (RE). Since UML2.0 has not yet provided a formal semantics for its diagrams and operators, it is ambiguous and it is very difficult to develop formal techniques based on its notations. Several approaches have been proposed in the last years trying to give semantics to UML2.0 Sequence Diagrams [17], to pose constraints on UML (based on *Object Constraint Language*) or to develop UML-profiles that solve the ambiguity problem in particular contexts.

In a previous work we presented a simple and (sufficiently) powerful formalism for specifying temporal properties in a user-friendly fashion. We proposed a scenario-based visual language that is an extended graphical notation of a subset of UML2.0 Sequence Diagrams. The language is called *Property Sequence Chart* (PSC). PSC can graphically express a useful set of both *liveness* and *safety* properties in terms of messages exchanged among the components forming the system. We also presented an algorithm, called Psc2Ba, to translate PSC into Büchi automata[3] [3]. Even thought PSC has an intuitive and user-friendly graphical notation, it might be still difficult to directly express properties in this language. In fact, during the early stage of the RE process

[3] In our context, a Büchi automata is an operational description of a temporal property formula. It represents all the system behaviors that respects the logic of the specified temporal property.

the requirements are usually too abstract and vague.

One relevant problem that arises during the requirement engineering process is the result of failing to make a clear transition between different levels of requirements description. According to the terminology adopted in [16], the term "*user requirements*" is used to mean high-level abstract requirement descriptions and the term "*system requirements*" is used to mean detailed and possibly formal descriptions. Often in practice, stake-holders are able to describe user requirements in informal way without detailed technical knowledge. They are rarely willing to use structured notations or formal ones.

Transiting from user requirements to system requirements is an expensive task, even if required. In fact, we are speaking about decisions made during this early phase of the software development process, when the system under development is vague also in the mind of the customer. What we need is a speculative and tool supported process that facilitates understanding and structuring requirements. A well recognized instrument by human society for problems understanding is conversation and discussion. Inspired to the human nature we think that a "*conversational*" tool is what we need at this phase. The tool we are proposing is called W_PSC. By means of a set of sentences (based on expertise in requirements formalization and on a set of well-known patterns [6] for specifying temporal properties used in practice) and classified according to temporal properties main keywords, W_PSC forces to make decisions that break the uncertainty and the ambiguity of user requirements.

2 Background

2.1 Our Context

Software Architecture (SA) acts as a *bridge* between the requirements and the implementation code (which has to reflect architectural properties) [7]. An SA specification represents a high-level design model and captures the system structure by identifying architectural components and connectors in order to assess at an early stage what is the best way to ensure that all key requirements are satisfied.

Usually, software architects go through informal user requirements, talk with customers, analyze existent architectural patterns [15] in order to understand which components they need to use, how such components behave and how they have to be connected. The relationship between requirements and architectures has recently received increased attention [18].

In this context, while user requirements embody some knowledge of the problem domain, system requirements describe properties we expect our system (structured as a given software architecture) satisfies. While user requirements might be informal and ambiguous, system requirements must be well formalized and unambiguous, since they will be used to drive the design and implementation stages and may be used for validating the system model

conformance to user requirements.

That is, the transition from informal user requirements to formal (architecture-level) system requirements is unavoidable and usually relies on stakeholders experience. Moreover, this transition spans from the problem space to the solution space where requirements are described in terms of components, connectors and their interactions.

2.2 Properties Sequence Charts (PSC)

PSC [1,13] is a simple but expressive graphical formalism for specifying temporal properties. Two are the main requirements of PSC, *simplicity* and *expressiveness*. Remaining close to the graphical notation of UML2.0 Sequence Diagrams, the requirement of simplicity is satisfied. The PSC expressiveness is measured with the *property specification patterns* proposed in [6].

PSC describes interactions between a collection of components that can be simultaneously executed and that communicate by message passing. PSC distinguishes among three different types of messages called *arrowMSGs* (see Figure 1.a). *Regular messages*: the labels of such messages are prefixed by "e:". They denote messages that constitute the precondition for a desired (or an undesired) behavior. It is not mandatory for the system to exchange a Regular message; however, if it happens the precondition for the continuations has been verified. *Required messages*: are identified by "r:" prefixed to the labels. It is mandatory for the system to exchange this type of messages. *Fail messages*: the labels are prefixed by "f:". They identify messages that

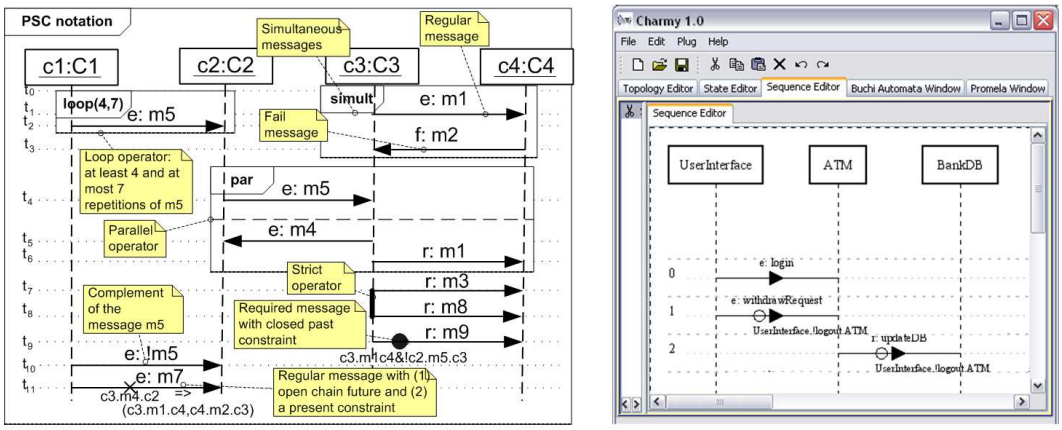


Fig. 1. PSC graphical notation (a) and the PSC tool (b)

should never be exchanged. Fail messages are used to express undesired interactions. We also define *Constraint* operators that impose "restrictions" on the set of messages (called *intraMSGs*) possibly exchanged between the considered message and its predecessor or successor (the predecessor of the first message of a PSC is the startup of the system). Restrictions specify either a chain of *intraMSGs* or a boolean formula (over a set of *intraMSG* labels).

Parallel, *Loop*, and *Simultaneous* operators are introduced with a UML 2.0 like graphical notation. For the sake of brevity, we omit the full description of PSC features and we entirely refer to [1,13] for it.

2.3 Specification patterns for finite-state verification

Specification patterns [6] are a repository with the intent of collecting patterns that commonly occur in the property specification of concurrent and reactive systems. A specification pattern has a scope that defines the range in which the pattern must hold. By recasting the notion of scope in our context, five basic kinds of scopes are distinguished: **Global** (the entire program execution), **Before** (the execution up to a given message), **After** (the execution After a given message), **Between** (any part of the execution from one given message to another one) and **After-Until** (like between but the designated part of the execution continues even if the second message does not occur).

One way to classify the patterns is based on the kinds of system behaviors they describe. A first classification splits the patterns into two main categories: *Occurrence Patterns* and *Order Patterns*. Occurrence Patterns are further partitioned in *Absence*, *Universality*, *Existence* and *Bounded Existence*. Order Patterns are further partitioned in *Precedence*, *Response*, *Chain Precedence* and *Chain Response*. For the sake of readability we do not go through a detailed description of the classification but we refer to [6] for it.

3 The Solution Space

Much effort has been spent in the last years in formalizing requirements and expressing them in some formalism. Scenarios (such as UML2.0 Sequence Diagrams) have been advocated as a means of improving requirements engineering and they have been confirmed as an important design artifact that can be used for a variety of purposes. Scenarios are particularly useful for adding details to an outline requirements description and represent paths of the system behavior representing possible interactions and relationships between participating components.

Several form of scenarios have been developed, each of which provides different types of information at different levels of details. While it could be not difficult to write "high-level" scenarios (e.g., use case scenarios) to document user requirements, more expressive and formal scenario-based notations are needed to document architectural system requirements. While it is useful to keep clear in mind this separation, it is also important *to bound the gap* between these different levels and to create a link among them. An informal and guided decision process should guide developers to move from user to system requirements.

In this section, a methodological approach for generating a formal specification to the user requirements is introduced. We require that the generation

of the formal specification, corresponding to user requirements, has a methodological guidance. Consequently a *"conversational"* graphical tool, which permits to automate the entire process, should be implemented. In the following we describe the *decision helper tool* we have in mind. The tool wants to be an attempt to bridge the gap between possibly informal requirement specifications (as found in practice) and formal ones (as needed in formal methods). The decision helper we are proposing is called W_PSC and it drives software

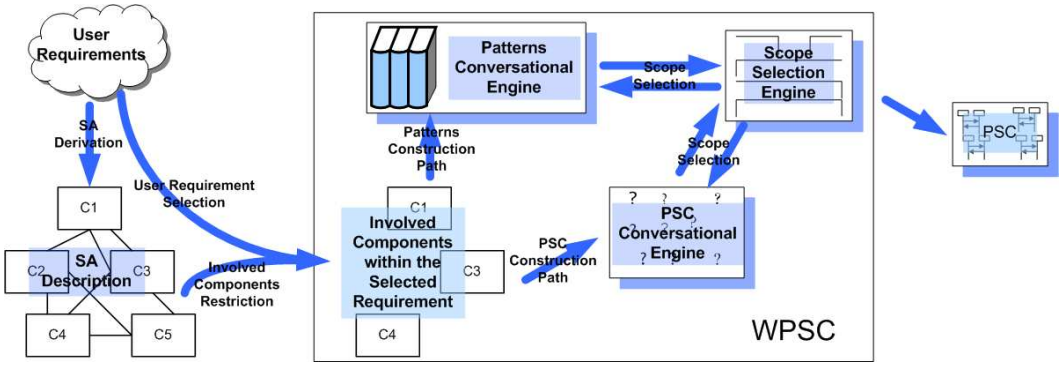


Fig. 2. W_PSC overview

architects in making decisions while writing *Property Sequence Charts* (PSC) introduced in Section 2.2. Within the PSC language, a property is seen as a relation on a set of exchanged system messages, with zero or more constraints. By means of W_PSC all the patterns, briefly described in Section 2.3, can be easily expressed. As already said, each pattern has a *scope* which is the extent of the program execution over which the pattern must hold.

W_PSC should offer a user-friendly wizard helpful while translating a user requirements description into PSC scenarios. In Figure 2 we show the W_PSC framework. The first phase comprises three primary steps: (i) ***Derivation***, (ii) ***Selection*** and (iii) ***Restriction***. By taking into account the requirements description, the first step concerns the (i) derivation of an SA for the system in terms of components, their exchanged messages and connectors. Subsequential, (ii) by selecting one or more user requirements and by identifying an informal property definition within them, the involved components are distinguished. At this point, the SA can be eventually (iii) "restricted" to the subset of distinguished components and related connectors.

Note that, an SA can be derived by extracting real world entities from the requirements description and by mapping these entities into architectural components [16,18]. Moreover, we are assuming that some simple guidelines have been followed to minimize misunderstandings. For instance, we are taking for granted that user requirements, described in the *Requirements Document* (RD), have been written by using language consistently (to distinguish between *mandatory* and *non-mandatory requirements*), by picking out key parts, by avoiding (as far as possible) the use of computer jargon, and possibly by using dedicated structured cards [14]. In this manner the user chooses

between W_PSC sentences in the user requirements specification.

When the first phase has been accomplished, the W_PSC user has in mind the SA (and a set of possible exchanged messages) to be given as input to W_PSC by means of a user friendly visual interface. Since such an interface has been already developed for the CHARMY tool [4] and the PSC tool has been already implemented as its plugin, it is possible to input the architecture by the same CHARMY interface.

Now the wizard is ready to propose a set of sentences that drive the W_PSC user through the ***Construction Path*** while deriving the PSC scenario. The Construction Path is composed of several steps and at each step it poses sentences helpful for requirements understanding and for accurately defining them. The Construction Path is twofold: on one hand, the set of sentences are dedicated to PSC specific features (***PSC Construction Path***); on the other hand it is devoted to the library of property specification patterns (***Pattern Construction Path***).

We split the Construction Path in two different paths because, even though the patterns capture a big variety of common property specifications, we let the user to whether going through a particular and specific solution (by exploiting PSC features) or trying to find an already existent elegant solution (i.e., a pattern). By a series of interactive images, specific text, field and structured dialog boxes, a set of specific sentences are arranged in a *dialog window tree* and a set of *window paths* can be identified from the root to each leaf. Dynamically, according to user decisions, a path is generated and the unique desired PSC scenario is produced.

The wizard engine for supporting the user through the PSC Construction Path is the ***PSC Conversational Engine***. This engine guides the user by means of specific sentences that are brought into focus for PSC features. It might be not easy for a PSC user to choose which type of messages and possible constraints are needed to properly express the informal property he has previously identified from user requirements. Thus, through the window path the user is helped on selecting those messages that are *arrowMSGs* and those ones that are *intraMSGs* (subjected to possible Constraint operators). For each arrow message a type (i.e., Regular, Required and Fail) must be chosen. For *intraMSGs* there will be a window for constructing allowed boolean formulae (through a graphical syntax-directed editor).

The ***Pattern Conversational Engine*** supports the user through the Pattern Construction Path. This engine asks the W_PSC user by a set of ad-hoc sentences focussed on guiding him through the choice of the appropriate patterns category. By taking into account the "original" pattern descriptions [6] (close to temporal logic jargons), we derive a set of non-technical sentences that can be easily understood. In other word, the sentences are formulated as natural language sentences in such a way that the user is able to quickly identify and select the needed patterns category without any particular knowledge of the patterns themselves. We also propose the creation

of a special online help text to answer technical questions for both PSC and Pattern Construction Paths.

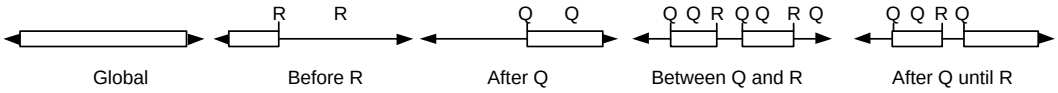


Fig. 3. Pattern scopes

The last phase concerns the ***Scope Selection Engine***. By following the same principles of the above described engines, the Scope Selection Engine exploits window showing interactive images that graphically represent extents of program execution. In Figure 3 we show the graphical representation of the scopes by following the one given in [6]. For W_PSC we propose images based on this representation. Acting with these images the user is driven while selecting the right scope without difficulties.

4 Case Study

In this section we describe how to put into practice the W_PSC approach in a very simple ATM withdrawal case study. Let us suppose that scenarios for withdrawing cash are described as part of the user requirements into the RD. A high-level SA description (depicted in Figure 4) of an ATM system can be derived. It allows users to: buy a refill card for its mobile phone, check its bank account, and make a withdraw operation. The system has been designed as the composition of a set of distributed components: a *User Interface*, the *Phone Company (PC)*, the *Bank DB* and a *ATM* that manages all the interactions between the user and the other entities.

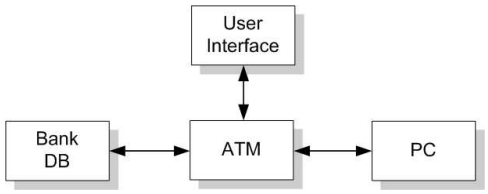


Fig. 4. ATM Application

The informal description of the selected property is: "*The ATM withdrawal shall provide the service for withdrawing cash; there will be a login and logout feature; the ATM will be connected to the bank Data-Base (DB) that will be updated after that a withdraw request has been satisfied*".

The components involved in this property are the *User Interface*, the *Bank DB* and a *ATM*.

Within such a user requirement description, it is not difficult to capture a desirable system property that states: "*If the withdraw request has been satisfied, the bank DB must be updated; the withdraw request is allowed only after the login request and only until the logout request*".

As already said above, it is a non-trivial task to choose both the needed type of message and the right scope. For instance, the above property might be erroneously expressed as an ordered sequence of regular messages *login*, *withdrawRequest*, *updateDB*, *logout* among the interested components. While such a scenario may be correct for scratching a possible system interaction, it is incorrect for formally specifying our system property. In fact, *updateDB* is mandatory and if it is not exchanged, the system will fail.

That is, by using W_PSC, the PSC formalization for this property can be obtained by performing these subsequential steps:

(i) By tacking into account the first part of the property within the above described property (i.e., *"If the withdraw request has been satisfied,"*), the first choice in which the user is guided is recognizing that the withdrawal request is optional. Thus, between the optional sentences, the user is asked to select the right W_PSC sentence (i.e.: If the message *withdraw_request* is exchanged then ...,).

(ii) Considering the following part of the property (i.e., *"the bank DB must be updated;"*) the involved message *bank_DB* is recognized as mandatory. Thus, the user is asked to choose the right sentence between the mandatory sentences (i.e.: The message *bank_DB* must be exchanged).

(iii) The last part of the property (i.e. *"the withdraw request is allowed only after the login request and only until the logout request"*) it is easily recognized as a scope. The Scope Selection Engine proposes the different choices and the user is then guided to choose the "**After-Until**" scope (i.e. "*After login*" and "*Until logout*" scope) that embraces the withdraw request and the DB update.

By composing the chosen sentences the property is rewritten as follows:

"After the *login* message has been exchanged and until the message *logout* is not exchanged, if the message *withdraw_request* is exchanged then the message *bank_DB* must be exchanged."

This description can be "compared" with the informal property definition identified within the RD. Obviously, it will be rarely the same, but it will be simple to understand if the generated property is the wanted one. In other words we are supposing that the user have in mind the property but he needs help in making decision: this textual representation helps in this sense thanks to the given feedback in terms of textual language.

The automatically generated PSC is showed in Figure 1.b. The resulting PSC is different from a simple sequence of ordered messages. Even though the previous example is a toy example, it is not difficult to grasp the main advantage of having such a conversation.

5 Related Work

Several works have been proposed in the last years attempting to bridge the gap between informal requirement descriptions to formal ones. For lack of space, we discuss only those works closest to our approach.

The approach described in [8,10] is able to translate OCL specifications into natural language by a multilingual syntax-directed editor. Even though foundations and design principles might be inherited, in a contrary manner from our approach, they guide the user to transit in the opposite way. In other words, once formal OCL specifications have been produced, they can be translated into natural language descriptions that can be understood by people who do not know OCL.

In [19] authors exploit a software tool that allows system engineers to write detailed use case descriptions using structured templates. The specification is guided by use case style guidelines, temporal semantics and an extensive dictionary of naval domain nouns. Once the use case description phase has been accomplished, system engineers derive use case specifications and, after parametrization, corresponding scenarios are automatically generated. Differently from our proposal, this approach is domain specific and it is dedicated to software engineers with specific domain expertise that are able to directly describe and subsequently specify parameterized use case diagrams.

In [9] the authors present STAIRS, a formal approach to the incremental specification of UML 2.0 interaction for capturing requirements. By referring to Section 2.2, STAIRS is primarily related to our work about PSC [1,13]. Like us, they can deal with mandatory, forbidden and optional scenarios and have the notion of refinements. They use the trace semantics of UML 2.0 but, as we pointed out in [1], UML 2.0 has yet again not provided a formal description of that semantics. Differently from them we provide PSC with a precise semantics via translation, by means of an algorithm implemented as a plugin of our Charmy tool [4], into Büchi automata. After translation PSC diagrams can be directly used by Charmy for model-checking software architectures. Moreover, as it is showed in [1,13], the expressiveness of PSC has been validated with respect to well known *property specification patterns* [6].

In [11] the authors propose an approach called play-in/play-out for specifying and executing behavioral requirements based on LSC [5]. Play-in makes capturing requirements quite intuitive, based on interactions with a prototype of the application GUI. Even if the interaction with the GUI seems fashioning, the user has to choose if the operation performed is mandatory or possible. This is done by selecting in a checkbox list. Thus, as the authors themselves point out, to use complex and sophisticated features of LSC requires being familiar with the LSC language. This aspect represents the more difficult part. In fact the user can friendly interact with the GUI application but is not helped in making decisions for selecting the right checkboxes in the list. On the contrary W_PSC aims exactly in guiding the user in making choices

while exploiting PSC features. Finally the play-in/play-out approach requires specifying a GUI for the application inside the play engine tool. This appears a limitation that can reduce the applicability of the approach to complicated and sophisticated GUI applications.

6 Conclusion and future work

Inspired to the human nature, in this paper we propose a "*conversational*" tool called *Wizard Property Sequence Charts*. By means of posed sentences W_PSC forces to make decisions that break the uncertainty and the ambiguity of user requirements. Sentences are derived from expertise in requirements formalization and from the commonly used *property specification patterns* [6].

W_PSC strives to guide developers while moving from user requirements to (architecture-level) system requirements. The PSC input is an SA and a set of messages possibly exchanged among the components forming the system. At least a non-fine-grained SA can be derived by extracting real world entities from the requirements description. Then, these entities are mapped into architectural components [16,18]. Later the SA can be obviously better refined and all steps we described can be reiterated. As system requirements specification language the tool makes use of PSC that is a simple but expressive graphical formalism, based on UML2.0 Sequence Diagrams, for specifying temporal properties.

On the future work side we plan to conclude the ongoing development of the tool and to actively use it in real industrial contexts in order to empirically evaluate impact and effort needed to use it.

References

[1] M. Autili, P. Inverardi, and P. Pelliccione. A graphical scenario-based notation for automatically specifying temporal properties. Technical report, Department of Computer Science, University of L'Aquila. Sumbitted for publication, 2005.

[2] M. Bernardo and P. Inverardi. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods.* SFM-03:SA Lectures, LNCS 2804, 2003.

[3] J. Buchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Method and Philosophical Sciences*, 1960.

[4] Charmy Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.

[5] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[7] D. Garlan. Software Architecture. In *Encyclopedia of Software Engineering, John Wiley & Sons*, 2001.

[8] R. Hähnle, K. Johannisson, and A. Rantac. An authoring tool for informal and formal requirements specifications. In *FASE '02*, pages 233–248, London, UK, 2002. Springer-Verlag.

[9] Ø. Haugen and K. Stølen. STAIRS - steps to analyze interactions with refinement semantics. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 388–402. Springer, 2003.

[10] K. Johannisson. *Formal and Informal Software Specifications*. PhD thesis, C. Technology and Göteborg Univ., SE-412 96 Göteborg, Sweden, 2005.

[11] R. Marelly, D. Harel, and H. Kugler. Specifying and executing requirements: the play-in/play-out approach. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 84–85, New York, NY, USA, 2002. ACM Press.

[12] Object Management Group (OMG). Unified Modeling Language (UML): Superstructure version 2.0, final adopted specification (02/08/2003).

[13] PSC home page: http://www.di.univaq.it/psc2ba, 2005.

[14] S. Robertson and J. Robertson. *Mastering the Requirements Process*, chapter 6. Harlow: Addison-Wesley, 1999.

[15] M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *COMPSAC97, 21st Int. Computer Software and Applications Conference*, 1997.

[16] I. Sommerville. *Software engineering (7th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[17] H. Störrle. Semantics of Interactions in UML 2.0. In *VLFM'03 Intl. Ws. Visual Languages and Formal Methods, at HCC'03, Auckland, NZ*, 2003.

[18] A. van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures, LNCS 2804*, 2003.

[19] X. Zhu, N. Maiden, and P. Pavan. Scenarios: Bringing requirements and architectures together. In *2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2003.

Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars

Jessica Winkelmann [1] Gabriele Taentzer [2]

Department of Computer Science
Technical University of Berlin
Germany

Karsten Ehrig [3]

Department of Computer Science, University of Leicester, UK

Jochen M. Küster [4]

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland

Abstract

The meta modeling approach to syntax definition of visual modeling techniques has gained wide acceptance, especially by using it for the definition of UML. Based on class structures and well-formedness rules, which could be formalized by OCL, the abstract syntax of visual modeling techniques is defined in a declarative way. Since meta-modeling is non-constructive, it does not provide a systematic way available to generate all possible meta model instances. But for example, when developing model transformations, it is desirable to have a large set of valid instances at hand to perform large-scale testing. In our approach, an instance-generating graph grammar is automatically created from a given meta model. This graph grammar ensures correct typing and cardinality constraints. To satisfy also the given OCL constraints, well-formedness checks have to be done in addition. As a first step, a restricted form of OCL constraints can be translated to graph constraints which are to be checked during the instance generation process.

Key words: OCL, meta model, graph grammar, UML

[1] Email: danye@cs.tu-berlin.de
[2] Email: gabi@cs.tu-berlin.de
[3] Email: karsten@mcs.le.ac.uk
[4] Email: jku@zurich.ibm.com

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Meta modeling is a wide-spread technique to define visual languages, with the UML [UML03] being the most prominent one. Despite several advantages of meta modeling such as ease of use, the meta modeling approach has one disadvantage: It is not constructive i. e. it does not offer a direct means of generating instances of the language. This disadvantage poses a severe limitation for certain applications. For example, when developing model transformations, it is desirable to have a large set of valid instance models available for large-scale testing. Producing such a large set by hand is tedious. In the related problem of compiler testing [BS97] a string grammar together with a simple generation algorithm is typically used to produce words of the language automatically. Generating instance-generating graph grammars for creating instances of meta models automatically can overcome the main deficit of the meta modeling approach for defining languages. The graph grammar introduced in [EKTW05] ensures cardinality constraints, but OCL constraints for the meta model are not considered until now. In this paper we present the main concepts of automatic instance generation based on graph grammars by an example. In addition, we show how restricted OCL constraints can be translated to equivalent graph constraints. The restricted OCL constraints that can be translated can express local constraints like the existence or non-existence of certain structures (like nodes and edges or subgraphs) in an instance graph. Positive ones have to be checked after the generation of a meta model instance, negative graph constraints can be checked during the generation. They can be transformed into application conditions for the corresponding rules, as defined in [EEHP04].

We first introduce meta models with OCL constraints in Section 2. Section 3 presents the main concepts for automatic generation of instances from meta models using the graph grammar approach. The generation process is illustrated at a simplified statechart meta model. We use graph transformation with node type inheritance [BEdLT04] as underlying approach. In Section 4 we explain how restricted OCL constraints can be translated into graph constraints. We conclude by a discussion of related and future work.

2 Meta Models with OCL-Constraints

Visual languages such as the UML [UML03] are commonly defined using a meta modeling approach. In this approach, a visual language is defined using a meta model to describe the abstract syntax of the language. A meta model can be considered as a class diagram on the meta level, i. e. it contains meta classes, meta associations and cardinality constraints. Further features include special kinds of associations such as aggregation, composition and inheritance as well as abstract meta classes which cannot be instantiated.

Each instance of a meta model must conform to the cardinality con-

straints. In addition, instances of meta models may be further restricted by the use of additional constraints specified in the Object Constraint Language (OCL) [Obj05].

Figure 1 shows a slightly simplified statechart meta model (based on [UML03]) which will be used as running example. A state machine has one top CompositeState. A CompositeState contains a set of StateVertices where such a StateVertex can be either an InitialState or a State. Note that StateVertex and State are modeled as abstract classes. A State can be a SimpleState, a CompositeState or a FinalState. A Transition connects a source and a target state. Furthermore, an Event and an Action may be associated to a transition. Aggregations and compositions have been simplified to an association in our approach but they could be treated separately as well. For clarity, we hide association names, but show only role names in Figure 1. The association names between classes StateVertex and Transition are called source and target as corresponding role names. The names of all other associations are equal to their corresponding role names. Since we want to concentrate on the main concepts of meta models here, we do not consider attributes in our example.

The set of instances of the meta model can be restricted by additional OCL constraints. For the simplified statecharts example at least the following OCL constraints are needed:

(i) A final state cannot have any outgoing transitions:
context FinalState inv:
self.outgoing->size()=0

(ii) A final state has at least one incoming transition:
context FinalState inv:
self.incoming->size()>=1

(iii) An initial state cannot have any incoming transitions:
context InitialState inv:
self.incoming->size()=0

(iv) Transitions outgoing InitialStates must always target a State:
context Transition inv:
self.source.oclIsTypeOf(InitialState) implies
self.target.oclIsKindOf(State)

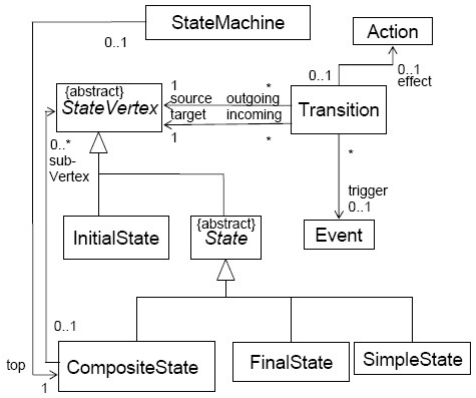


Fig. 1. Meta Model for Statecharts

3 Generating Statechart Instances

In this section, we introduce the idea of an instance-generating graph grammar that allows one to derive instances of a meta model in a systematic way. Given an arbitrary meta model, the corresponding instance-generating graph grammar can be derived by creating specific graph grammar rules, each one depending on the occurrence of a certain meta model pattern. The idea is to associate to a specific meta model pattern a graph grammar rule that

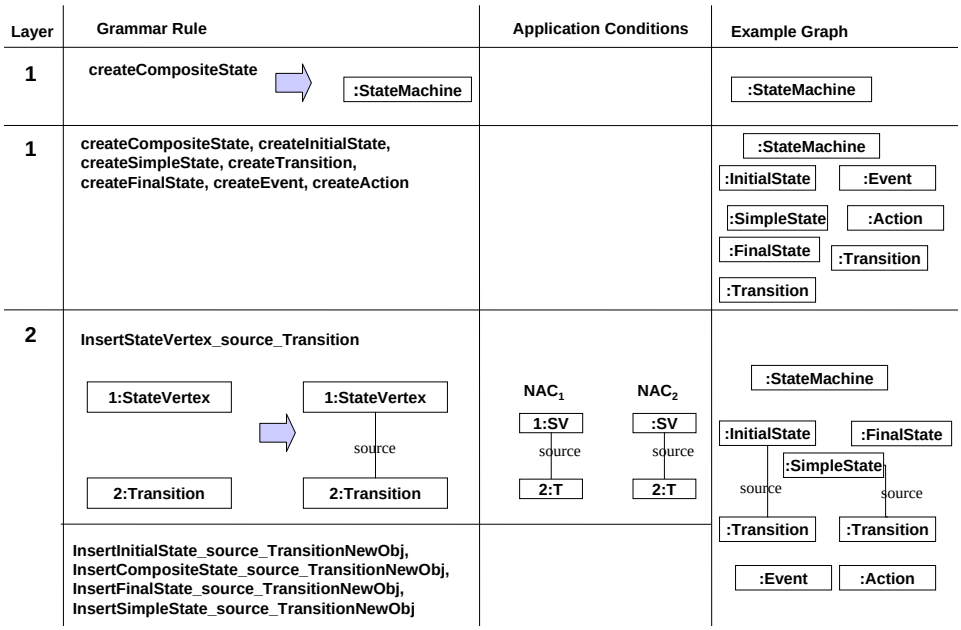


Fig. 2. Example Grammar Rules 1

creates an instance of the meta model pattern under certain conditions. An instance-generating graph grammar also requires a start graph and a type graph. The start graph will be the empty graph and the type graph is obtained by converting the meta model class diagram to a type graph. Given a concrete meta model, assembling the rules derived, the type graph created and the empty start graph will lead to an instance-generating graph grammar for this meta model. For a detailed description see [EKTW05]. Overall, we use the concept of layered graph grammars [EEdL+05] to order rule applications. In the following, we describe the rules that we derive for the meta model of state machines (see Figure 1).

First, we will get a create rule for each non-abstract class within the meta model, allowing us to create an arbitrary number of instances of all non-abstract classes. The rules of layer 1 are applied *arbitrarily often*, meaning that layer 1 does not terminate and has to be interrupted by user interaction or after a random time period. For the sample meta model we get the rules createStateMachine, createCompositeState, createSimpleState, createFinalState, createInitialState, createTransition, createEvent, and createAction in layer 1.

Layer 2 consists of rules for link creation for associations with multiplicity $[1, 1]$ at one association end. The rules have to be applied *as long as possible*. We have rules that create links between existing instances and rules that create an instance (of a concrete type) and a link to this instance starting at an instance that is not yet connected to another instance. New instances can only be created if there are not enough instances in the graph what is ensured by (negative) application conditions. For the association source between StateVertex and Transition, we derive four rules: one rule creates a link source between an existing StateVertex and an existing Transition. Further,

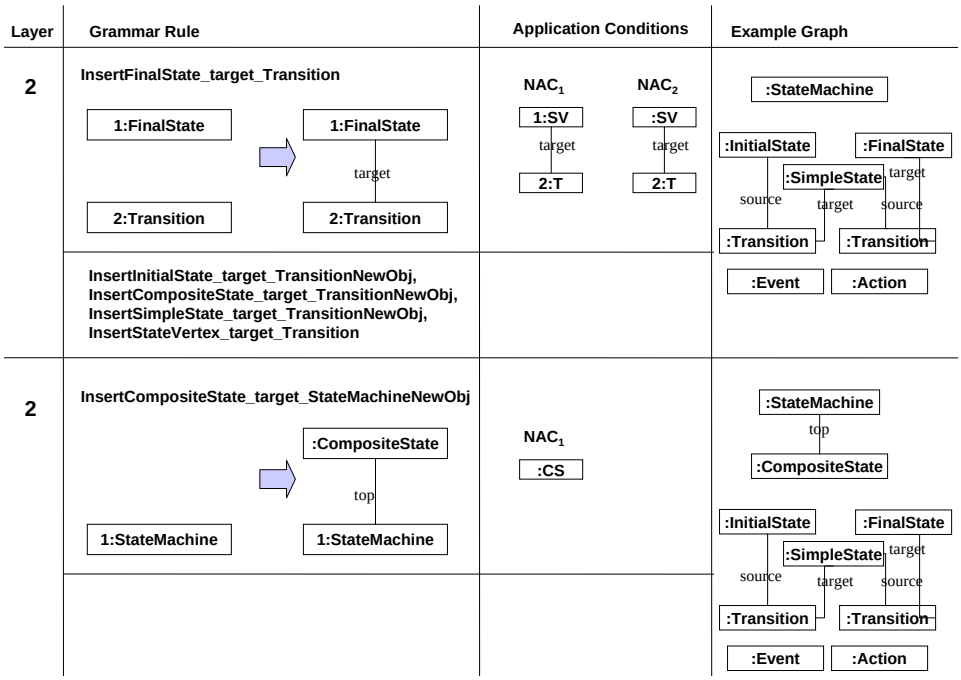


Fig. 3. Example Grammar Rules 2

for each concrete class that inherits from class StateVertex one rule is derived that creates the StateVertex, an InitialState, a CompositeState, SimpleState or a FinalState, and the link source. Note that the abstract class StateVertex could be matched to any of its concrete subclasses InitialState, CompositeState, FinalState, and SimpleState. For the association target between StateVertex and Transition, similar rules are derived. For the association top between StateMachine and CompositeState, we derive two rules. One of them is shown in Figure 3, creating a CompositeState to a StateMachine if no CompositeState exists in the instance graph.

Layer 3 consists of rules creating links for associations with multiplicity $[0, 1]$ or $[0, *]$ at the association ends. The graph grammar derivation rules in layer 3 can be applied *arbitrarily often*. The rules in layer 3 are terminating. But in order to generate all possible instances, the rule application can be interrupted by user interaction or after a random time period. The rules create links between existing instances, so they have negative application conditions prohibiting the insertion of more links than allowed by the meta model cardinalities. For the running example, the rules of layer 3 are shown in Figure 4.

We further get rules that insert links for the association between Transition and Action and the association between Transition and Event as well as association between CompositeState and StateVertex.

The example rules shown in Figures 2 - 4 construct a simple instance graph consisting of a StateMachine with its top CompositeState containing three state vertices and two transitions between them. In the application

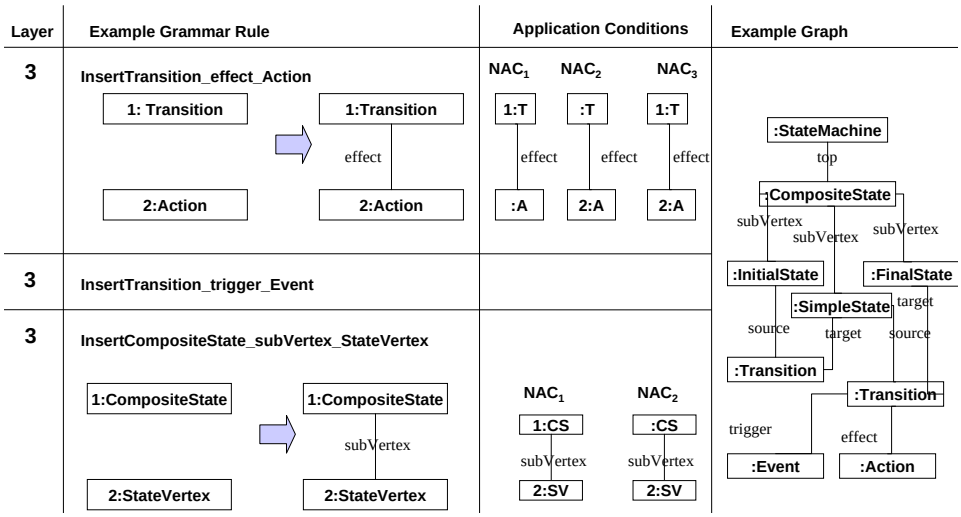


Fig. 4. Example Grammar Rules 3

conditions shown in Figures 2 - 4 the node types are abbreviated (CS for CompositeState etc.).

4 Translation of Restricted OCL Constraints into Graph Constraints

Up to now there is no general way to transform OCL constraints into equivalent graph constraints, which are introduced in [EEHP04]. As a first approach, we show how restricted OCL constraints can be translated to equivalent graph constraints. In contrast to the translation of meta models to graph grammars which was described formally in the previous chapters, we discuss first ideas for the translation of OCL constraints only and sketch how they can be ensured. Besides having one common formalism the motivations for translating OCL constraints into graph constraints is their later consideration within the derivation process (sketched below).

We restrict OCL constraints to equality, size, and attribute operations for navigation expressions, called *restricted OCL constraints*. In future work, OCL constraints and graph constraints have to be further compared concerning their expressiveness.

Graph constraints are properties on graphs which have to be fulfilled. They are used to express contextual conditions like the existence or non-existence of certain nodes and edges or certain subgraphs in a given graph. Application conditions for rules were first introduced in [EEHP04]. They restrict the capability of rules, e.g. a rule can be applied if certain nodes and edges or certain subgraphs in the given graph exist or do not exist.

Definition 4.1 [graph constraint] *Graph constraints* over an object P are defined inductively as follows: For a graph morphism $x : P \to C$, $\exists x$ is a (*basic*) graph constraint over P. For a graph morphism $x : P \to C$ and a graph

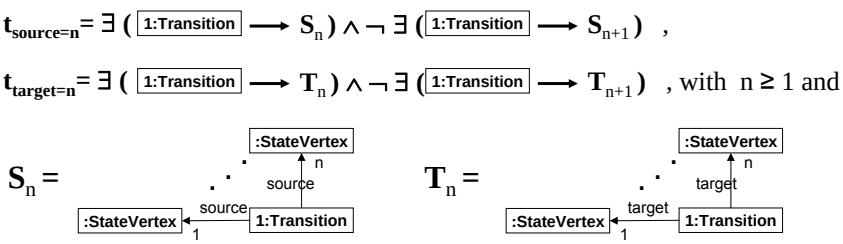


Fig. 5. All Transitions have exactly n source [target] vertices

Fig. 6. All Transitions have the same number of source and target vertices

constraint c over C, $\forall(x, c)$ and $\exists(x, c)$ are (*conditional*) graph constraints over P. For graph constraints $c, c_i (i \in I)$ [over P], true, false, $\neg c$, $\wedge_{i\in I} c_i$ and $\vee_{i\in I} c_i$ are (*Boolean*) graph constraints [over P].

A graph morphism $p : P \to G$ *satisfies* a basic graph constraint $\exists x$ if there exists a graph morphism $q : C \to G$ with $q \circ x = p$. A graph morphism $p : P \to G$ *satisfies* a conditional graph constraint $\forall(x, c)$ $[\exists(x, c)]$ if for all [some] graph morphisms $q : C \to G$ with $q \circ x = p$, q *satisfies* c. A graph morphism p *satisfies* a Boolean graph constraint $\neg c$ if p does not satisfy c; p *satisfies* $\vee_{i\in I} c_i$ $[\wedge_{i\in I} c_i]$ if p satisfies all [some] c_i with $i \in I$.

A graph morphism $p : P \to G$ *satisfies* a basic graph constraint $\exists x$ if there exists a graph morphism $q : C \to G$ with $q \circ x = p$. A graph morphism $p : P \to G$ *satisfies* a conditional graph constraint $\forall(x, c)$ $[\exists(x, c)]$ if for all [some] graph morphisms $q : C \to G$ with $q \circ x = p$, q *satisfies* c. A graph morphism p *satisfies* a Boolean graph constraint $\neg c$ if p does not satisfy c; p *satisfies* $\vee_{i\in I} c_i$ $[\wedge_{i\in I} c_i]$ if p satisfies all [some] c_i with $i \in I$.

A graph G *satisfies* a graph constraint c of the form $\exists x$, $\exists(x, d)$ $[\forall(x, d)]$ if all [some] graph morphisms $p : P \to G$ satisfy c. A graph G satisfies $\neg c$ if G does not satisfy c and $\vee_{i\in I} c_i$ $[\wedge_{i\in I} c_i]$ if it satisfies all [some] c_i with $i \in I$.

With this definition of graph constraints the counting of elements is possible. For the state chart example we can express graph properties like: "All Transitions have exactly n source [target] vertices" (Figure 5), or "All Transitions have the same number of source and target vertices" (Figure 6). Therefore we define Boolean graph constraints $t_{source=n}$ and $t_{target=n}$ expressing that the Transition has n source [target] vertices and not $n+1$ source [target] vertices, where $S_n[T_n]$ denotes the star with n sources [targets]. In the conditional graph constraint in Figure 6 we need the basic graph constraint that maps only a Transition node to a Transition node, since the Transition node in S_n has to be the same as in T_n.

The *restricted OCL constraints* that can be translated are divided into *atomic navigation expressions* and *complex navigation expressions*.

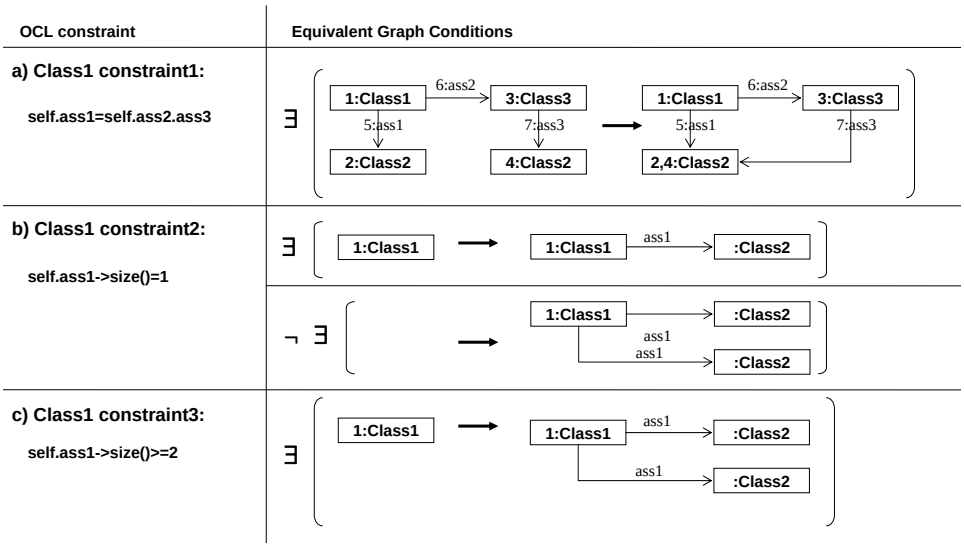


Fig. 7. Examples for Translation of OCL Constraints

Atomic navigation expressions:

Atomic navigation expressions are OCL expressions that

- express equivalent navigations,
- end with operation *size()* (if the result is compared with constants),
- end with operations *isEmpty()*, *notEmpty()* or *isUnique()*, or
- end with attribute operations (not considered explicitly in the paper).

The navigation expressions contain navigation along association ends or association classes only.

Atomic navigation expressions can be transformed into basic graph constraints of the form $\exists x$ or boolean formulae over basic graph constraints.

A navigation expression stating that two navigations have the same result, like self.ass1=self.ass2.ass3, can be transformed into a graph constraint, see Figure 7 a). Here the conclusion of the constraint ensures that ass1 and ass3 are connected to the same instance of Class2.

Operation size() can be translated into a Boolean graph constraint that is composed of two basic graph constraints, see Figure 7 b). The first constraint ensures that there exist the minimum number (= value of the constant) of association ends, the second prohibits the existence of more than the constant value association ends. If the comparison operation is $\leq$ or $\geq$ the OCL constraint can be translated into just one graph constraint.

Operations isEmpty() and notEmpty() can be translated back to a size() operation: self.ass1->isEmpty() is translated back to self.ass1->size()=0, self.ass1->notEmpty() to self.ass1->size()>=1.

Collection operation isUnique() can be translated into a size() operation, if the body of the collection operation is a navigation expression ending at an instance set: self.ass1->isUnique(navexp) is translated back to

self.ass1.navexp->size()<=1.

Complex navigation expressions:

Definition 4.2 [complex navigation expressions] Atomic navigation expressions are complex navigation expressions. Given complex navigation expressions a, b and c, expressions not(a), a and b, a or b, a implies b, and if a then b else c are complex navigation expressions.

Complex navigation expressions can be transformed into conditional graph constraints as described in the following.

An OCL expression of the form a implies b is equivalent to the expression not(a) or b. So we have to translate not(a) or b into an equivalent conditional graph constraint. First the expressions a and b are transformed into graph constraints c_a and c_b as described above. We have to combine the two graph constraints c_a and c_b by the operator $\wedge$, and therefore we have to find the part that has to be identified in both expressions. So we build the intersection expression of the premise and the conclusion, that is a navigation part contained in both expressions (in the example constraint for Transition in Figure 8, the node of type Transition is in this intersection only). This intersection expression ie occurs in the left graph of the graph constraints c_a and c_b. Having c_a, c_b and ie, we can build a conditional graph constraint that is equivalent to the OCL constraints as follows: Build the basic graph constraint $b: \ ie \rightarrow ie$. Build the conditional graph constraint $\exists(b, \neg(c_a) \vee (c_b))$, where the intersection expression is mapped to the corresponding elements in c_a and c_b. See the description of Figure 8 for an example.
OCL constraints of the form if a then b else c can be translated back into two implies operations: (a implies b) and ((not a) implies c). The implies expressions are translated as described before into two graph constraints which then are combined by the logical operator ($\wedge$) to a new one that is equivalent to the OCL constraint.

Ensuring of graph constraints:

Ensuring of graph constraints can be done in two ways: One is to check constraints once the overall derivation of an instance model has terminated which would also be the approach followed when checking OCL constraints directly. However, this leads to the generation of a large number of non-valid instances in between. A more promising approach is to take the constraints into consideration during the derivation process: For each class in the meta model the corresponding graph constraints can be identified. For rules of layer 1, constraints are ignored. For rules of layer 2 and 3, negative constraints of the form $\neg\exists x$, $\neg\exists(x, c)$, $\neg\forall(x, c)$, where x is a basic graph constraint and c is a graph constraint, for the participating classes are evaluated before a possible application of a rule. If the resulting instance violates a constraint, the previous application of a rule is not executed. Here we use the translation of

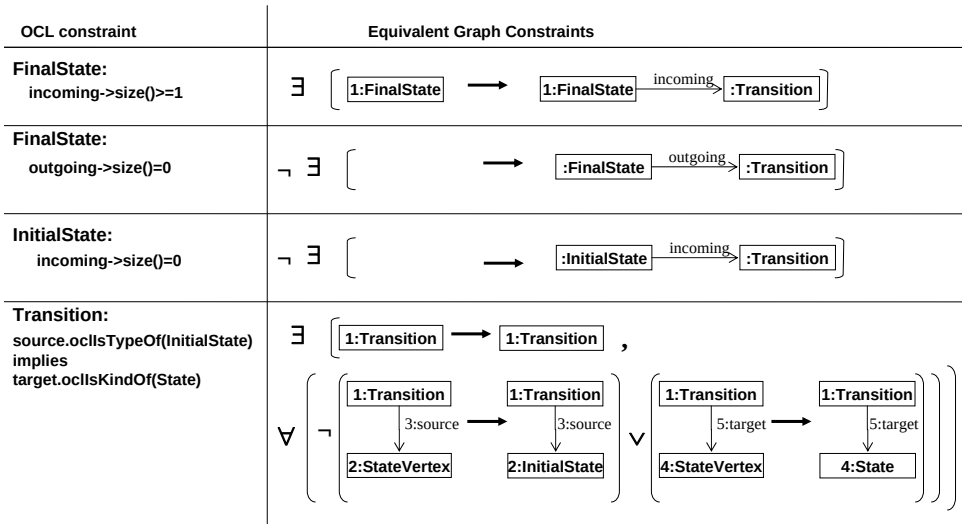


Fig. 8. Translation of OCL Constraints for Statechart Meta Model

graph constraints to application conditions as presented in [EEHP04]. Positive constraints of the form $\exists x$, $\exists(x, c)$, $\forall(x, c)$ are checked after termination of layer 3. If a positive constraint is violated, the model can be fixed by adding additional elements required by the positive constraint. It remains to future work to determine those negative constraints that can be violated by adding the elements required by a positive constraint and to extend the formalization in the previous sections to constraints.

Translation of the OCL constraints for the statechart meta model:

Figure 8 shows the translation of the OCL constraints for the simple statechart meta model example in Figure 1. The first translates the OCL constraint context FinalState inv: self.incoming->size()>=1 (that is an atomic navigation expression) into an equivalent basic graph constraint. This constraint corresponds to the size()-operation constraint shown in Figure 7 c). The second translates the OCL constraint context FinalState inv: self.outgoing->size()=0 into an equivalent basic graph constraint, corresponding to the graph constraints shown in Figure 7 b). Note, that the positive graph constraint is not needed if size() is compared to 0. The third one is similar. The OCL constraint context Transition inv: self.source.oclIsTypeOf(InitialState) implies self.target.oclIsKindOf(State) is a complex navigation expression. It is equivalent to the expression (not(self.source.oclIsTypeOf(InitialState))) or (self.target.oclIsKindOf(State)), stating that each source instance of a Transition instance is not an InitialState or the target instance is a State. The two OCL expressions can be translated into two basic graph constraints shown in Figure 8 (the lower part of the last graph constraint). We have to combine the two basic graph constraints by operator $\vee$ and we have to express that the Transition instance in one expression is the same as in the other expression. Therefore we have to build the intersection expression of the premise and

the conclusion, which contains the Transition only. The complete conditional graph constraint states: all nodes of type Transition have a source node of type InitialState or a target node of type State. This is equivalent to: *Transitions outgoing InitialStates must always target a State.*

5 Related Work

One of the related problems is the one of automated snapshot generation for class diagrams for validation and testing purposes, tackled by Gogolla et al. [GBR05]. In their approach, properties that the snapshot has to fulfill are specified in OCL. For each class and association, object and link generation procedures are specified using the language ASSL. In order to fulfill constraints and invariants, ASSL offers try and select commands which allow the search for an appropriate object and backtracking if constraints are not fulfilled. The overall approach allows snapshot generation taking into account invariants but also requires the explicit encoding of constraints in generation commands. As such, the problem tackled by automatic snapshot generation is different from the meta model to graph grammar translation.

Formal methods such as Alloy [All00] can also be used for instance generation: After translating a class diagram to Alloy one can use the instance generation within Alloy to generate an instance or to show that no instances exist. This instance generation relies on the use of SAT solvers and can also enumerate all possible instances. In contrast to such an approach, our approach aims at the construction of a grammar for the metamodel and thus establishes a bridge between metamodel-based and grammar-based definition of visual languages.

6 Conclusion and Future Work

In this paper we have presented the main concepts for translating a meta model to an instance generating graph grammar. This translation has been illustrated by a simple statecharts example. The meta classes and associations as well as their multiplicities are translated directly to type graph, start graph and graph rules. To handle also the OCL constraints during the instance generation process, they are first translated to graph constraints and then partly to application conditions of rules. In this paper, we discussed this translation process for restricted OCL constraints. In future work, OCL constraints and graph constraints have to be further compared concerning their expressiveness. Moreover, we started to give OCL a new kind of semantics which has to be set into relation with other OCL semantics.

Automatic derivation of instances from meta models is a complex task which needs tool support. So far, we have automated the construction of an instance generating graph grammar by providing a model transformation that automatically derives an instance generating graph grammar from a meta

model. For a complete description of this implementation we refer to the URL http://tfs.cs.tu-berlin.de/agg/MM2GraGra. Although the current model transformation does not support all features of meta models yet, it nevertheless shows the feasibility of our approach. We plan to investigate the usage of our techniques for systematic testing of model transformations at different development stages of our generation algorithm.

References

[All00] *The Alloy Analyzer - 3.0 Beta* http://alloy.mit.edu/, 2000.

[BEdLT04] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, *Proc. Fundamental Aspects of Software Engineering 2004*, volume 2984. Springer LNCS, 2004.

[BS97] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[EEdL+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 214–228. Springer Verlag, 2005.

[EEHP04] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, LNCS 3256, pages 287–303, Rome, Italy, October 2004. Springer.

[EKTW05] K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Automatically Generating Instances of Meta Models based on a Graph Grammar Approach. Technical Report 2005–09, Technical University of Berlin, Dept. of Computer Science, November 2005. To appear.

[GBR05] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling*, 2005. To appear.

[Obj05] Object Management Group (OMG). *OCL 2.0 Specification. OMG document ptc/2005-06-06*, June 2005.

[UML03] Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02*, August 2003.

On Challenges for a Graphical Transformation Notation and the UMLX Approach

Edward D. Willink [1]

GMT Consortium
www.eclipse.org [2]

Abstract

Freely available experimental transformation languages have begun to stimulate practical usage of textual transformation notations. The forthcoming QVT transformation languages may provide standardisation or at least interchange capabilities for these experimental languages. Graphical transformation notations are proving rather less successful. We identify many disadvantages of the graphical approach, consider how they can be circumvented and describe changes in the UMLX notation and tool support to improve usability and QVT compatibility.

Key words: Model Transformation, Transformation Editor, Graphical Transformation Notation.

1 Introduction

Model transformations are becoming steadily more practical and rigorous with the advent of better meta-model based tools such as ATL [7] or Tefkat [6]. The transformation language for each of these tools is proprietary and so inhibits the wider exploitation of these transformations. The increasingly imminent QVT standard [8] for a suite of three languages may avoid incompatibility problems, possibly by rendering the existing languages obsolete, more likely by defining an interchange point so that transformations for language A may be transformed to a QVT language and from there to language B.

The current transformation languages and the QVT submission are largely textual in syntax, which is perhaps surprising given that they operate on meta-models that are often drawn using a graphical style derived from UML.

A graphical notation can be visually attractive and this provides a significant advantage over a textual notation. An interesting picture may provoke

[1] Email: EdWillink@iee.org

[2] The author is not associated with the QVT-merge consortium of which his employer, Thales Research and Technology (UK) Ltd, Reading, England is indirectly a part.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

fruitful discussion within a team or with passers by. Program text often has a more restricted readership.

In this paper we try to understand why graphical transformation notations are proving less successful and consider when and how this lack of success should be remedied. We first consider the generic advantages and disadvantages of textual and graphical notations, before examining the reasons why some graphical notations have proved successful and others have not. In Section 2 we examine a variety of graphical transformation notations in greater detail, noting their similarities with respect to the use of declarative patterns in principle, but significant differences in practice and even greater differences with respect to transformation rules. Then in Section 3 we outline the tool support for UMLX and notational enhancements to improve alignment with QVT.

1.1 Textual and Graphical Notations and Tools

A textual notation for a language has many advantages over a graphical notation. Tooling requirements are limited and so a wide variety of editors can be used. Printing or viewing is straightforward and activities such as searching or comparing pose few challenges. Even when more sophisticated syntax-sensitive editing is desired, there are a number of customisable editors to choose from.

A textual notation does not suffer particularly from scalability issues. The use of multiple source files and hierarchical language constructs enables very large overall line counts to be managed. Unduly long lines are readily avoided at the expense of a slightly increased line count.

In contrast, a graphical notation has many disadvantages. Specialised editors are required often using file formats with limited scope for interchange with alternative editors. Ancillary tasks such as printing require similarly specialised tooling that consequently must form part of the editor. The extra pagination and rendering complexities are not always satisfactorily realised for an adequate range of print media. Other activities, such as searching also rely on the specialised editor. With so much reliance on a specialised editor, the choice of editors is limited, and very limited if significant financial investment is to be avoided.

Graphical notations may also suffer from scalability; there is a limit to the amount of zooming that can be tolerated to enable a diagram to fit on a single sheet of paper. This is a major problem when the notation fails to represent a reasonably sized and sensibly modularised problem on a single sheet, and a minor problem if navigation between diagrams is cumbersome.

1.2 Successful Graphical Notations

A graphical notation must offer significant benefits to overcome these major disadvantages, so it is worth reviewing where and why graphical notations are

successful.

A textual language often aligns the vertical dimension with a sequential order, so that earlier lines precede later lines in execution order, or spatial layout. This suits the procedural style of programming that is so widespread through the dominance of languages such as C or Java. A critical weakness in textual languages arises when a more declarative perspective is taken. It is difficult to identify all uses of a particular concept such as a variable, so it is necessary to search the code for all occurrences of a name.

A graphical notation imposes no inherent layout constraints. This freedom allows some very poor diagrams to be drawn when little attention is paid to ease of understanding. This does not matter for diagrams drawn on the 'back of an envelope', or a whiteboard, where the audience is live and the sole purpose of the diagram is to clarify understanding or intention. These diagrams should then be destroyed. Diagrams that are to be preserved should be drawn with much greater discipline, so that a cold audience can easily grasp their meaning. It is often helpful to observe some form of left to right and/or top to bottom discipline to ease comprehension by providing the audience with a starting point. A few well drawn diagrams may provoke useful discussions about a design approach, whereas textual representations are less amenable to casual review.

The lack of an inherent layout makes graphics well suited to a declarative exposition of many interrelated concepts. Each concept is a graph node denoted by a symbol, and each interrelationship is a graph edge denoted by a line. The lines between symbols are easy to identify and so all relationships involving a particular node are easily determined.

With this insight, it is not surprising that State Machines have been so successful graphically, since the many peer states can be shown with equal or weighted import. The presence and absence of transitions between states is very evident. For similar reasons, Entity Relationship Diagrams or Class Diagrams are also useful to enable the structural relationship between diverse data elements to be visualised.

Graphical notations for functional and dynamic relationships are often less successful. Program Flow Charts were popular in 1970s but are now little used. The Schlaer-Mellor [9] notation for a Data Flow Diagram involves lines for flows between processes and stores; this notation has insufficient precision to be used as more than an overview. On the other hand, the SDL [4] notation has a rich set of symbols for expressing sequential computations. This notation can have excessive precision; the graphics appears more complicated than comparable text.

A variety of forms of Block Diagram have been used for decades in the Electronics industry. Many of these diagrams have insufficient precision for code generation but could be given that precision once adequate, quite possibly MDA-related, tools mature. These diagrams are a declarative exposition of a variety of potentially concurrent activities with some similarities to the revised

Activity Diagram in UML2.

The above selection of usages from a variety of fields demonstrates that graphical notations succeed for declarative problems but fail for procedural problems. Transformations based on UML-like meta-models might be expected to continue the graphical tradition of class diagrams and favour a graphical notation. However the extended functionality for imperative transformation languages is procedural, an area where UML proves less satisfactory.

1.3 Utility

In addition to the handicaps that may arise from the difficulties of providing a suitable graphical presentation, further difficulties may arise from a poor notation.

If a notation lacks obvious meaning, the utility of the notation as a stimulus for informal review and discussion may be impaired, and potential users may be discouraged by a need for extensive training. For transformation authors, the learning cost may be ameliorated by good tool support, but for casual reviewers the notation must be as obvious as possible.

A new notation should offer obvious and genuine advantages over alternatives. Graphical notations can have a clear advantage in appearing at least superficially attractive and may back this up by providing compact notation and easy to use tooling. But genuine advantages require a notation to have as much rigour as possible, so that as many different forms of error are eliminated by the design, thereby improving the productivity of the notation. Where possible, this rigour should be hidden from practical programmers who may be discouraged by the too overt appearance of, for example, set theory. Although, with OCL 2 becoming an essential part of so many transformation approaches, it may be that programmers will be forced to extend their mathematical background.

1.4 Conclusions

From the above we may conclude that a graphical notation can be superior when it provides a declarative exposition. To satisfy the potential for providing diagrams for casual review, the notation should be clear, precise, compact and easy to understand, yet rich enough to express realistic problems.

A graphical notation should provide sufficient precision to ensure that diagrams are useful, but should recognise that not everything is sensibly presented graphically. The notation must therefore co-exist with a textual form so that users have a free choice to use whichever of textual or graphical expositions is clearest.

The scalability issues for a graphical notation should be addressed by ensuring that a diagram can be modularised sufficiently to fit on a page and in the tool support by providing good navigation between pages.

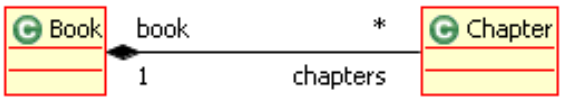


Fig. 1. UMLX Composition Definition.

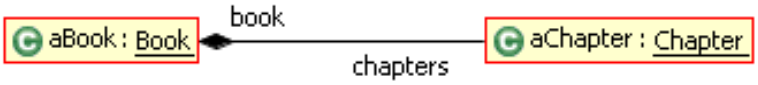


Fig. 2. UMLX Composition Instance.

The underlying principles behind the notation should be rigorous to assist in provision of portable, reliable, maintainable and re-usable transformations.

The notation should be easy to learn, fun to use and well supported by tools to provide a pleasant programming environment. Support by free tools may serve to increase the user base more rapidly.

2 Graphical Transformation Notation

2.1 Relationships and Instances

It is reasonable to assume that anyone using meta-model transformations is familiar with UML, and so basic UML concepts such as composition require less learning than their textual counterpart.

Figure 1[3] shows a simple composition in which a `Book` may contain zero or more `Chapters`. A `Chapter` must be contained by exactly one `Book`, that may be accessed as the `book` property of the chapter.

UML practitioners might also recognise Figure 2 that shows a similar relationship between two objects rather than two classes. A particular instance of a `Book` is identified as `aBook` and contains an instance of a `Chapter` identified as `aChapter`.

In traditional UML usage, `aBook` and `aChapter` are instances whose identification facilitates exposition of the remainder of the UML design; there is a single {`aBook`, `aChapter`} tuple for the whole design.

In model transformation usage, the object diagram is re-interpreted to define a pattern, so that wherever the pattern is satisfied, an appropriate transformation rule can be activated. `aBook` and `aChapter` are therefore variables within the pattern that bind to instances in the model to be transformed; there is a distinct {`aBook`, `aChapter`} tuple for each possible rule activation for the transformation.

The relationship between `aBook` and `aChapter` is not shown in a textual transformation language; it is implicit in the declaration of two variables with

[3] The diagrams in this paper are drawn using UMLX, which is based on and follows the Eclipse/GEF EDiagram example in adding an icon at the top left of symbols.

types from the meta-model comprising `Book` and `Chapter`. This implicit relationship avoids typing, but inhibits casual review by readers who are unfamiliar with the meta-model and may lead to obscure error messages for programmers who misunderstand it.

2.2 Presentation

Although graphical transformation notations have adopted the style of Figure 2 there has been some divergence in their presentation and elaboration.

In the revised merged QVT submission [8], the class name underlines and the line decorations are omitted. Omission of the underline is a minor stylistic deviation from UML. Omission of the line decorations deprives the reader of the distinction between composition and association and the disambiguation of multiple associations involving the same classes. AGG [3] uses a more conventional Graph Transformation notation and so underlines and line decorations are again omitted, and instance names are replaced by instance numbers.

Gmorph [10] and GRE [2] are much closer to UMLX [12]. Gmorph underlines both instance and class name while GRE uses a stereotype notation [4] for the class name.

2.3 Multiplicity

UML provides three relevant decorations for relationships; the multiplicity, the property name, and the diamond for composition. The last two of these can assist in understanding the correspondence between pattern and meta-model. Re-use of multiplicity in this purpose would be confusing, and so some graphical notations such as the QVT submission just omit the multiplicity. Figure 2 therefore denotes the relationship between a single `aBook` and a single `aChapter`.

GRE showed multiplicity in the style of UML but did not exploit it. Its successor, GReAT, recognised that non-unit multiplicity could support patterns involving sets of objects rather than just objects. The corollaries of this interpretation are discussed in [1]; a pattern could now specify that its rule was applicable only to each book containing two or more chapters, rather than just to each book (that might contain some chapters). However, in GReAT, the multiplicity was shown as an instance stereotype thereby identifying the absolute size of the set of matched objects for each free variable, rather than the relative size of the set of matched objects between the ends of the decorated relationship.

UMLX followed GRE in using the UML presentational style for multiplicity and used the UML multiobject notation where a set of objects rather than just an object was bound to a variable. UMLX followed the disciplined principles of GReAT to define the meanings of these sets. UMLX further recognised

[4] Text between angle brackets.

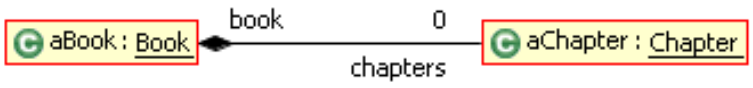


Fig. 3. UMLX Composition Instance With Zero Multiplicity.

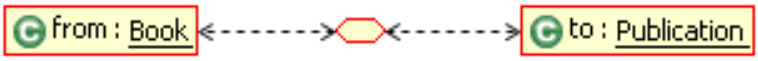


Fig. 4. UMLX Evolution.

that a zero multiplicity signified negation, so that a simple zero as in Figure 3 denotes a pattern that matches each `Book` that contains no `Chapters`. Other notations have introduced a distinct crossing-out symbol for this purpose.

The QVT submission also exploits the UML multiobject to display sets of objects, however the submission has limited capabilities for patterns that involve sets of objects. UMLX has a more comprehensive capability defined in [11]. The GReAT analysis of multiplicity identifies some limitations in its use. For example, in a complex pattern in which there is a cycle, interpretation of the cycle clockwise may lead to a different meaning to the anticlockwise interpretation. This is clearly unacceptable and must be reported as an error. The problem is analogous to the need for parentheses to impose an intended meaning on an arbitrary expression involving 'and' and 'not' operators. The graphical equivalent of a parenthesis requires an explicit grouping of part of the cycle so that there is a single relationship between the residual part of the cycle and the grouped part. Provision and evaluation of this grouping as a graphical facility is work in progress for UMLX.

2.4 Rules

The principle of using patterns to define the application context of a transformation rule is common to perhaps all graphical transformation approaches. The divergence between approaches arises in the relation between input and output context.

GRE showed separate input and output patterns with creation relationships between them.

UMLX was inspired by GRE but replaced the creation relationship by declarative preservation and evolution relationships. A preservation relationship keeps the input element for re-use on as an output element. An evolution relationship may add an output element or elements with respect to an input element or elements, and may also delete an input element or elements with respect to an output element or elements. Preservation extends the Keep operation of Graph Theory [5], to support keeping not just a node, but also all its composed descendants. Evolution combines Add and Delete operations in a multi-directional relationship that always defines a traceability relationship.

GReAT abandoned the potentially declarative characteristics of GRE to

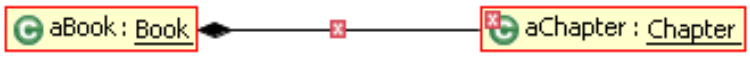


Fig. 5. UMLX Composition Instance With Errors.

pursue an imperative approach. This achieves layout economy by overlaying shared input and objects but the use of colours to distinguish input-only, output-only and shared objects lacks intuition and cannot be rendered in black and white. Imperative operators are graphically sequenced to define the transformation. The result is an unfortunate mix of imperative and declarative meaning in the same diagram.

The proposed graphical syntax for QVT makes no attempt to relate input and output patterns visually; each is drawn independently, with the relationship between them defined in a textual region by OCL expressions involving the names of input and output objects.

Gmorph also relies on text to associate independent input and output graphics. AGG uses shared instance numbers to associate independent input and output graphs.

3 UMLX Tooling

We have already described how UMLX attempts to align with the goal of a providing a declarative and easy to understand notation by re-using the familiar UML notation for defining patterns, extended semantically by a solid definition of the meaning of multiplicity and sets of objects. A declarative graphical extension supports definition of the transformation between the patterns with the need to resort to text.

In order to fulfil the ease of use criterion, the original implementation of UMLX based on GME and inflexible Microsoft technologies, has been replaced by Eclipse plug-ins based upon GEF and more particularly its EDiagram example that uses EMF for meta-models. This provides a free, portable and standard integration platform. Sadly, it exchanges GME's disciplined meta-modelled approach to defining a graphics editor by the much more flexible but rather manual code cutting capabilities of GEF, EMF and Java. It is hoped that GMF may provide the best of both worlds for the next revision.

The Eclipse support for UMLX exploits an Outline view of one or more meta-models to provide a dynamic palette of Drag and Drop elements that can be dropped onto a sheet to instantiate, or onto a graphical element to re-instantiate, a legal design element. Widespread use of the Outline, Drag and Drop, in-place editing and standard GEF editing interactions enables designs to be built up with considerable ease.

An editor supporting graphical transformations should assist the user in drawing transformations that comply with their meta-model, so the decorations are supplied automatically whenever an unambiguous relationship is

drawn between two classes. When an illegal relationship is drawn, the problem is shown as in Figure 5 which shows the impact on Figure 2 following deletion of `Chapter` and the `Book` to `Chapter` relationship from the meta-model. The pattern is still syntactically correct but because the meta-model has been changed, the pattern is no longer semantically valid. A similar problem arises in textual languages when a subroutine call is invalidated by a change to its signature. Each inappropriate graphical element shows a red Eclipse error marker. These also appear in the Eclipse Problem, Outline and Resource views, so that the graphical markers behave in a similar way to textual error markers in the Java editor.

The editor supports partitioning a design into sheets using three different diagram types. Meta-Model Diagrams support maintenance of Ecore meta-models, that are instantiated within Transformation Rule Diagrams where the UMLX transformations are drawn. A further Transformation Context Diagram supports aggregating many UMLX-defined or QVT-defined rules as part of a QVT compatible Transformation.

4 Summary

The relative characteristics of textual and graphical notations have been contrasted first from a relatively generic point of view and then by contrasting a number of different graphical transformation notations.

The many advantages of a textual notation suggest that a graphical notation is most likely to be superior when it uses a declarative style that is easily understood by casual reviewers. With a graphical notation that is superior in some contexts, a transformation programmer may then be offered a choice of notations so that the most appropriate can be chosen for each context.

UMLX complies with these declarative perspectives. It is hoped that the revised Eclipse-based tooling available from http://www.eclipse.org/gmt will provide a friendly easy to use and productive environment that will encourage the use of declarative transformations.

References

[1] Agrawal, A., Levendovszky, T., Sprinkler, J., Shi, F., Karsai, G.: Generative Programming via Graph Transformations in the Model-Driven Architecture, OOPSLA 2002 Workshop on Generative Techniques in the context of Model Driven Architecture, November 2002 URL: http://www.softmetaware.com/oopsla2002/karsaig.pdf

[2] Agrawal, A., Karsai, G., Shi, F.: A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations, URL: http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_0_0_2003_A_UML_base.pdf

[3] Buttner, F., and Gogolla, M.: Realizing UML Metamodel Transformations with AGG In: Proceedings of the 4th International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT 2004, Barcelona, Spain, March 2004, URL: http://wwwcs.uni-paderborn.de/cs/ag-engels/GT-VMT04/GT-VMT04-camera-ready.zip

[4] CCITT Recommendation Z.100: Specification and Description Language SDL, Annexes A-F to Z.100. 1988. Blue Book, Volume VI.20 - VI.24, ITU, General Secretariat- Sales Section, Places des Nations, CH-1211 Geneva 20.

[5] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach, In Rozenberg, G., ed., The Handbook of Graph Grammars, Volume 1, Foundations, World Scientific, 1996.

[6] DSTC QVT Team: Tefkat, URL: http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/

[7] Jouault, F., and Kurtev, I.: Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, October 2005, URL: http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev__transforming_models_with_atl.pdf

[8] Revised submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10), OMG Document, ad/2005-03-02, URL: http://www.omg.org/docs/ad/05-03-02.pdf

[9] Schlaer, S., and Mellor, S.: Object-Oriented Systems Analysis: Modeling the World in Data, Yourdon Press, 1988.

[10] Sendall, S.: Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?, OOPSLA 03 Workshop Generative techniques in the context of MDA, 2003 URL: http://cui.unige.ch/~sendall/files/sendall-mda-workshop-OOPSLA03.pdf

[11] Willink, E.: Towards a Formalization of UMLX, URL: http://dev.eclipse.org/viewcvs/indextech.cgi/*checkout*/gmt-home/subprojects/UMLX/doc/UmlxFormalization/UmlxFormalization.pdf

[12] Willink, E.: UMLX : A Graphical Transformation Language for MDA, URL: http://dev.eclipse.org/viewcvs/indextech.cgi/*checkout*/gmt-home/doc/umlx/Oopsla2003.pdf

View Creation of Meta Models by Using Modified Triple Graph Grammars

Johannes Jakob [1] Andy Schürr [2]

Real-Time Systems Lab
Darmstadt University of Technology
Darmstadt, Germany

Abstract

Model-based software development is a hot topic of the software engineering community. Most activities in this area including the standardization efforts of the OMG are targeted towards the development of meta modeling tools, adaptable code generators, and model transformations tools. The needs for the specification of model views that simplify the definition of model transformation, abstract from details of specific modeling languages and tools or support the adaptation of generic modeling approaches to a specific domain are usually out of scope. This paper presents, therefore, a unified approach for the declarative definition of updatable model views. New interpretations of the well-known concept of triple graph grammars are used for that purpose which support, for the first time, the construction of non-materialized views. The adaptation of the presented approach to the world of the Model Driven Application development standards of the OMG and the recently finalized model transformation language QVT is under development.

Key words: view creation, meta modeling, triple graph grammars

1 Introduction

Any system engineering process requires the manipulation of development artifacts at various levels of abstraction. Keeping all these artifacts and their traceability relationships in a consistent state often turns out to be a nightmare. This is especially true for model-based software engineering, where often many modeling tools are used in parallel, e.g. for requirements elicitation purposes, safety and security analysis, and software design. Today available model integration and transformation approaches offer semi-automatic support for

[1] Email: Johannes.Jakob@es.tu-darmstadt.de
[2] Email: Andy.Schuerr@es.tu-darmstadt.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

preserving the consistency of the data of these tools. They start with the definition of meta models for the regarded modeling languages on a specific level of abstraction and they add rules for constraint checking and update propagation purposes between instances of different meta models. The specification of different layers of abstraction as *meta model views* is usually out of scope. Future meta modeling and model transformation languages should offer better support for the views construction purposes for the following reasons:

- So-called *view points* are a popular software engineering concept that allows one to look at the same integrated model from different perspectives. These view-points are just a special case of views that abstract from and hide irrelevant details of the underlying model.
- Domain-specific or even *project-specific modeling approaches* are either based on meta-case tool technology that excludes the usage of standard modeling tools or resort to the implementation of specific wrappers and add-ons on top of standard tools. In some cases these wrappers are a special kind of (meta) model view.
- Furthermore, it is often necessary to decouple the implementation of model checking and model transformation tools from *vendor-specific tool interfaces*. Again views are a standard technology to standardize the modeling concepts and interfaces of a family of tools for the same domain and to simplify the replacement of a specific tool.

These were our motivations for a new line of research that uses the declarative model transformation approach of triple graph grammars (*TGGs*) as a starting point. A new variant of *TGGs*, so-called *VTGGs*, are introduced for view specification purposes. Combining *TGGs* and *VTGGs* results in a *unified meta-model-based view definition and model transformation approach*.

The rest of this paper introduces *VTGGs* and is structured as follows: In Section 2 an example of two interdependent meta models is introduced, where one meta model plays the role of a view onto the other one. A short discussion of related work concerning the construction of (database) views is presented in Section 3. The following Section 4 introduces *VTGGs*, a modified variant of *TGGs* used for view specification purposes. Section 5 concludes this paper, discusses open issues, and future work. Please note that a detailed presentation of the translation process of *VTGG* rules into view implementing graph transformation rules had to be omitted due to lack of space.

2 Running Example

This section introduces the running example that is used throughout the rest of the paper. The overall scenario we have in mind is related to a model-based software development process (Fig. 1a). A software engineer uses two different COTS (commercial of the shelf) tools for requirements elicitation and software design purposes (like DOORS and Matlab/Simulink). Initially, these tools are

not related to each other and offer rather generic means as well as APIs for any kind of software engineering project. Engineers using these tools are faced with the following problems: they have no means to abstract from the details of a specific tool's API, and they are usually running into problems when domain-specific development data integration and transformation rules have to be specified. Not only for the last purpose, model transformation languages are needed that are able to operate on top of model views. In more detail, a view definition approach is needed that supports

- definition of multiple overlapping views for a single model,
- update propagation from views to models and vice versa,
- synchroneous and asynchroneous propagation of changes,
- manipulation of virtual (non-materialized) views,
- high-level declarative specification of views,
- model-to-view mappings with complex restructuring operations.

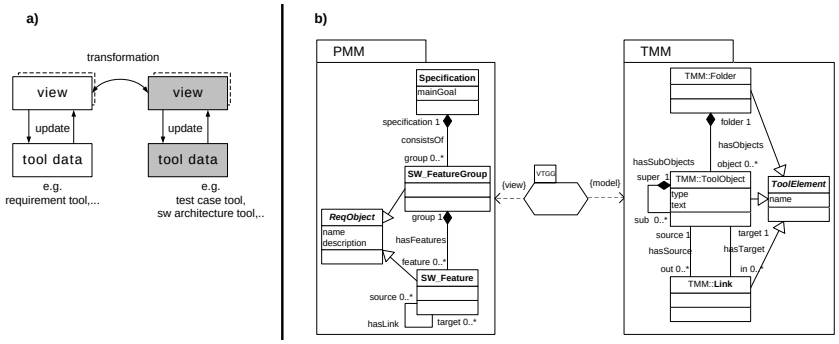


Fig. 1. a) Range of use of view creation, b) VTGG meta model (schema)

In the sequel, we will first present the example of a project-specific meta model (view) definition (PMM) together with its related tool-specific meta model (TMM) using MOF 2.0 as a meta modeling language (cf. Fig. 1b). The PMM defines some basic concepts for a feature-oriented requirements engineering process, whereas the TMM reflects to a limited extent the data models of general purpose requirements engineering tools like DOORS. In principle, specialization is only with the usage of non-derivable abstract superclasses possible. Due to lack of space, the correspondences are simplified to the VTGG package between both meta models.

2.1 Project Meta Model

Fig. 1b) (PMM package) shows the project-specific meta model of our requirements engineering example. The class `Specification` represents the root of a two-level hierarchy of features of a regarded software product. Each project has only one `Specification` instance (a project is represented by the PMM). In addition, the `Specification` has an attribute `mainGoal`, a

text block which explains the most important goals of the software development project. Furthermore, a specification may have an arbitrary number of `SW_FeatureGroup` (software feature group) elements. For the purpose of this paper we present only one kind of feature group and omit, e.g., a distinction between system, hardware, and software features. The `SW_FeatureGroup` as well as `SW_Feature` inherit the attributes `name` and `description` from the abstract class `ReqObject`. The attribute `name` introduces a short identifier, the attribute `description` a text block with a more detailed explanation of the regarded requirement. Moreover, `SW_FeatureGroup` is a structural element that is used as a container for a group of related software describing features `SW_Feature` (cf. association `hasFeatures`). Each `SW_Feature` belongs to only one `SW_FeatureGroup`. Finally, the association `hasLink`) is used to link related features to each other.

2.2 Tool Meta Model

The tool-specific meta model depicted in Fig. 1b) (TMM package) represents a cut-out of the data structure of a typical requirements engineering CASE tool. The class `folder` is the top-level structural element of this tool. A `Folder` contains a set of `ToolObject`s (association `hasObjects`). Any `ToolObject` possesses a `type` attribute as well as a `text` block attribute. Additionally, it contains a set of `ToolObject` instances in turn (cf. association `hasSubObjects`). The `type` attribute is used to clearly distinguish different sorts of requirements objects stored in the regarded tool; text fragments of (almost) arbitrary length are assigned to `text` attributes. Furthermore, two `ToolObject`s may point to each other via associations to separate `Link` class instances (cf. associations `hasSource, hasTarget`). All introduced classes of the TMM inherit the attribute `name` from the abstract class `ToolElement`.

3 Related Work

In the previous section we have already outlined our requirements for a model transformation approach that supports definition and manipulation of model views. When we are looking for tools that offer this kind of support we have to distinguish two different categories: meta-case tools like Pounamu [17] or MetaEdit+ [11] mainly use the term "model view" as a short-hand for "visualization of a model" (Model-View-Controller design pattern). What we have in mind in this paper is something different: logical model views in the sense of the database community that are again models and not just visualizations of models. A majority of the afore-mentioned tools offers rather specific support for the visualization of models, but no support for the definition of logical views. Approaches like MViews [5] or view transformations for AHL nets [3] are borderline cases. They support the definition of logical views, but presented examples deal with visualizations of models only.

To the best of our knowledge no meta-case tool or model transformation approach fulfills our model view definition requirements. This is especially true for OMG's QVT (Query, View, and Transformation) [1] language standard which excludes in its current version explicitly any support for the definition of views. Of course, one may argue that a view of a model is just another model which is kept consistent with its underlying model using standard model transformation techniques. ATOM3 is a prominent example of a meta modeling tool that favors this approach [6]. ATOM3 adopted the proposal made in [13] and uses a special form of triple graph grammars (TGGs) for the declarative definition of updatable views. These views are mainly used for visualization purposes and complex restructuring operations like mapping of view associations onto model objects (or vice versa) are not yet supported.

The main drawback of all view definition solutions based on unmodified model transformation techniques is that we have to *materialize* all views instead of using more light-weight software engineering concepts for the construction of abstraction layers. What we would like to achieve is that views on top of extensionally defined models are implemented as functional API layers. Therefore, we are looking for an approach that supports the declarative specification of (meta) model views plus the automatic generation of "light-weight" view API implementations based on standard adapter design patterns.

The view definition approaches presented in the database management literature suffer from similar drawbacks. Relational database management systems like Oracle offer very limited support only for the definition of updatable views; traditionally updatable views are restricted to projection of columns and selection of rows of an underlying base table. More sophisticated data integration approaches for data warehouses or federated database systems like AMOS-II [16] often limit their support to queries that are translated and propagated using so-called mediator concepts. And even database management systems like SBQL [9] with their advanced versions of "instead of trigger" concepts rely on rather low-level procedural implementations of the translation of basic query and update operations on views into queries and updates of the underlying databases. The most interesting view definition concepts have been recently added to federated P2P DBMS. They rely on bidirectional schema transformations and maybe used to keep a set of databases with a set of materialized derived views in a consistent state [10]. The basic constructs of these schema transformations are comparable to the low-level operational model manipulation constructs of QVT; higher level specification concepts are not yet available.

To summarize, we are not aware of any higher-level languages and tools that offer support for an integrated specification of model transformations and non-materialized updatable views comparable to the modified triple graph grammar concept presented here.

4 Triple Graph Grammars

This section describes the basics of our unified model view definition approach based on *triple graph grammars (TGGs)*. Since the introduction of TGGs in [14], quite a number of modifications and extension have been published [2,7,8]. The model view definition approach presented here is based on the model transformation extensions of [7], where TGGs have been combined with MOF 2.0. Thus, the meta models of our running example are also MOF 2.0 compliant (Fig. 1b), i.e. MOF 2.0 plays the role of a graph schema definition language for TGGs.

In principle, a triple graph grammar is a regular graph grammar with the empty graph as the axiom and a set of graph grammar rules. These rules generate a language of graphs or, more precisely, the set of all consistent graphs which is a subset of the set of all schema-compliant graphs. The interesting point of a TGG is the fact that specified rules consist of three subrules and generated graphs consist of three related subgraphs. Two components always represent a pair of related graphs and the third component introduces traceability relationships between the regarded pair of graphs. Furthermore, TGGs are used as input for a transformation process that yields a set of regular so-called *operational graph transformation rules* tailored for a specific purpose like "forward" model transformation or "backward" propagation of changes.

In the following, we will introduce a new variant of TGGs, called *VTGGs* for model view definition purposes. VTGGs introduce a new set of restrictions for their rules combined with a new way how to translate TGG rules into regular graph transformations rules.

The following rules demonstrate how to use TGGs for view definition purposes and what modifications have been required for that purpose. The rules are depicted within the scope of the introduced example in chapter 2. The mapping relationships from virtually existing PMM objects to really existing TMM objects are modeled as links between *objects* combined with the tag `{map}` (correspondence node). In all cases elements on the left side of a rule belong to the virtually existing *view* computed from the really existing elements on the right of side of a regarded rule. According to the approach of [8], the creation of new objects or links are denotated with the `{new}` tag. Invariant constraints are denotated as OCL constraints.

4.1 Creating the Initial Model and View Elements

This subsection introduces the initial VTGG rule that matches the empty axiom graph and creates the top-level object of the underlying model and its view. The application of a *TGG* rule to a pair of related model graphs is based on finding matches for all untagged elements of the rule. In the initial state of the derivation of a model and its view there are no objects which could be matched by any pattern.

Fig. 2a shows a simplified version of the first rule of our example, which

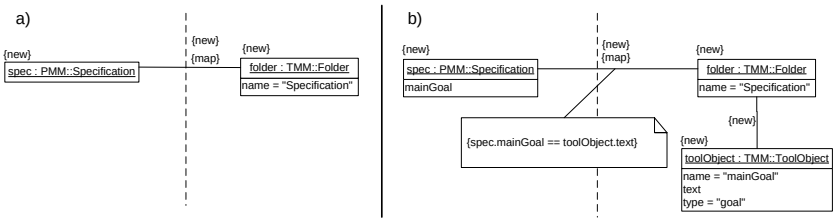


Fig. 2. Mapping of classes, attributes and attributes to classes

is extracted form the rule in Fig. 2b. It specifies that a new `Folder` instance is really created, when we want to create a new `Specification` instance. For this purpose both the `Specification` instance on the PMM side and the `Folder` instance on the TMM side carry the tag `{new}`. All view objects creating VTGG rules have a similar form. A single instance of a new virtual *view* object is mapped onto a single instance of a new real object (often called "seed object" in the DBMS literature). This seed object maybe linked to an arbitrary number of new additional helper objects in the general case. The class name `Specification` (modeled in the PMM 1b) is assigned to the attribute `name` in order to identify the folder as a `Specification`.

More generally spoken, this view shows the simplest form of a VTGG rule, an object to object mapping. Furthermore, the rule shows the simplest case of handling the attributes of virtual (view) and real objects. In contradiction to the meta model of Fig. 1b we do assume that `Specification` instances do not possess any attributes and that they are translated into `Folder` objects with the value `"Specification"` assigned to their attribute `name`. More complex relationships between attribute values of view and real objects are explained later on.

4.2 Creating Isolated Model and View Objects

The following rule shows in addition to the rule above, how to map a virtual attributed *view* object onto a set of real attributed objects.

Fig. 2b displays the entire initial VTGG rule for our running example. Additional to the rule fragment described above (Fig. 2a), this rule maps the attribute `mainGoal` of class `Specification` to an own object `toolObject` on TMM side. Furthermore, the two attributes `name` and `type` are set to constant values for identification purposes. The OCL constraint annotation of the `{map}` relation specifies that the value of the attribute `mainGoal` is mapped to the `toolObject`s attribute `text`. For navigation purposes as modeled in the TMM, a link between the `folder` and `toolObject` is created. Of course, there is no corresponding link on PMM side.

4.3 Creating Context-Dependent Model and View Objeccts

This subsection describes a VTGG rule for creating new objects that are linked to already existing objects.

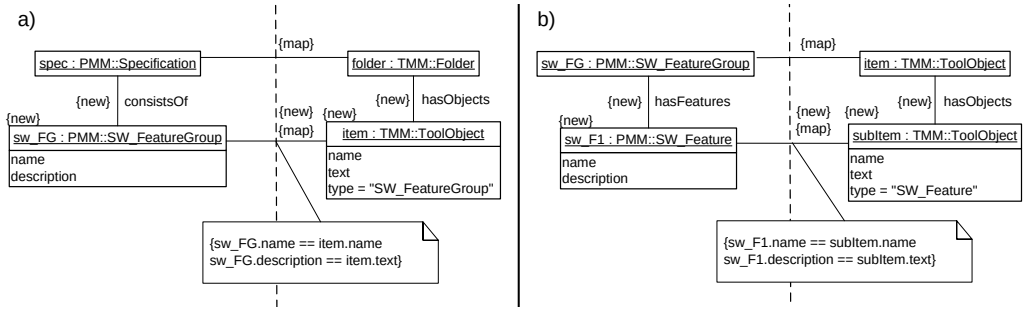


Fig. 3. Mapping of associated classes

In Fig. 3a the new object `sw_FG`, which represents an instance of the class `SW_FeatureGroupe`, is created with an additional new link to the already existing `Specification`. Also `sw_FG` is mapped to a `ToolObject` instance `item` as a *view*, like described in the section before. For identification purposes we assign the constant `"SW_FeatureGroup"` to the attribute `type`. Furthermore, the OCL constraint of the rule states that `sw_FG.name` is mapped onto `item.name` and that `sw_FG.description` is mapped onto `item.text`. In addition, the association of the created `sw_FG` instance with the context object `spec` corresponds to the association of the created `item` instance with the context object `folder`. In the same manner the rule in Fig. 3b is modeled. The creation of a new instance of `SW_Feature` corresponds to the creation of a `ToolObject` instance, `subItem`. Also the link will be created like described before. The second rule represents the association of `ToolObject` to itself as shown by the TMM (Fig. 1b).

4.4 Mapping of Associations

As an enhancement of the hitherto existing TGGs (dealing with object mappings only), this subsection describes a new type of rules that translates a virtual association between two objects into an arbitrarily complex substructure of the underlying model graph.

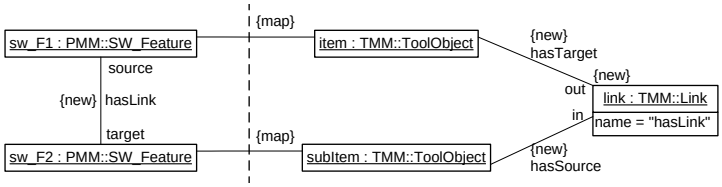


Fig. 4. Mapping of an association to a class

As depicted in Fig. 4, the new link `hasLink` between the two "existing" instances of `SW_Feature` corresponds to the new instance of the class `Link` together with its two associated links `hasTarget` and `hasSource`. These links are created at the same time as the new `Link` object and establish the needed associations to the regarded `ToolObject` instances. This rule can be used for creating cross-reference relationships between already existing related re-

quirement instances. In order to be able to distinguish different kinds of cross-references the class `Link` has the attribute `name` (parallel links are not possible). The association name `"hasLink"` as a string, is assigned to this attribute.

4.5 Interpretations of VTGGs

One goal of using VTGGs is the automatic translation of VTGG rules into executable Java code. The translation process may associate quite different operational semantics with a VTGG as follows: in a first step we translate a VTGG graph schema into a regular graph schema (MOF 2.0 meta model) and the set of VTGG rules into a set of regular graph transformation rules. Different translations are under development for maintenance of fully materialized views (as described in [de Lara])"light-weight" views implemented by a layer of adapter objects (object adapter pattern [4] p. 141), and purely virtual views which reuse model objects as adapter objects and which do not create any additional objects or links at all (class adapter pattern [4] p. 141). The generated rules are then compiled into Java code using the standard code generator of the graph transformation tool Fujaba [15]. This flexibility of (V)TGGs in general is one of the main advantages of the presented view specification approach.

5 Conclusion

In this paper we have introduced a modified version of triple graph grammars for view specification purposes. These VTGGs differ from regular TGGs in four ways:

- The view defining side (subrule) of a VTGG rule consists of a single new object (also with links to already existing objects) or link only; this single intensionally defined view element is mapped onto an arbitrarily complex substructure of the extensionally defined underlying model.
- TGG rules used for model integration purposes always add at least one object to each regarded model, whereas VTGG rules for associations add a single link to the model view only.
- The translation of a VTGG into a regular Fujaba graph transformation system does not preserve the involved meta models, but creates a new meta model by weaving the class hierarchies of the input meta models. (not presented here due to lack of space)
- Graph transformation rules generated from VTGG rules do not manipulate two related model instances, but translate read and write operations on the virtually existing model view into read and write operations of the actually existing underlying model. (not presented here due to lack of space)

Precise definitions of different variants of VTGGs are under development as well as an implementation of the (V)TGG approach as a plug-in of the Fujaba/MOFLON meta modeling environment [12].

References

[1] *QVT Merge Group: Revised Submission for MOF 2.0 Query, View, Transformation Request For Proposal (ad/2005-03-02) Version 2.0* (2005).

[2] Becker, S. and B. Westfechtel, *Incremental Integration Tools for Chemical Engineering: An Industrial Application of TGGs*, in: *29th Intl. Workshop Graph-Theoretic Concepts in Computer Science*, LNCS **2880** (2003), pp. 46–57.

[3] Ermel, C. and K. Ehrig, *View transformation in visual environments applied to algebraic high-level nets.*, ENTCS **127** (2005), pp. 61–86.

[4] Gamma, E., R. Helm, R. Johnson and J. Vlissides, Addison-Wesley PCS, Addison-Wesley Publishing Company Inc., USA, 1995.

[5] Grundy, J. and J. Hosking, *Constructing integrated software development environments with mviews*, in: *International Journal of Applied Software Technology Vol.2*, 1996.

[6] Guerra, E. and J. de Lara, *Event-driven grammars: Towards the integration of meta-modelling and graph transformation.*, in: *ICGT*, 2004, pp. 54–69.

[7] Königs, A. and A. Schürr, *MDI - a Rule-Based Multi-Document and Tool Integration Approach*, Special Section on Model-based Tool Integration in Journal of Software&System Modeling (2005).

[8] Königs, A. and A. Schürr, *Tool Integration with Triple Graph Grammars - A Survey*, in: *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, ENTCS (2005).

[9] Kozankiewicz, H. and K. Subieta, *SBQL Views - Prototype of Updateable Views.*, in: *ADBIS*, 2004.

[10] McBrien, P. and A. Poulovassilis, *Defining Peer-to-Peer Data Integration Using Both as View Rules.*, in: *DBISP2P*, 2003, pp. 91–107.

[11] MetaCase, "MetaEdit+," (2005).
URL http://www.metacase.com

[12] Real-Time Systems Lab - TU Darmstadt, "MOFLON," (2005).
URL http://www.moflon.org

[13] Rekers, J. and A. Schürr, *A Graph Based Framework for the Implementation of Visual Environments*, in: *Proc. VL'96 12th Int. IEEE Symp. on Visual Languages, Boulder, Colorado* (1996), pp. 148–155.

[14] Schürr, A., *Specification of graph translators with triple graph grammars*, in: Mayr and Schmidt, editors, *Proc. WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*, 1994, pp. 151–163.

[15] Software Engineering Group - University of Paderborn, "FUJABA," (2005).
URL http://www.fujaba.de

[16] T. Risch, T. K., V. Josifovski, *Functional Data Integration in a Distributed Mediator System*, in: *Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data* (2003).

[17] Zhu, N., J. C. Grundy and J. G. Hosking, *Pounamu: A meta-yool for multi-view visual language environment construction.*, in: *VL/HCC*, 2004, pp. 254–256.

Towards Verifying Model Transformations

Anantha Narayanan [1,2] and Gabor Karsai [3]

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

In model-based software development, a complete design and analysis process involves designing the system using the design language, converting it into the analysis language, and performing the verification and analysis on the analysis model. Graph transformation is increasingly being used to automate this conversion. In such a scenario, it is very important that the conversion preserves the semantics of the design model. This paper discusses an approach to verify this semantic equivalence for each transformation. We will show how to check whether a particular transformation resulted in an output model that preserves the semantics of the input model with respect to a particular property.

Key words: Graph Transformation, Verification, Bisimulation.

1 Introduction

Domain specific modeling languages (DSMLs) greatly simplify the task of the system designer, presenting a higher level of abstraction that is easy to work with. DSMLs also facilitate analysis by providing an appropriate abstraction. However, it is not always the case that the same language is suitable for both design and analysis. For instance, Statecharts are very powerful for designing concurrent systems, but their analysis is usually not simple. Extended Hybrid Automata (EHA) were introduced in [3] as an intermediate, simpler language with a more restricted syntax. Subsequent work [4] has shown that this intermediate format can be used to generate verification models that may be verified using model checking tools such as SPIN [5].

[1] The research described in this paper has been supported by a grant from NSF/CSR-EHS, titled "Software Composition for Embedded Systems using Graph Transformations", award number CNS-0509098.

[2] Email: ananth@isis.vanderbilt.edu

[3] Email: gabor.karsai@vanderbilt.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

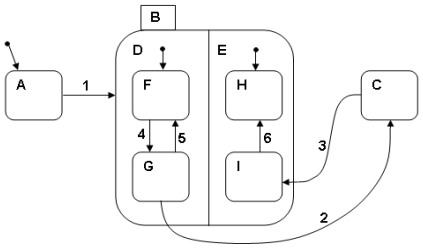


Fig. 1. A sample Statechart model

Graph transformation has been suggested as a powerful and convenient method for transforming design models into analysis models. The transformation must ensure that the analysis model preserves the semantics of the design model, and truthfully represents the design. As a first step towards this goal, it would be useful to establish that the transformed model is semantically equivalent to the source model, with respect to the property we wish to verify. In this paper, we study this notion of equivalence between the two graphs, and a way to check if there exists a bisimulation relation between the graphs. If it is possible to prove that the analysis model behaves in exactly the same way as the design model with respect to a certain property, then we can conclude that checking for the property in the analysis model is equivalent to checking for the same property in the original design model. In the following sections, we will go through the basics of graph transformation principles and tools, and demonstrate our approach to checking the equivalence using Statechart models and EHA models.

2 Background

2.1 Model Integrated Computing

Model Integrated Computing (MIC) [1] is an approach to system development using domain specific models to represent the architecture and behavior of the system and its environment. The development process involves the creation of a meta-model that defines the abstract syntax of the domain, from which a Domain Specific Design Environment (DSDE) is generated. The DSDE can be used to create domain specific models. These models are usually transformed to other formats, such as executable code, or to perform analysis. The MIC tool suite containing GME [6] and GReAT [7] were used in developing the examples for this paper.

2.2 GReAT

The transformations in this paper will be written in GReAT [7], a language for specifying graph transformation rules. GReAT belongs to the class of practical graph transformation systems such as AGG [8], PROGRES [9] and FUJABA [10]. It uses UML and OCL to specify the domains of the transformation.

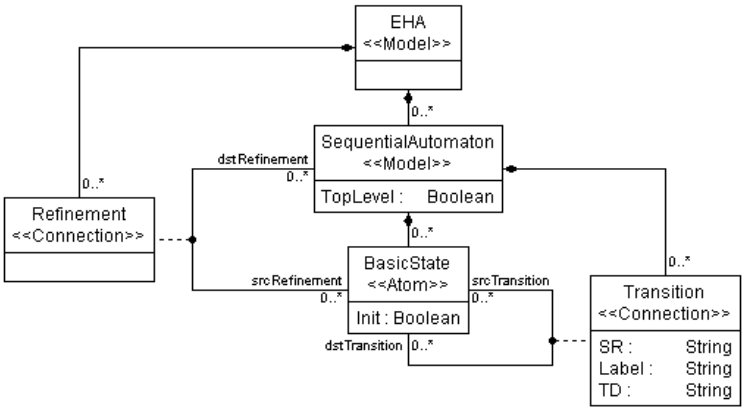


Fig. 2. EHA meta-model in UML

GReAT allows users to compose source and target meta-models by defining temporary vertex and edge types that can span across multiple domains and will be used temporarily during the transformation. This enables us to tie the different domains together to make a larger, heterogeneous domain that encompasses all the domains and cross-links. This feature plays an important role in our approach to verifying transformations.

2.3 Statecharts

State machines, based on Harel's statecharts [11] are used in UML to represent the reactive behavior of systems. State machines are constructed from *states* and *transitions*. States may be simple, composite or concurrent. States may be connected by directed edges called transitions. Transitions connecting states contained in different levels of hierarchy are called *inter-level* transitions. Figure 1 shows an example of a Statechart. Transitions 2 and 3 in the figure are inter-level. A *state configuration* is a maximal set of states that the system can be active in simultaneously. State configurations are closed upwards, meaning that if a system is in a state A, then it must also be in A's parent state. Some valid state configurations in Figure 1 are {A}, {B, F, H} and {B, G, I}.

2.4 EHA

Extended Hierarchical Automata (EHA) were introduced as an alternate representation to provide formal operational semantics for Statechart diagrams. EHA offer an alternative simplified hierarchical representation for Statecharts that helps in correctness proofs [2]. The meta-model for EHA in UML is shown in Figure 2.

Each Statechart model can be represented by one EHA model. Every compound state in the Statechart model is represented by a Sequential Automaton in the EHA. There is one top level Sequential Automaton for the EHA, which

represents the initial automaton. Each state in the Statechart has a corresponding Basic State in the EHA. If a state is compound in the Statechart, then it is further "refined" into a Sequential Automaton in the EHA, which will contain Basic States corresponding to all the states within the compound state in the Statechart. Similarly, these states may be refined further.

Transitions in EHA are always within a single Sequential Automaton, i.e. there are no inter-level transitions in an EHA. Inter-level transitions in Statecharts are elevated based on the scope of the transition, to the Sequential Automaton representing the lowest common ancestor of the start and end states of the transition in the Statechart. EHA transitions have special attributes called "source restriction" and "target determinator", which keep track of the actual source and target of the transition in the Statechart. The conversion of Statechart models into EHA models will be discussed in detail in the next section.

3 Verifying graph transformations

Graph transformation systems such as GReAT allow users to transform models of one meta-model to models of another meta-model using a collection of pattern matching rules. However, it is not certain whether the output of the transformation preserves the semantics of the source model that we intend to analyze. Important semantic information may easily be lost or misinterpreted in a complex transformation, due to errors in the graph rewriting rules or in the processing of the transformation. We need a method to verify that the semantics that we are interested in analyzing are indeed preserved across the transformation.

We propose an approach to check whether the semantics of the input model were preserved in the output model of a transformation. We are *not* trying to prove the correctness of the graph transformation rules in general, but check if a *particular* generated model is a valid representation of a *particular* source model, in order to verify a particular property about the source model. We accomplish this by defining an equivalence relation between objects of the input and the output model, and use this to check if the two models are similar in behavior.

3.1 Bisimilarity

Two systems can be said to be *bisimilar* if they behave in the same way, i.e. one system simulates the other and vice-versa. A bisimulation relation can be defined formally as follows.

Definition 3.1 Given a labeled state transition system (S, Λ, $\rightarrow$), a *bisimulation relation* is defined as an equivalence relation R over S, such that for all p, $q \in$ S, if (p, q) is in R, and for all $p' \in$ S and $\alpha \in \Lambda$, $p \rightarrow^{\alpha} p'$ implies that there exists a $q' \in$ S such that $q \rightarrow^{\alpha} q'$ and (p', q') is in R, and conversely,

for all q' $\in$ S, $q \rightarrow^{\alpha} q$' implies $p \rightarrow^{\alpha} p$' and (p', q') is in R.

Though this definition is given in terms of a single set S, we can think of equivalence of two transition systems in terms of a global set containing both the system's states. In our approach to verifying whether the semantics are preserved across a transformation, we will check whether there is a bisimulation relation between the source model and the target model.

3.2 Transforming Statecharts into EHA

The EHA notation for Statecharts can be obtained by a graph transformation process [2]. The basic steps of the transformation are listed below:

(i) Every Statechart model can be transformed into an EHA model, with one top level Sequential Automaton in the EHA model.

(ii) For every (primitive or compound) state in the Statechart (except for regions of concurrent states), a corresponding basic state is created in the EHA.

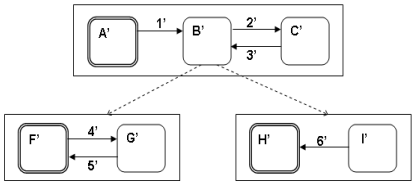
(iii) For every composite state in the Statechart model, a Sequential Automaton is created in the EHA model, and a "refinement" link connects the Basic State in the EHA corresponding to the state in the Statechart, to the Sequential Automaton in the EHA that it is refined to.

(iv) All the contained states in the composite state are further transformed by repeating steps (ii) and (iii). The top level states in the Statechart will go into the top level Sequential Automaton in the EHA.

(v) For every non-interlevel transition in the Statechart model a transition is created in the EHA between the Basic States corresponding to the start and end states of the transition in the Statechart model.

(vi) For every inter-level transition in the Statechart model, we trace the scope of the transition to find the lowest parent state s_P that contains both the source and the target of the transition. A transition is created in the EHA, in the Sequential Automaton corresponding to s_P. The source of the transition in the EHA is the Basic State corresponding to the highest parent of the source in the Statechart that is within s_P, and the target in the EHA is the Basic State corresponding to the highest parent of the target in the Statechart that is within s_P. The transition in the EHA is further annotated, with the "source restriction" attribute set to the basic state corresponding to the actual source in the Statechart, and the "target determinator" set to the basic state corresponding to the actual target in the Statechart.

Figure 3 shows the EHA model obtained by transforming the Statechart model shown in Figure 1. The table on the top right of the figure shows the values for the *source restriction* and *target determinator* annotations for two of the transitions.



Transition Label	SR	TD
2'	G'	
3'		I', F'

Fig. 3. Sample EHA model

3.3 Behavioral equivalence of the Statechart model and the EHA model with respect to reachability

A "state configuration" in a Statechart is a valid set of states that the system can be active in. If a state is part of an active configuration, then all its parents are also part of the active configuration. A transition in the Statechart can take the system from one state configuration to another state configuration, where the source and target states of the transition are subsets of the initial and final state configurations. A state configuration S_f in the Statechart is said to be "reachable" from a state configuration S_i if there exists a series of valid transitions that can take the system from S_i to S_f.

Similarly, a state configuration in an EHA model is a set of Basic States. If a Basic State is part of an active configuration, and is part of a non-toplevel Sequential Automaton, then the Basic State that is refined into this Sequential Automaton is also a part of the active configuration. For instance, B', F', I' is a valid active configuration in Figure 3. A transition in the EHA can take the system from one state configuration to another state configuration, where the union of the source of the transition and its source restriction are a subset of the initial state configuration, and the union of the target of the transition and its target determinator are a subset of the final state. A state configuration S_f in the EHA is said to be "reachable" from a state configuration S_i if there exists a series of valid transitions that can take the system from S_i to S_f.

An EHA model truly represents the reachability behavior of a Statechart model, if every reachable state configuration in the Statechart has an equivalent reachable state configuration in the EHA and vice versa.

For every state s in the Statechart, we have a unique Basic State s' in the EHA. We can specify an equivalence relation R, such that (s, s') $\in R$ and say that s' is equivalent to s. A state configuration S in the Statechart is equivalent to a state configuration S' in the EHA if for all $s \in S$ there is an equivalent $s' \in S'$, and for all $s' \in S'$, there is an equivalent $s \in S$. Furthermore, for every transition t in the Statechart, we have a unique transition t' in the EHA. We can specify an equivalence relation R_t, such that (t, t') $\in R_t$ and say that t' is equivalent to t.

Given the relations R and R_t, we can check if there is a bisimulation relation between the two models using the following definition.

Definition 3.2 Given a state configuration S_A in the Statechart model, and

its equivalent state configuration S_B in the EHA model, the equivalence is a bisimulation if for each transition t from S_A to a state configuration S_A' in the Statechart, there exists an equivalent transition t' in the EHA from S_B to a state configuration S_B', and S_B' is equivalent to S_A' (and vice versa)

If this relation is a bisimulation, then verifying the EHA model for reachability will be equivalent to verifying the Statechart model for reachability. If the check fails, it means that there was an error in the transformation.

3.4 Checking for bisimilarity by using cross-links to trace equivalence

GReAT allows us to link input model elements to target model elements using special associations that belong to a composite meta-model, and we call them *cross-links*. These cross-links are maintained throughout the transformation, and used to trace the equivalence relations R and R_t.

When a transformation creates the Basic States and the transitions in the target EHA model, it is known to which states and transitions they correspond to in the Statechart model. What is not certain is whether all states in the Statechart are transformed correctly, all composite states are refined correctly, all transitions are transformed correctly, and all transitions connect the correct sets of states. When a rule matches a state or a transition in the Statechart and creates the equivalent Basic State or transition in the EHA, a cross-link association called "equivalentTo" is created between the Statechart object and its corresponding EHA object. When the transformation completes, the relations R and R_t can be traced using these associations.

Rather than checking for all possible state configurations in the Statechart, it would be more efficient to consider every transition in the Statechart and its minimal required source configuration. Any superset of this state configuration will be a valid starting configuration, and will not have to be investigated further. For every transition t in the Statechart model, and its equivalent transition t' in the EHA model, if their start state configurations S_A and S_B are equivalent, and also their end state configurations S_A' and S_B' are equivalent, then there exists a bisimulation for this particular instance, according to our definition.

The implementation follows straightforwardly from the discussion. At the end of the transformation, we have access to the source model graph, the output model graph, and also the cross-links between the two. We collect the set of all the transitions from the source graph. For each transition in this set, we find the equivalent transition in the EHA by following the "equivalentTo" cross-link. Now we can compute the minimal source state configuration S_A for the transition in the Statechart model, and the source state configuration S_B for the EHA model. We check the equivalence of S_A and S_B by taking every state s in S_A, finding its equivalent state s' form the EHA, and checking if s' is in S_B, and vice versa. The target states are also checked similarly. If this check succeeds for all transitions in the Statechart, and there are no more

transitions in the EHA, then the two systems can be said to be bisimilar with respect to checking reachability. In other words, we can conclude reachability in the Statechart model by verifying it in the EHA model. If this check fails, then there may be errors in the transformation, and the generated EHA model does not truly represent the input Statechart model.

The final step is checking the reachability in the EHA model. [4] provide ways to generate a Promela model from an EHA model, which can be checked using the SPIN model checker. To check the reachability of a certain state configuration, a claim can be attached to the SPIN model that verifies whether that configuration is reachable in the model. Alternately, a claim can be made in SPIN that says that the state is not reachable. If it is indeed reachable, the SPIN verifier refutes this claim and presents a counter-example, as a trace that leads to this state configuration. This represents a valid series of transitions in the EHA that leads to the specified state configuration. As a corollary, we may use the cross-links created during the transformation, to reproduce this trace in the Statechart model. In this way, reachability in the Statechart model can be verified by verifying it in the EHA model.

It should be noted that the technique described above is not an attempt to prove the correctness of the graph transformation rules in general. This is a method to verify if a particular instance of a transformation is valid, and must be executed for each transformation individually. We also do not try to prove the general semantic equivalence of models. We identify the equivalence relations with respect to a specific property and test if there is a bisimulation. The complexity of the transformation is not increased significantly by this method. As the cross-links are created every time the objects of the output model are created, and as we directly trace these cross-links during checking, the complexity of the check is proportional to the size of the model, and not the state space of the model. In other words, we can perform this check without actually having to execute the models.

4 Related work

We now discuss some related work in the area of automatic verification using model checking, graph transformations and other types of proofs.

4.1 Verifying properties by converting models into an intermediate format

[2] [3] convert Statechart models into EHA models. [4] create Promela models from the EHA models, which can be verified using the SPIN model checker. Our approach will be useful in these instances, to provide a certificate that the intermediate formats truly preserve the property we wish to verify using them. An interesting research problem is whether our approach can be used to check whether the generated Promela model (which is code in plain text) truly represents the EHA model it was generated from.

4.2 Operational semantics using graph transformations

[12] [13] [14] are some works on using graph transformation rules to specify the dynamic behavior of systems. [14] presents a meta-level analysis technique where the semantics of a modeling language are defined using graph transformation rules. A transition system is generated for each instance model, which can be verified using a model checker. [15] verifies if a transformation preserves certain dynamic consistency properties by model checking the source and target models for properties p and q, where property p in the source language is transformed into property q in the target language. This transformation requires validation by a human expert. Our method does not check whether the models themselves satisfy a property, but automatically does check whether the models are equivalent with respect to that property.

4.3 Certifiable program generation

[16] considers the problem of verification of generated code by focusing on each individual generated program, instead of verifying the program generator itself. The generator is extended such that it produces all logical annotations that are required for formal safety proofs in a Hoare-style framework. These proofs certify that the program does not violate certain conditions during its execution. While the proofs in this case are not related to semantic correctness, the idea of providing an instance level certificate of correctness instead of proving the correctness of the generator has been a great motivation for our ideas.

5 Summary

We have described a method for checking if a certain execution of a transformation produced an output model that preserved the semantics of the input model. This check is important when the output model is used for verification and analysis, as errors in the transformation may result in an output model that does not truly represent the input model. We are studying how such an equivalence can be established when the target model abstracts away a lot of detail in the source model. Our method does not attempt to prove the correctness of the transformation itself, but checks whether a particular execution produced a correct result. This check does not adversely affect the complexity of the transformation.

References

[1] Sztipanovits J., Karsai G., "Model-Integrated Computing", *IEEE Computer*, pp. 110-112, April, 1997.

[2] Varró D. "A Formal Semantics of UML Statecharts by Model Transition Systems", Proc. ICGT 2002: 1st International Conference on Graph Transformation, *LNCS*, vol. 2505, pp. 378-392.

[3] Mikk E., Lakhnech Y., and Siegel M., "Hierarchical automata as model for statecharts", In R. Shyamasundar and K. Euda, editors, ASIAN97 Third Asian Computing Conference. Advances in Computer Science, volume 1345 of *LNCS*, pages 181196. Springer-Verlag, 1997.

[4] Latella D., Majzik I., and Massink M., "Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker", *Formal Aspects of Computing*, 11(6), pp. 637 664, 1999.

[5] Holzmann G., "The model checker SPIN", *IEEE Transactions on Software Engineering*, 23(5), pp. 279-295, 1997.

[6] Ldeczi A. et. al., "Composing Domain-Specific Design Environments", *IEEE Computer*, November 2001, pp. 44-51.

[7] Agrawal A., Karsai G., Ledeczi A., "An End-to-End Domain-Driven Software Development Framework", 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, October 26, 2003.

[8] Gottler H., "Attributed graph grammars for graphics", H. Ehrig, M. Nagl, and G. Rosenberg, editors, Graph Grammars and their Application to Computer Science, LNCS 153, pages 130-142, Springer-Verlag, 1982.

[9] Schürr A., Winter A. J., and Zündorf A., In [20], chap. The PROGRES Approach: Language and Environment, pp. 487 550. World Scientific, 1999.

[10] Nickel U., Niere J., and Zündorf A.. Tool demonstration: The FUJABA environment. In The 22nd International Conference on Software Engineering (ICSE). ACM Press, Limerick, Ireland, 2000.

[11] Harel D., "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 8(3), pp. 231274, 1987.

[12] Corradini A., Heckel R., Montanari U. "Graphical operational semantics", In Proc. ICALP2000

[13] Schmidt A., Varró D., "CheckVML: A Tool for Model Checking Visual Modeling Languages", In Proc. UML 2003: 6th International Conference on the Unified Modeling Language, *LNCS*, vol. 2863, pp. 92-95.

[14] Varró D., "Automated Formal Verification of Visual Modeling Languages by Model Checking", *Journal of Software and Systems Modeling*, col. 3(2), pp. 85-113.

[15] Varró D. and Pataricza A., "Automated Formal Verification of Model Transformations", In Critical Systems Development in UML 2003, pp. 63-78.

[16] Denney E., Fischer B., "Certifiable Program Generation", GPCE 2005, *LNCS*, vol. 3676, pp. 17-28.

Augur 2 — A New Version of a Tool for the Analysis of Graph Transformation Systems ⋆

Barbara König Vitali Kozioura

Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany
{koenigba,koziouvi}@fmi.uni-stuttgart.de

Abstract

We describe the design and the present state of the verification tool Augur 2 which is currently being developed. It is based on Augur 1, a tool which can analyze graph transformation systems by approximating them by Petri nets. The main reason for the new development was to create an open, flexible and extensible verification environment. Also, compared to the previous version, Augur 2 will include more functionality and new analysis techniques.

1 Introduction

In the last few years we have developed the verification tool Augur 1 (or simply Augur) [12] which analyzes graph transformation systems (GTSs) by approximating them with Petri nets. Using this tool we have conducted several case studies, verifying, for instance, a mobile system with a firewall [4], a mutual exclusion protocol [10] and the insertion of elements into red-black trees [2]. Other examples of systems for which our technique is suitable are dynamic pointer structures on a program heap, object graphs and reconfigurable networks with mobile processes. The tool can be obtained from http://www.fmi.uni-stuttgart.de/szs/tools/augur/.

We started with a small tool that reads GTXL files and produces GXL files (GXL respectively GTXL are XML standards for the encoding of graphs and graph transformation systems [14]). Afterwards we faced the constant necessity of adding new features and new functionality. More specifically, we added analysis algorithms for Petri nets [20] based on coverability graphs [15] and backward reachability [1]. Furthermore we established an interface to Graphviz [1] for visualization purposes. We also added the possibility to specify forbidden paths in graphs using regular expressions [16], we implemented the

⋆ Research supported by DFG project SANDS.

[1] http://www.graphviz.org/

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

finite complete prefix technique for graph transformation systems [5,8] and we started to extend the tool in order to use it for the purpose of test case generation [11]. Probably the most extensive addition was to add support for counterexample-guided abstraction refinement [13].

The architecture of Augur 1 was strongly oriented towards the concrete task of approximated unfolding of GTSs. This made all changes mentioned above hard to implement and led to several versions of the tool, each with a different functionality. Hence the new version of Augur (called Augur 2) will have a more general and extensible software architecture and will have more functionality concerning analysis and visualization methods.

Another new feature of Augur 2 will be the possibility to work with attributed graphs, i.e., graphs with (integer and string) attributes assigned to nodes and edges. As a future research topic we plan to extend existing analysis techniques accordingly. Support for input and output will also be extended, for instance we are currently working on an interface to Agg [19]. Also, we plan to have a simple pointer-manipulating programming language, which can be translated into graph rewriting, as an additional means of input.

The kernel part of the tool is already developed and has successfully passed through a number of tests. At the moment the tool is being extended with various visualization and analysis methods.

2 Graph Transformation Systems and Verification Techniques

We use hypergraph rewriting where left-hand and right-hand sides can be (almost) arbitrary hypergraphs. Compared to the double-pushout approach our GTSs have to observe some restrictions: especially, the interface graph of a rule must be discrete, no nodes can be deleted and rules must be consuming, i.e., at least one edge is deleted. While the last two restrictions are essential for the unfolding-based approach we are following, the first restriction (the interface is discrete) will be lifted in Augur 2.

In order to illustrate the basic ideas behind the tool, we will start with a simple example. Fig. 1 shows a GTS, which models a network consisting of connections, private servers, internal and external processes, where the network is constantly extended during runtime. Furthermore processes may cross connections. The property we want to verify is that no external process will ever visit a private server. We reduce this property to "no Error edge will be created" by adding a rule creating an edge labelled "Error" as soon as the forbidden situation has been detected.

Since GTSs are in general Turing-powerful over-approximation techniques are needed for their analysis. In our case we abstract GTSs by Petri nets, which are a conceptually simpler formalism and for which several verification techniques have already been developed. More specifically, the tool is based on an approximated unfolding technique for GTSs, presented in [3]. Compared to

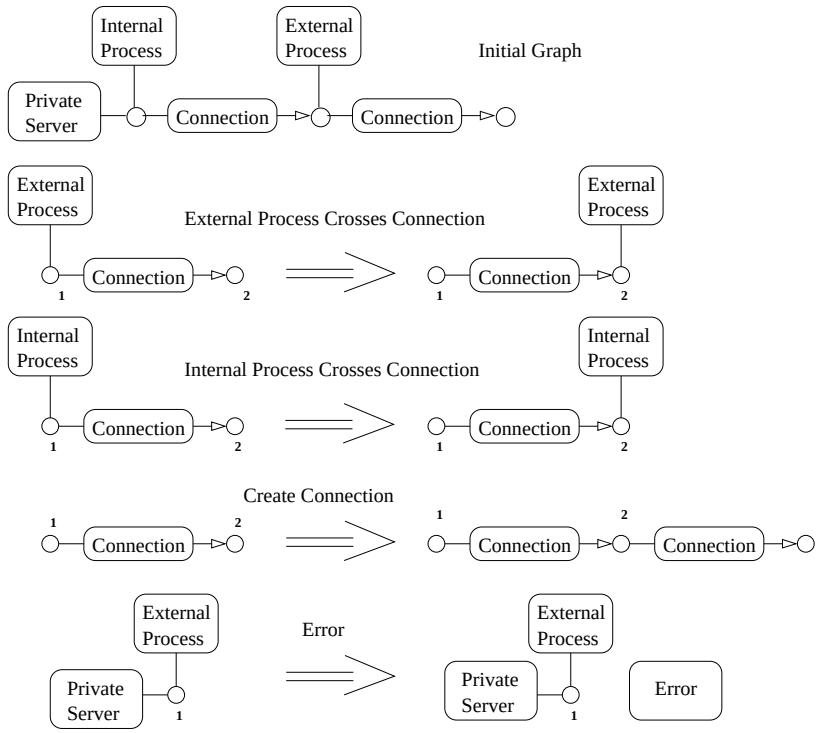


Fig. 1. Example graph transformation system.

a standard unfolding technique we are additionally using folding steps which over-approximate, but guarantee a finite approximation. The tool constructs an over-approximation, which is a so-called Petri graph (i.e., a hypergraph with a Petri net structure over it, see [3]). The hyperedges are at the same time the places of the net. Fig. 2 depicts the coarsest over-approximation for the example GTS in Fig. 1.

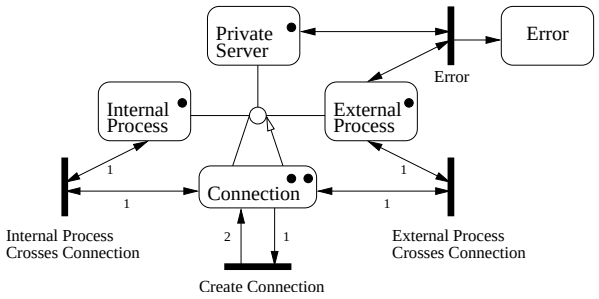


Fig. 2. Petri graph approximating the GTS (first approximation).

The Petri graph is an over-approximation in the following sense: (i) every reachable graph can be mapped to its hypergraph component via a (usually non-injective) graph morphism and (ii) the multi-set image of its edges corresponds to a reachable marking of the net. For instance the five edges of the initial graph correspond to the five tokens of the initial marking of the net. More generally there exists a simulation relation between the reachable graphs and the reachable markings of the net, obtained by firing enabled transitions.

More details can be found in [3,6].

If the over-approximation is too coarse and does not allow to verify the property, techniques for refining the approximation are available. One such technique is counterexample-guided abstraction refinement [13] which starts from a concrete counterexample found by coverability checking on the Petri net. Another possibility is to use depth-based refinement [6] which constructs an over-approximation exact up to a pre-defined depth in the unfolding. Counterexample-guided abstraction refinement usually results in smaller approximations and faster verification.

The edge "Error" of the Petri graph in Fig. 2 can be covered by firing transition "Error". This means that either the property does not hold or the over-approximation is too coarse. One can show that the run is spurious, i.e., it has no counterpart in the original GTS, which indicates that we have approximated too much. Applying abstraction refinement gives us a refined Petri graph, which is depicted in Fig. 3. There exists no edge labelled "Error", i.e., such an edge can also not be covered by any reachable marking. So, from the correspondence between reachable graphs and markings it follows that the property has been successfully verified.

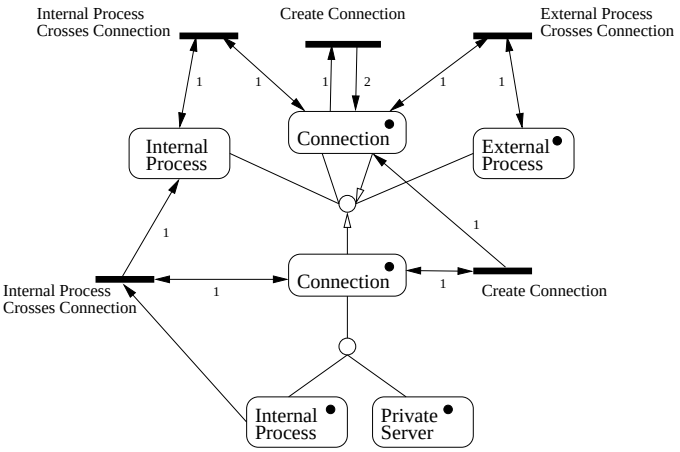


Fig. 3. Petri graph after counterexample-guided abstraction refinement.

3 Software Design

In this section we will present the main ideas behind the new implementation, which lead to an open and flexible new verification tool.

The central part of the software design is the concept of *algorithms*, which are implemented as classes. Each program module working with the common data structures should be realized as an algorithm and new algorithms can be added during the whole life time of the system. As examples of algorithms we mention here different operations on Petri graphs (firing of transitions, building the coverability graph, searching for matches of left-hand sides, performing folding/unfolding steps, etc.) and input/output operations (readers

and writers from/to different data formats).

All algorithms work with the same data structures which makes it possible to use some algorithms as sub-operations inside others and to assemble new algorithms out of existing ones. A *scenario* is a special algorithm which uses as input and output only the external data sources, such as XML files (Fig. 4). Scenarios are at the top level of the system and call other algorithms. A typical example for a scenario is the approximated unfolding algorithm which reads a graph transformation system and outputs a Petri graph.

All algorithms and scenarios are managed in a central database system (see Fig. 4). The database consists of several tables, the most important being the *algorithm table*, which is shown in Table 1.

Calling Algorithm	Label	Algorithm To Call	History Path
main	a	unfold	∅
unfold	a	findMatch2	(*,*)(a, findMatch1)(*,*)
unfold	a	findMatch1	∅
unfold	b	findMatch3	∅

Table 1
Example for the algorithm table in the database system.

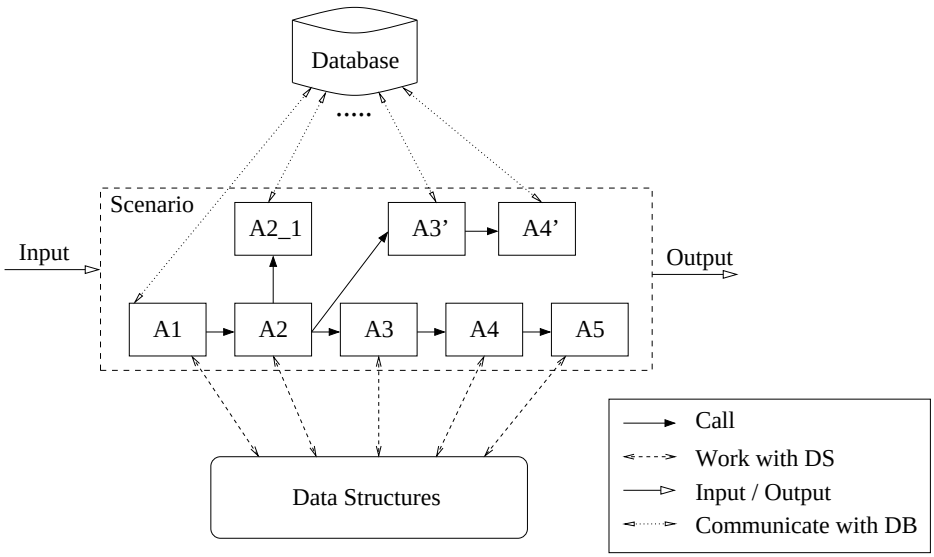


Fig. 4. Schematic depiction of a scenario.

The first column of the table represents the name of the algorithm calling another algorithm as a sub-operation. The second column is the label of the place where the sub-operation is being called. Labels are used by algorithms in order to indicate which kind of other algorithms they intend to call. Then, the information in the database determines which of the several available

algorithms for this task is chosen. This information, i.e., the name of the algorithm, is given in the third column. Finally, the fourth column is a regular expression representing the history path or the call stack of the algorithm in the first column. Depending on this history different algorithms can be called from the same place in the code. We use the information of the first matching entry in the table. Hence, this table makes it easy to exchange a sub-operation by another sub-operation performing basically the same function in a different way (optimized for the concrete situation).

Table 1 represents a typical example—the control of the match finder algorithm, which searches for matches of left-hand sides in a (large) hypergraph. This operation is one of the critical parts in the calculation of the approximating unfolding and there are different ways to implement it [23,18,22]. We call three different match finder algorithms depending on the place in the unfolding algorithm where the match finder is called and on the local history of calls. At the place labelled 'b' we will always call "findMatch3". If at the place labelled 'a' the algorithm "findMatch1" has already been called, "findMatch2" will be called next. This is very similar to what happens in our implementation since in different situations matches have to be located in slightly different ways. For instance for a folding step two matches have to be found instead of one.

Another example is coverability checking for Petri nets, for which we currently use two different algorithms: computation of coverability graphs and backward reachability. The current layout of the tool also makes it easy to replace an old inefficient version of an algorithm by a more efficient one and to use different versions of an algorithm in different situations.

Besides the algorithm table there exists some other information needed to manage the behavior of algorithms. For example, there is a table describing the reusability of algorithms, i.e., which says whether a new object should be created when a new algorithm is requested or if a previously created object can be reused. Also, there exist protocols governing the communication between algorithms. For example algorithms can notify each other about changes in the data structures (validation protocol).

4 System Architecture

After presenting the general ideas behind Augur 2, we will now describe the architecture behind this tool (see Fig. 5).

We will explain Fig. 5 in roughly chronological order, starting with the input (a graph transformation system and a specification of the property to be verified) and ending with the final output, which says whether the property holds. The system starts by reading the graph transformation system from an external source. At the moment we consider the following three possibilities:

- Read the GTS from a file in GTXL-format (implemented).
- Use Agg as an input source in order to draw graph transformation systems

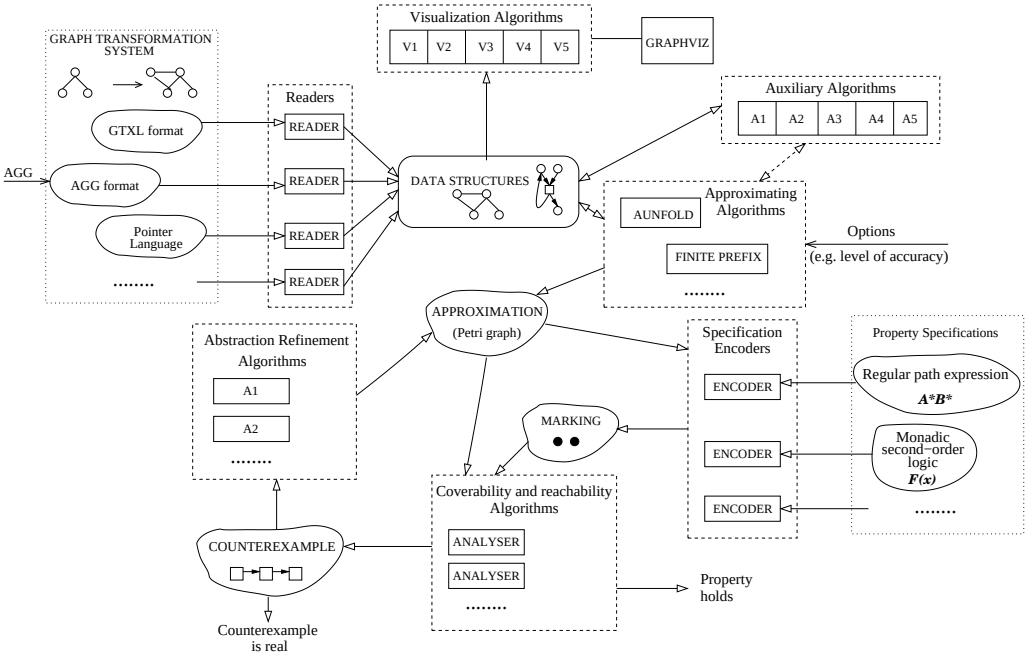


Fig. 5. Schematic representation of Augur 2.

(under development).

- Write a program in a simple pointer-manipulating language, which is then converted automatically to a graph transformation system (under development).

After reading the GTS from an input source and converting it to the internal data structures it can be visualized using Graphviz and abstracted using, for instance, the approximated unfolding algorithm AUNFOLD. A different possibility is to calculate the finite complete prefix of the unfolding of the system (which—in the case of finite-state systems—represents all reachable graphs in a partially ordered structure).

Apart from the graph transformation system, we require the property which has to be verified as additional input. For this purpose we consider in Augur 2 the following two possibilities:

- Specify a regular expression with the set of hyperedge labels as the alphabet. This regular expression describes forbidden paths which should not occur in any reachable graph (implemented).
- Monadic second-order logic for hypergraphs (under development, for the underlying theory see [7]).

These specification languages have to be translated into properties on Petri net markings, since the analysis has to be done directly on the Petri net structure underlying the Petri graph. The coverability of these markings can then be checked using various algorithms (described above). We also plan to implement an (approximative) reachability checker for Petri nets.

If the property does not hold, a counterexample for the net is generated. In the case of spurious counterexamples one of the refinement algorithms is used to obtain a more exact approximation. This procedure can be iterated.

Whenever a non-spurious counterexample is found, we have detected an error in the GTS, i.e., the property to be verified does not hold.

5 Conclusion

Several tools are available for the analysis of graph transformation systems. While some groups [21,9] pursue the idea of translating graph transformation systems into the input language of a model checker, others attempt to develop new specialized methods for graph rewriting. Work from our side goes in this latter direction, as well as [17], which led to the tool GROOVE for verifying finite-state GTS. Properties different from reachability (such as termination and confluence via critical pair analysis) can be analyzed using Agg [19].

In this paper we have summarized our plans for the development of Augur 2, a new version of an analysis and verification tool based on unfolding techniques. Some functionality is already present in the current version Augur 1, furthermore the core part of Augur 2, including the database management, has already been implemented. This tool will enable us to conduct further case studies, which will give us valuable stimulations for the future development of the verification techniques.

Among other ideas our future plans are to implement in Augur 2 the possibility to use and analyze attributed graph transformation systems.

Finally, we recently concluded the implementation of a graphical user interface. A screenshot, together with windows visualizing the graph and net components of a Petri graph, is shown in Fig. 6.

Acknowledgements: We want to thank all students who helped us with the implementation of Augur: Ingo Walther, Sinan Turan, Nicolas Relange, Julian Bart, Martin Horsch, Olga Danylevych and Ganna Monakova. Furthermore we would like to thank Paolo Baldan, Andrea Corradini and Tobias Heindel for valuable discussions on the theoretical background of the tool.

References

[1] Abdulla, P. A., B. Jonsson, M. Kindahl and D. Peled, *A general approach to partial order reductions in symbolic verification*, in: *Proc. of CAV '98* (1998), pp. 379–390, LNCS 1427.

[2] Baldan, P., A. Corradini, J. Esparza, T. Heindel, B. König and V. Kozioura, *Verifying red-black trees*, in: *Proc. of COSMICAH '05*, 2005, proceedings available as report RR-05-04 (Queen Mary, University of London).

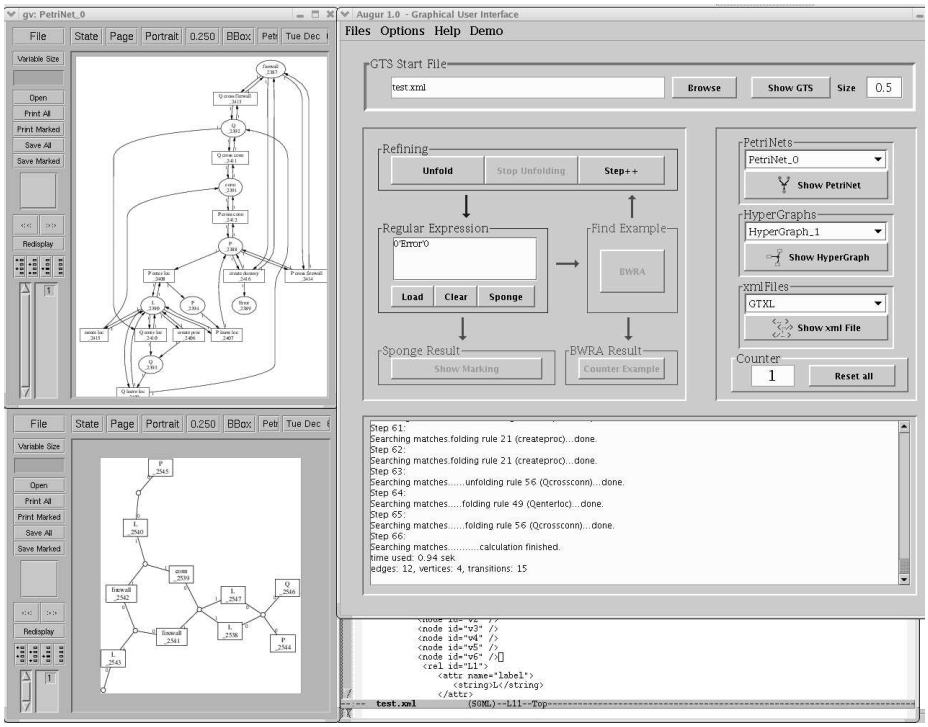


Fig. 6. Screenshot of Augur at work.

[3] Baldan, P., A. Corradini and B. König, *A static analysis technique for graph transformation systems*, in: *Proc. of CONCUR '01* (2001), pp. 381–395, LNCS 2154.

[4] Baldan, P., A. Corradini and B. König, *Static analysis of distributed systems with mobility specified by graph grammars—a case study*, in: *Proc. of IDPT '02* (2002).

[5] Baldan, P., A. Corradini and B. König, *Verifying finite-state graph grammars: an unfolding-based approach*, in: *Proc. of CONCUR '04* (2004), pp. 83–98, LNCS 3170.

[6] Baldan, P. and B. König, *Approximating the behaviour of graph transformation systems*, in: *Proc. of ICGT '02* (2002), pp. 14–29, LNCS 2505.

[7] Baldan, P., B. König and B. König, *A logic for analyzing abstractions of graph transformation systems*, in: *Proc. of SAS '03* (2003), pp. 255–272, LNCS 2694.

[8] Bart, J., "Effiziente Entfaltungsalgorithmen für Graphersetzungssysteme," Master's thesis, Universität Stuttgart (2005), no. 2290.

[9] Dotti, F. L., L. Foss, L. Ribeiro and O. M. Santos, *Verification of distributed object-based systems*, in: *Proc. of FMOODS '03* (2003), pp. 261–275, LNCS 2884.

[10] Dotti, F. L., B. König, O. M. dos Santos and L. Ribeiro, *A case study: Verifying a mutual exclusion protocol with process creation using graph transformation systems*, Technical Report 08/2004, Universität Stuttgart (2004).

[11] Horsch, M., *Test case generation for rule-based translators* (2005), Studienarbeit (Student research project) No. 1984, Universität Stuttgart.

[12] König, B. and V. Kozioura, Augur—*a tool for the analysis of graph transformation systems*, EATCS Bulletin **87** (2005), pp. 125–137, appeared in The Formal Specification Column.

[13] König, B. and V. Kozioura, *Counterexample-guided abstraction refinement for the analysis of graph transformation systems*, in: *Proc. of TACAS '06* (2006), LNCS. to appear.

[14] Lambers, L., *A new version of GTXL: An exchange format for graph transformation systems*, in: *Proc. of GraBaTs'04*, 2004.

[15] Reisig, W., "Petri Nets: An Introduction," EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, Germany, 1985.

[16] Relange, N., "Verifikation dynamischer Systeme: Reguläre Ausdrücke zur Spezifikation verbotener Pfade," Master's thesis, Universität Stuttgart (2004), no. 2192.

[17] Rensink, A., *Canonical graph shapes*, in: *Proc. of ESOP '04* (2004), pp. 401–415, LNCS 2986.

[18] Rudolf, M., *Utilizing constraint satisfaction techniques for efficient graph pattern matching.*, in: *Proc. of TAGT '98* (1998), pp. 238–251, LNCS 1764.

[19] Taentzer, G., *AGG: A tool environment for algebraic graph transformation*, in: *Proc. of AGTIVE '99* (1999), pp. 481–488, LNCS 1779.

[20] Turan, S., *Effiziente Berechnung der Überdeckbarkeit bei Petri-Netzen* (2004), Studienarbeit (Student research project), No. 1935, Universität Stuttgart.

[21] Varró, D., *Towards symbolic analysis of visual modeling languages*, in: *Proc. of GT-VMT '02*, ENTCS **72** (2002).

[22] Varró, G. and D. Varró, *Graph transformation with incremental updates*, in: *Proc. of GT-VMT '04*, ENTCS **109** (2004), pp. 71–83.

[23] Zündorf, A., *Graph pattern matching in PROGRES*, in: *Proc. of TAGT '94* (1994), pp. 454–468, LNCS 1073.

BPSL Modeler - Visual Notation Language for Intuitive Business Property Reasoning

XU Ke [1]

Department of Automation, Tsinghua University
Beijing, China

LIU Ying [2]

IBM China Research Laboratory
Beijing, China

WU Cheng

Department of Automation, Tsinghua University
Beijing, China

Abstract

The urgent need for reliable business applications demands the emergence of a powerful yet easy-to-use language for business property reasoning. The Business Property Specification Language (BPSL) and its supporting tool (BPSL modeler) are presented to address the issue. BPSL modeler facilitates the specification and understanding of business properties by simplifying the expression of complex logics and common behaviors in business processes and exploiting intuitive notations for property representation. It also serves as a key component of our method for model checking business processes. Important ideas and features of BPSL modeler are provided in this paper to help understand its effectiveness.

Key words: Business Property Specification Language, Temporal Logic, Business Property Template, Formal Verification

1 Introduction

Driven by the growth of complexity in existing business systems and the urgent need for ensuring reliable business applications, it has been recognized to be a promising approach to integrate formal verification techniques like

[1] Email: xk02@mails.tsinghua.edu.cn
[2] Email: aliceliu@cn.ibm.com

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

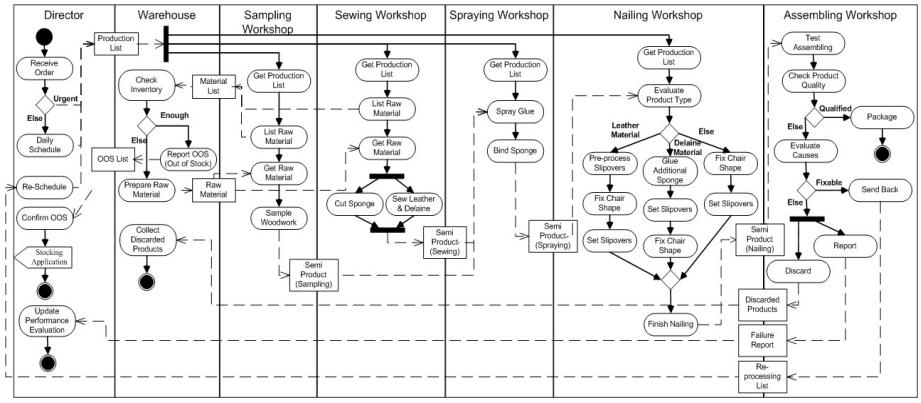


Fig. 1. Business Process for Sofa Manufacturing

$$\begin{aligned}\text{ReportGeneration}=G(\neg\text{reportOOS}) \rightarrow G((\text{receiveorder} \wedge \text{urgent}) \rightarrow (F((\neg\text{sentback } U (\text{sentback} \wedge \\ (X \neg\text{sentback } U (\text{sentback} \wedge \text{report.finished})))) \vee (\neg\text{sentback } U (\text{sentback} \wedge (X \neg\text{sentback } U \\ (\text{sentback} \wedge (X \neg\text{sentback } U (\text{sentback} \wedge \text{report.finished}))))))) \vee G(\neg(\neg\text{sentback } U (\text{sentback} \wedge \\ (X \neg\text{sentback } U (\text{sentback} \wedge (X \neg\text{sentback } U \text{sentback}))))) \wedge F \text{package.inexecution})))\end{aligned}$$

Fig. 2. LTL Specification of ReportGeneration Property

model checking [4] into business domains to help efficiently verify their business processes [13]. Specifying user desired properties is a critical step in the application of model checking. These properties, as expressed in formulas such as temporal logics [4] of CTL or LTL, captures specific user requirements on business processes and are thus called business properties (in short, *BP*). However, the complexity and rigidness of logical formulas is a serious obstacle for business analysts to write and understand their desired properties since most of them are not logical experts. Therefore, tool support for intuitive and easy specification of business properties, automatic generation of their logical formulas are critical for the application of model checking business processes.

Take the following real scenario as an example. Figure 1 illustrates a partial model of a sofa manufacturing process in UML activity diagram. In the process, the factory director delegates manufacturing orders to different workshops. The assembly workshop assembles the semi-finished products and checks their qualities. In order to ensure the correctness of such a complex process design, an even more complex property may need to be specified with which the process model is supposed to comply. For example: "*Without considering the situation of Out Of Stock(OOS), whenever an urgent order is received a product must eventually be packaged if it is re-processed less than three times. However, a failure report must to be generated after its second sent back for re-precessing and before the product is re-precessed the third time*". This property can be captured with LTL as shown in figure 2.

The complexity of business properties in traditional temporal logics can be easily understood from the example. Therefore, the primary contribution of this work is the proposal of an easy-to-use visual notation language called BPSL and its supporting tool BPSL modeler that are tailored for the applica-

tion in business domains. BPSL modeler enjoys the advantage of maintaining good usability and understandability for specifying business properties while still preserving their strict formal semantics to support the formal verification. These advantages are enabled by tailoring BPSL for specifying common behaviors in business processes and absorbing existing business knowledge into the language. As a comparison, we will illustrate in section 3 how the above property can be more intuitively specified with BPSL modeler.

2 Related Works

An important working direction for facilitating temporal property specification is to provide visual extensions for existing logics [3][5][11]. This benefits users by helping understand different semantics of temporal operators with their visualized formalisms. On the other hand the Property Specification Language(PSL) [6], an IEEE standard in digital circuit community, focuses on reducing the complexity in property specification by providing a more flexible choice of temporal operators. When taking a close investigation in business domain, in REALM (Regulations Expressed as Logical Models) [7], an extension of propositional temporal logics is contained to specify compliance rules in business models. A domain specific model checking language tuned for business applications named Strix is proposed in [1]. The property specification in Strix directly uses CTL connectives to explore the business process model.

However, the above works still suffer from the following deficiencies: (1) When plural property is considered (e.g. the one in figure 2), providing visual notations alone is not enough for making a specification language easy-to-use and a property intuitive to understand; (2) Existing knowledge in specifying different business entities (e.g. *activity*, *resource*) and their relations (e.g. *response*, *exclusion*) are not fully exploited to facilitate business property specification and understanding; (3) It lacks the tool support for automatically generating formal semantics from intuitive business property representations so as to enable the quick integration between business process modelers and formal verification tools.

The visual notation language of BPSL is extended from PSL by specifically tuning it for business domains. Distinct features of BPSL modeler include: (1) BPSL provides an intuitive representation of business properties such that business people can easily understand them with their existing knowledge and experience; (2) By categorizing frequently used business property templates and providing the "push button" generation of their BPSL definitions, BPSL modeler absorbs existing business knowledge and facilitates business property specification; (3) BPSL modeler supports the auto-generation of underlying formal semantics of each property based on both the logic of CTL and LTL. Consequently, it not only ensures the preciseness of BPSL, but also facilitates the reasoning of business models with existing formal verification tools.

The rest of the paper is organized as follows. In section 2 the framework

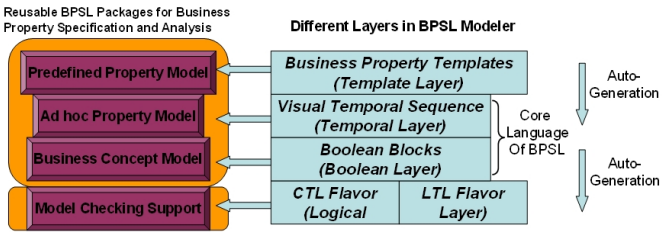


Fig. 3. Framework of BPSL Modeler

and basic concepts in BPSL modeler are introduced to help understand its ideas. In section 4, the features of BPSL modeler are explained in detail together with the analysis of corresponding characteristics in business property specification. Section 5 illustrates how BPSL modeler plays the role in our method of model checking business processes. Section 6 concludes the paper.

3 BPSL Modeler - Framework and Basic Concepts

To better understand the ideas in BPSL modeler for business property specification, figure 3 illustrates its framework. BPSL consists of a Boolean layer and a temporal layer. In Boolean Layer, Boolean Blocks (in short, *BB*) are basic elements for capturing the attributes of different business entities (e.g. activity, resource) and form a basic concept model for specific business domain. In temporal layer, Temporal Sequence (in short, *TS*) is the visual representation that specifies the logical relations between different business entities and the temporal constraints on specific business models. Above the two layers, BPSL modeler concludes frequently used business properties or patterns from business experiences in the form of Business Property Templates. BPSL modeler supports the push button generation of the BPSL definition for each template and in turn the CTL/LTL definition for each *BP* as its formal semantics. Consequently, a benefit of the framework is that property specifications in BPSL modeler can be easily reorganized to form reusable BPSL packages for property generation in different business application domains. Besides, the CTL/LTL formal foundation also facilitates the integration of business property specification with existing formal verification techniques since the reasoning of the two temporal logics have a wide tool support such as RuleBase [2], etc. A full syntax, semantics and visual notations of BPSL can be found in [14].

To better understand BPSL, figure 4 illustrates the BPSL implementation for the previous "*ReportGeneration*" property. The auto-generated formal semantics of this specification with BPSL modeler coincides exactly with the one in figure 2. We will explain the details of the specification in the incoming two sections by investigating the basic concepts and advanced features of BPSL modeler involved in this example.

- ***Boolean Block(BB):*** Represented by an octagon, a *BB* is a three-tuple consists of its name (e.g. *Report*), stereotype (e.g. *Activity*) and a set of

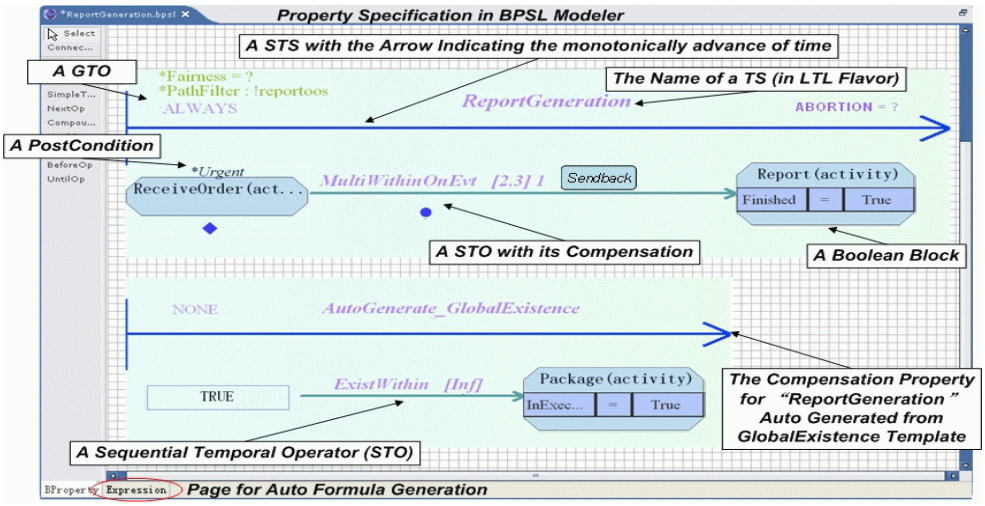


Fig. 4. Re-specify ReportGeneration property in BPSL Modeler

atomic attributes (e.g. whether it is finished). It can thus be used to identify specific business entities that are qualified by the pre-defined conditions in *BB*. For example, the above *BB* of *Report* indicates a *Report* activity that has already been finished.

- ***Simple Temporal Sequence (STS):*** A *STS* specifies the temporal relation among a sequence of *BB*s or business properties along paths were time advances monotonically. As can be found in [14], BPSL supports 14 stereotypes of Sequential Temporal Operators (*STO*s) with different semantics to specify these relations in different situations. While some of the *STO*s have a direct mapping to basic temporal operators (e.g. *Next 1* for *X*, *AllWithin Infinity* for *G*, *PossiblyLeadsto BeforeInfinity* for *EF*, etc), others can be used to express rather complex temporal relations in a simple and compact manner. For example, the property in figure 4 is itself a *STS*. The *STO* of *MultiWithin OnEvt* specifies that when an *Urgent* order is *Received*, a *Report* should be finished for once between the second and the third occurrence of event *Send Back*.
- ***Compound Temporal Sequence (CTS):*** Different from *STS*, a *CTS* specifies the different logical relations (e.g. *And, Or, Not, Imply, IFF*) and predefined temporal relations (e.g. *Before* for the weak Until operator *W* [10], *After* for *F*, *Until* for *U*) between two *STS*s with corresponding Compound Temporal Operators (*CTO*).
- ***LTL and CTL Flavor:*** In order to enhance the expressiveness of BPSL and obtain a wider tool support for formal verification, Temporal Sequence (*TS*s) in BPSL can be interpreted in either LTL or CTL flavor. *TS*s in different flavors possess different available temporal operators in BPSL in order to avoid the confusion of their semantics. For example, the temporal sequence of the above *ReportGeneration* is a LTL flavored specification. As can be found in [14], different *STO*s are associated with different notations

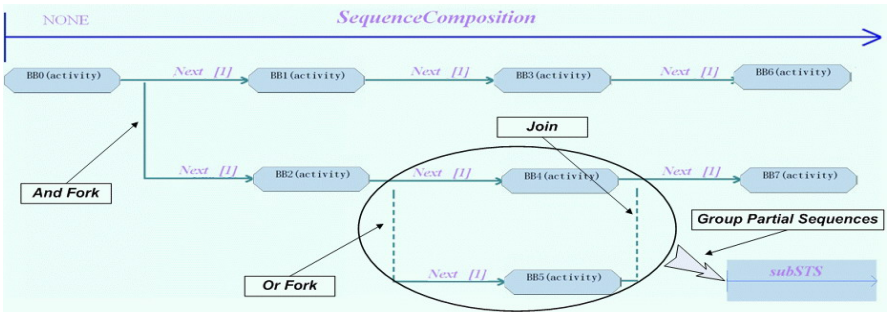


Fig. 5. Temporal Sequence Composition and Grouping

so as to distinguish their semantics in the context of CTL and LTL (e.g. *Within, MultiWithinOnEvent*, etc in LTL flavor and *CertainlyLeadsTo, PossiblyLeadsToBeforeEvent*, etc in CTL flavor). It is then *BPSL* modeler's job to automatically generate the logical formalisms for each property according to its semantics in different flavors and hide the complexity from users.

- ***Global Temporal Operator (GTO):*** Four types of *GTO*s are supported: *"(Possible) Always"* for *G* and *AG(EG)*, *"(Possible) Eventually"* for *F* and *AF(EF)*, *"Repeat"* and *"Never"*. *"Repeat"* and *"Never"* guarantees that the *TS* must hold at least n times or never holds in the business process.
- ***Abortion:*** The abortion condition in a *TS* indicates the circumstance in which the evaluation of a *TS* should be forced to stop (e.g. by an external cancel event). In the above case, the question mark indicates that no abortion condition is explicitly specified.
- ***Postcondition:*** A postcondition can be associated with *BB*s in *TS*. It specifies whether the rest of the *TS* after a *BB* is necessary to be further evaluated. For example, the postcondition "*urgent*" associated with "*ReceiveOrder*" in the example indicates that only when the received order is urgent should the occurrence of report activity be evaluated.

4 Features of BPSL and BPSL Modeler

This section introduces the advanced features in BPSL modeler in accordance with the analysis of some characteristics in business property specification.

4.1 Sequence Composition and Grouping

Business people may not be familiar with logical reasoning, but most of them are familiar with popular process modeling techniques like UML Activity Diagram. Therefore, in order to make property visualization in BPSL more acceptable to business people by exploiting their existing experience, BPSL properties are also presented in a "process" oriented form. That is, Temporal Sequence (*TS*s) in BPSL not only specifies the temporal relations between two *BB*s or *BP*s, but can actually be composed to form a long chain of property

sequence with And-Fork, Or-Fork and Join relations. Figure 5 illustrates such an example. The fork, join operators specify the different and/or relations when evaluating each branch in the *TS*. Besides, in BPSL the grouping of a partial *TS* is also supported which corresponds to the concept of sub-process in business process model. In figure 5 the details "*subSTS*" are hidden by grouping the corresponding parts in the original *STS*. Sequence composition and grouping enables the abbreviation of complex business property specifications in BPSL and flexibly adjusting the granularity of property representation.

4.2 Property Compensation

Traditional logical operators often have their relaxed versions to answer the question of "whether a property is still satisfied if certain condition does not hold". For example, *"P W Q"* is the relaxation of *"P U Q"* in that the weak until operator W does not order that Q must occur in the model while U does. In BPSL, such conditions like Q, which decide whether a property is satisfied strongly or weakly, are called Compensation Conditions (*CC*s). BPSL further generates the idea of property relaxation by enabling the flexible association of Compensation Properties (*CP*s) with the CC that each *STO* and *CTO* in BPSL may have. More specifically, a process model *PM* satisfies a business property *BP* with *(CC, CP)* iff $PM| = BP$ or $PM| = G(\neg CC) \wedge CP$. As a result, a strongly/weakly held property *BP* is a special case for property compensation in BPSL when its *CP* is *False*(as denoted by a rectangle)/*True*(as denoted by a diamond). For example, in figure 4 the *CC* for *MultiWithin OnEvt* operator is that there must be a third occurrence of event *Send Back* in the process. Consequently, the property of "$AutoGenerate_{G}lobalExistence$" which serves as its *CP* (as denoted by a circle) further specifies the rest of the semantics for *ReportGeneration* property that in case there is never a third occurrence of *Send Back*, a *Package* activity will be eventually executed. Compensation mechanism in BPSL effectively allows business analysts to specify their requirements by connecting different properties in their familiar *"if..then..else"* fashion. A reference of predefined *CC*s with temporal operators in BPSL can be found in [14].

4.3 Filtering and Fairness

Business processes can be rather complex so as to provide a full view of the daily businesses in an enterprise. Therefore when reasoning a business process it is often desired that redundant information (e.g. exception handling) which may not be the primary concern for business analysts can be neglected to avoid reaching wrong conclusions. Thus it will be of great help if the granularity of business reasoning can be flexibly controlled in the step of business property specification. BPSL provides this mechanism by supporting filter and fairness conditions. Borrowed from model checking [4], the fairness condition specifies that the evaluation of business property will only be done on process execution

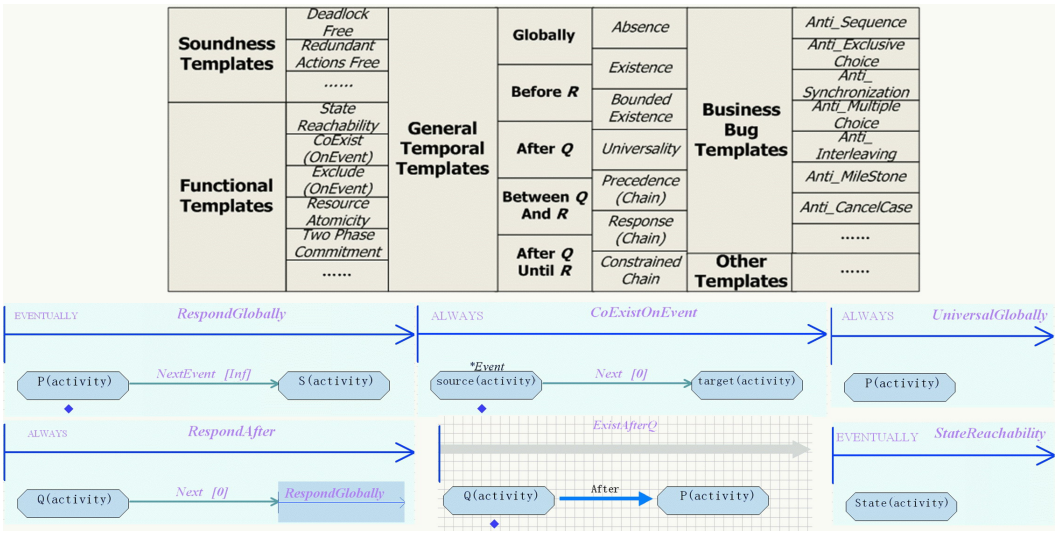


Fig. 6. Business Property Templates and Implementation Example

paths where it can hold infinitely often. On the contrary, the filter condition specifies that all the execution paths in the business process along which the filter condition may be satisfied will be neglected in the property evaluation. For example, while the example in figure 4 does not contain any fairness condition, its filter condition implies that the *ReportGeneration* property will not be evaluated in unusual cases where *Out Of Stock* may happen.

4.4 Business Property Templates

Years of practices and researches in business domain form a considerable accumulation of relevant experiences. BPSL modeler provides the capability of capturing these existing knowledge in the form of business property templates to facilitate both property specification and understanding. Four predefined common categories of business property templates (Soundness Templates, General Temporal Templates, Business Bug Templates and Functional Templates) are concluded in BPSL modeler based on which frequently used business patterns can be automatically generated and reused. The definition of these business property templates can be automatically implemented within the expressiveness of BPSL and can in turn be formally interpreted in CTL/LTL with BPSL modeler. With the size limitation, figure 6 illustrates several examples of these templates and their implementations.

Soundness template concludes the common workflow soundness [8] definitions that each business process may satisfy; General temporal template implements the general temporal patterns based on the survey result in [9] and their semantic mapping [10] on CTL/LTL; Business bug template corresponds to traditional workflow patterns [12]. Each bug template is used to falsify a true realization of a specific workflow pattern in the business process model; Functional template captures some useful functional requirements in

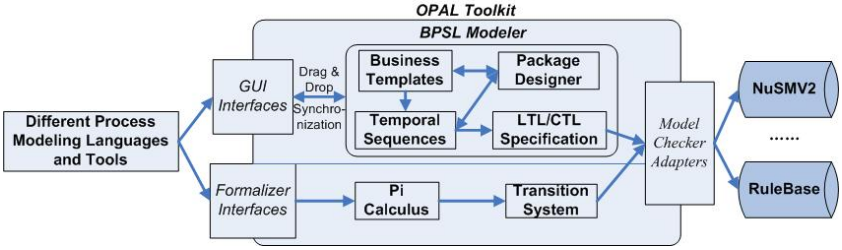


Fig. 7. BPSL Modeler as a Component for OPAL Toolkit

business processes. For example, *ResourceAtomicity* can be used to impose constraints on the process such that "for certain kind of resource (like money), it should never be created or destroyed in the process".

5 BPSL Modeler as a Component for Model Checking

BPSL Modeler is a key component of our OPAL (Open Process AnaLyzer) toolkit for model checking business processes (figure 7). In OPAL, business processes in different modeling techniques are formalized with Milner's Pi Calculus through standard Formalizer Interfaces. On the other hand, a set of GUI Interfaces are provided to connect property specification in BPSL modeler with process modelers. For example, in our current implementation, the GUI/Formalizer interface for Websphere Business Integrator (WBI) is provided such that not only the semantics of business processes modeled with WBI can be automatically formalized, but also Boolean Blocks in BPSL can be directly generated by drag-and-dropping different WBI elements into BPSL modeler. These Boolean Blocks are then synchronized with corresponding elements in WBI so that they are always referencing to the same business entities. The relations between Boolean Blocks can either be generated from templates, or built directly with Temporal Sequences. In BPSL Modeler, a Package Designer is further implemented to store and edit user customized business properties and templates as a knowledge base for their later reuse. Available model checkers can thus be integrated with OPAL by transforming he auto-deduced transition system for process models and the LTL/CTL formulas into the format they accept through Model Checker Adapters.

6 Summary

Business property specification is a major step in reasoning business process models and ensuring its reliability. In this paper, the visual notation language of BPSL and its supporting tool BPSL modeler are proposed to enable the intuitive specification of business properties and facilitate the reasoning of business process models. BPSL modeler simplifies the complexity of business property specification by taking different business characteristics into consideration, e.g. the visual representation, property compensation and filtering,

etc. It exploits existing knowledge in the practices and researches in business domain to make property specification of minimum efforts. In BPSL modeler, both the logics of LTL and CTL are supported and formal semantics can be auto-generated for each business property template and business property so as to ease the integration between BPSL modeler and existing formal verification tools. Our future work will include extending the application of BPSL modeler into more real cases to enlarge its values.

References

[1] Bard, B. *Seeing by Owl-Light - Symbolic Model Checking of Business Requirements*, Technical Report, IBM T.J. Watson Research Lab, 2002.

[2] Beer, I.S., B. David, et al. *RuleBase: Model checking at IBM*, International Conference on Computer Aided Verification, 1997, 480-483.

[3] Brambilla, M., A. Deutsch, et al. *The role of visual tools in a Web application design and verification framework: A visual notation for LTL formulae*, Lecture Notes in Computer Science, **3579**, 2005, 557-568.

[4] Clarke, E.M., Jr. O G., D. A. Peled, *Model Checking*, MIT Press, 1999.

[5] Del, B., A. L. Rella, E. Vicario, *Visual specification of branching time temporal logic*, Proc. 11th IEEE Symp. on Visual Languages, 1995, 61-68.

[6] Geist, D., *The PSL/Sugar specification language a language for all seasons*, Lecture Notes in Computer Science, **2800**, 2003, 3.

[7] Giblin, C., Y. Liu, et al. *Regulations Expressed As Logical Models (REALM)*, 18th Annual Conference on Legal Knowledge and Information Systems, IOS Press, Accepted, 2005.

[8] Hofstede, A., M. Orlowska, J, Rajapaske. *Verification problems in conceptual workflow specification*, Data and Knowledge Engineering, **24(3)**, 1998, 239-256.

[9] Matthew, B.D., S. A. George, C. C. James. *Patterns in Property Specifications for Finite-State Verification*, Proc. 21st International Conference on Software Engineering, 1999.

[10] Property Patterns, 2005, http://patterns.projects.cis.ksu.edu

[11] Rao, A. C., A. Cau, H. Zedan, *Visualization of Interval Temporal Logic*, Proc. 5th Joint Conference on Information Sciences, 2000, 687-690.

[12] van der Aalst, W.M.P, ter Hofstede, A.H.M, et al. *Workflow patterns*, Distributed and Parallel Databases, **14**, 2003, 5-51.

[13] Wang, W. L., Z. Hidvegi, et al. *E-process design and assurance using model checking*, Computer, **33(10)**, 2004, 48-53.

[14] Xu, K, L. Ying, W. Cheng. *A Property Specification Language for Verifying Business Process Models*, IBM Technical Report, RC23830(C0512-005), 2005.

Simulation and Formal Analysis of Workflow Models

Máté Kovács [1,3], László Gönczy [2,3]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary

Abstract

We present a framework for the simulation and formal analysis of workflow models. We discuss (i) how a workflow model, implemented in the BPEL language, can be transformed into a dataflow network model, (ii) how potentially incorrect execution paths can be incorporated, and (iii) how the properties of a workflow can be formally verified using the SPIN model checker. For the several model transformation steps from workflow to analysis models, we use graph transformations.

Key words: BPEL, Workflow, Verification, Fault simulation, Dataflow Networks

1 Introduction

In the past data was kept on paper. A piece of paper, containing information, could be interpreted as sort of a token flowing through the basic activities. This kind of a paper is called the *work item*. The colleagues carry out *activities* on work items. The *workflow* comprises all the activities. As such, the workflow defines the order in which the activities have to be carried out.

Today offices have significant IT infrastructure to enhance the efficiency and productivity often using computer aided business process coordination. There are several languages that allow a very high-level, executable description of workflows, for instance, BPEL (Business Process Execution Language) [11] or XPDL (XML Process Definition Language) [13]. The complexity of workflows is close to that of regular programming languages. Therefore, new problems arise with electronic business process execution.

[1] Email: km432@hszk.bme.hu
[2] Email: gonczy@mit.bme.hu
[3] This work was partially supported by the SENSORIA European project (IST-3-016004).

Computer-based workflow execution involves communication between loosely coupled information systems. This makes the testing of distributed workflows very difficult as data taken from several databases is often required to be manipulated. Moreover, the side effects of the transactions generated in a test phase need to be undone. Another choice is to establish the entire test environment with multiple servers and databases containing the test data. Both solutions are time consuming and expensive.

As a consequence, there are several semantic requirements that a workflow has to meet before enactment, such as:

- There must not be any deadlock. In case of a deadlock the workflow execution would come to a halt.
- All activities have to be reachable. In case of unreachable activities there might be unused resources in the company, which is far from desirable.
- Each variable has to be written before being read. Reading an uninitialized variable could lead to an unpredictable result.

The most important contribution of the framework is the ability to check the last requirement of those above mentioned. There are other solutions to verify the first two [1,7].

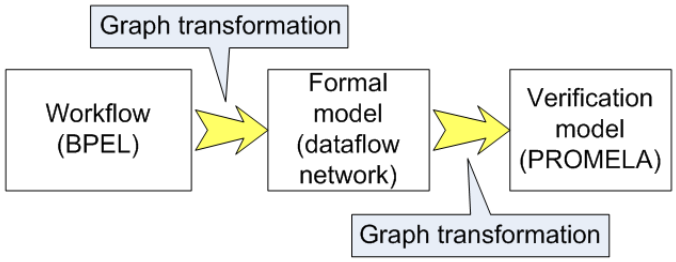


Fig. 1. The workflow analysis method

In this paper we discuss a method to formally verify the above mentioned properties of a BPEL model. As Fig. 1 illustrates the high level description of workflows, such as BPEL, needs to be transformed into a low level mathematical notation which can be verified automatically. We have chosen dataflow networks [2] for this purpose.

The formalism of dataflow networks is meant to model complex, distributed computing systems with well defined semantics. The abstraction level of this formalism is between the BPEL model and the verification model, the PROMELA implementation. It combines the state based description of finite state automata, and the data (token) flow of Petri nets. A further advantage of modeling with dataflow networks is that with additional rules and states of nodes, *fault simulation* can also be performed.

Workflows can be checked against requirements such as mentioned above. The properties of the process that are to be checked need to be formulated as linear temporal logical expressions (LTL), regarding the PROMELA (Process

Meta Language) [12] implementation of the dataflow network representation of the workflow. The SPIN model checker [12] will evaluate these LTL expressions.

2 From workflow models to dataflow networks

2.1 Workflows

Workflow description languages resemble very much to regular structured programming languages. The structural elements of workflows are the sequence, selection, iteration and parallel execution constructs.

Activities have input and output parameters. These data elements are called *messages* that are passed between different computers running in a distributed environment.

In BPEL data is maintained using variables. The value of variables may be sent and received as messages, and the control flow may be determined by them. The manipulation of the variables is called data handling.

We assume that the workflow to be checked is implemented in BPEL, using only a subset of the language. The transformation deals with the *basic* and *structured activities* of BPEL - like those in Fig. 2 - and data handling but all kinds of event handling is ignored.

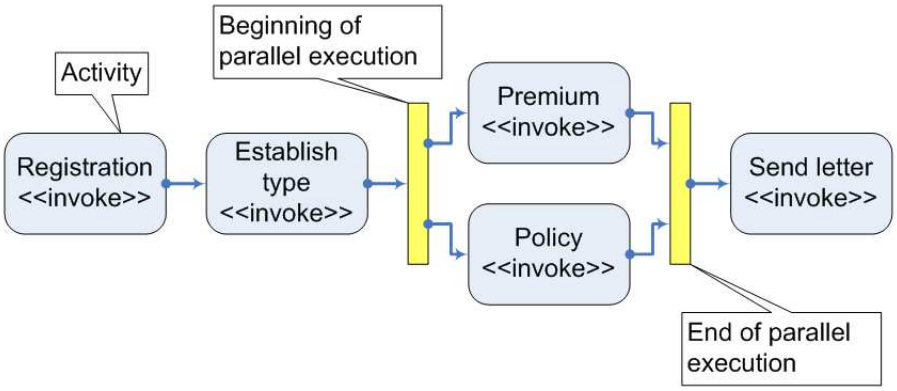


Fig. 2. Workflow concepts and corresponding BPEL keywords

A small fragment of the workflow in an insurance company is shown in Fig. 2. First, the client reports the damage which is recorded. Then the type of the damage is established. Next the insurance company has to decide whether to compensate the damage or not. This needs two independent activities that can be executed simultaneously. Finally a letter is sent to the client containing information about the decision.

2.2 Dataflow networks

Dataflow networks are designed to model distributed communicating systems. A dataflow network consists of data processing *nodes* interconnected with

channels. Channels transmit *tokens* between nodes. A channel does not contain the token queue. In contrast of the original [2] dataflow network formalism, we define *ports* which contain the token queue. A port is the connection between a node and a channel. A token remains in the input port until it is not removed during the application of a firing. A port can potentially contain an infinite number of tokens. A token is an atomic abstract data unit represented by its color. Each node is a finite state automaton that has *states*, and state transition *rules*. A rule consists of two parts. The first defines the *firing condition*, the second declares the *action* that should be performed in case of a firing. During the transition the node removes the tokens according to the condition, changes the state, and puts several tokens to the output ports.

2.3 Mapping workflows to dataflow networks

In the dataflow network model the control flow is represented by tokens of a special color: the *control tokens*. The number of control tokens in the network corresponds to the number of activities executed in parallel.

The execution order of activities can be defined by the structured activities: sequence, selection, iteration and parallel execution. In the dataflow network model they are represented by at least two nodes. The one at the beginning distributes the control tokens, the final one collects them representing the synchronization of the branches. The control constructs - that perform a selection - need additional nodes to represent the evaluation of the conditional expressions.

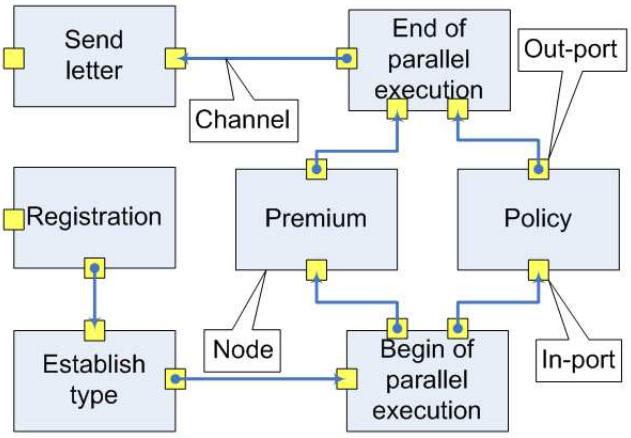


Fig. 3. Mapping the workflow in Fig. 2 to dataflow network

Applying the transformation rules on the workflow in Fig. 2 results in a dataflow network illustrated in Fig. 3. The rules are summarized in the table below.

workflow pattern	dataflow network pattern
variable	a node with appropriate state space (see Fig. 4)
basic activity	two nodes if the activity has in and out parameters, otherwise one node
sequence	two extra nodes, one at the beginning and one at the end
parallel execution	one start node multiplying the control flow, and one end node restoring the single control flow
selection (switch)	one node at the beginning and one at the end, the cases linked to each other according to the order of condition evaluation
a *case* in a selection	a node for each variable in the expression of the condition
iteration (while)	a modified selection with two branches: one leading to the activity being iterated and then back to the condition, the other leading to the next activity in the workflow

The transformation does not preserve the concrete values of variables. We only use an abstract state space to check the order of the variables being initialized, read, and written.

3 Fault simulation

A workflow execution engine can coordinate the work of many people, and it is usually connected to multiple computers of independent organizations. These computers are loosely coupled with no guarantee on their availability. Thus failures have to be considered. It is reasonable to add some redundancy to the workflow and then to check whether the planned fault tolerance was reached.

Error propagation [8] in the control flow is modeled with tokens of a specific color: the *faulty control tokens*. The abstract state of variables (i.e. "Written", "Written and read", etc.) is represented by the states of nodes.

3.1 Simulation of dataflow errors

In this case the error is only spreading across the variables but it does not have an effect on the control flow.

Fault injector activities are needed which write faulty data to their output

regardless of the input and the color of the control token they received. We assume that healthy nodes always write healthy data to their output unless the control or the input is erroneous. This way the error confinement region of a fault injector variable can be determined.

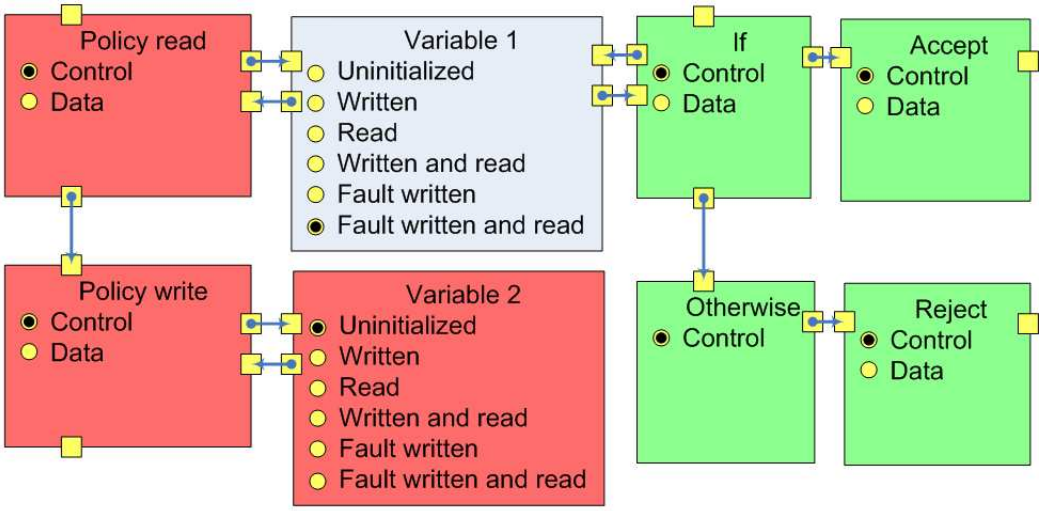


Fig. 4. Fault spreading

In Fig. 4 a small fragment of the dataflow network representation of the insurance company's workflow is shown. Variable 1 contains faulty data. The red nodes are involved with spreading the error among variables, the green nodes deal with control flow infection.

1. In the first step "Policy read" receives a control token in its input port. Changes the state "Control" to "Data" and sends a token of color *reading* to its input variable.
2. The reading token is received by the node representing the input variable (Variable 1). Now we assume that the variable contained faulty data i.e. the input variable's state was either "Fault written" or "Fault written and read". The state is switched to "Fault written and read" and a faulty control token is placed on the output port.
3. A faulty control token is received from Variable 1. The state is switched from "Data" to "Control" and a token of color *faulty write* is put on the output port towards the second node of the activity.
4. "Policy write" receives the faulty write token from its control port, switches the state from "Control" to "Data" and sends a faulty write token to its output variable.
5. "Variable 2" switches to "Fault written" state and sends back a control token to the second node of the activity.
6. The second activity node receives the control token from the output variable, switches its state to "Control" and sends a control token to its output port.

3.2 Simulation of control flow errors

If a branching condition is evaluated using a faulty variable, the fault of the condition variable infects the control flow. This is illustrated by node "If" in Fig. 4.

1. The node called "If" is part of all the nodes of the switch construct. It receives a control token, then it changes the state from Control to Data and sends a reading token to the node called "Variable 1".
2. The state of the node representing a variable is changed from "Faulty written" to "Faulty written and read" and a faulty control token is sent back to the node of the selection construct.
3. The faulty control token is received by node called "If". It chooses non-deterministically whether to pass the token to the activity called Accept, or to the next condition.

   At this point the control token is changed to faulty control, all the activities that receive the faulty control token write faulty data and pass on faulty control token. Let us suppose that the faulty control token is sent to activity Accept.
4. The faulty control token is received by Accept, faulty data is written as it is shown in Section 3.1, and the faulty control token is passed on.

4 Verification of workflow models

As usually in structured programming languages, in PROMELA too we can use variables and subroutines called proctype. The types of variables are subsets of integer in order to guarantee that the program has finite state space.

The PROMELA provides a further type of variables, the FIFO channels. This way the channels of dataflow networks do not have to be implemented.

The SPIN is capable of exhaustive state space examination of a PROMELA program evaluating system requirements in the form of LTL expressions. This way the dynamic properties of a dataflow network can be verified.

The transformations are property preserving in a sence that every execution path of the BPEL process can be found in the PROMELA program too. If a property is valid, concerning the PROMELA model, then it holds for the BPEL process as well.

4.1 From dataflow network models to PROMELA implementation

Dataflow network constructs are mapped into PROMELA language patterns.

- Channels in the dataflow network are mapped into the channels of the PROMELA language. This can be done since there is always at most one token in a channel.

- Tokens are mapped into symbolic constants.
- State variables of a node are mapped into global integer type variables.
- Each node is mapped into a proctype construct. The initial state is set by the first instruction. The state transition rules are implemented by an infinite iteration containing the rules as conditional atomic sequence of instructions.
- A state transition rule of a node is mapped into an atomic sequence of instructions.

4.2 The verification

In Fig. 3 the activities called Premium and Policy are executed simultaneously. An interesting question is, whether the fault of one activity effects the other. The requirement needs to be formulated as LTL formula, the logical variables that take part in the formula have to be defined in form of C style define macros. Using the PROMELA source and the logical variable definitions the SPIN evaluates the LTL formula.

Example 4.1 Here we demonstrate how to formulate the requirement which states that the fault of activity "Policy out" should have no effect on "Policy in" regarding the model illustrated in Fig. 3.

```
#define Policy_out state_Policy_output!=Fault_written
#define Premium_in state_Premium_input!=Fault_written
G(!Policy_out -> Premium_in)
```

The first two lines define the meaning of the logical variables contributing to the value of the logical expression in the third line. Variable "Policy_out" is true, if and only if the value of the PROMELA variable "state_Policy_output" is "Fault_written". The meaning of "Premium_in" is defined analogously. The LTL formula of the requirement is shown in the third line. It states that at any state along the discrete time-line where "Policy_out" is false, "Premium_in" is true.

The evaluation of a formula, such as the one shown in Example 4.1, can result in positive and negative answers. The positive result guarantees that the PROMELA model meets the requirement. The negative may be a good test case of the BPEL process. However, because of the concrete values of BPEL variables, the process may not be able to run into the faulty execution path.

5 Related work

There is much research done about the formal verification of workflows implemented in several languages. In [9] a BPEL process is directly transformed into PROMELA code, the requirements are verified by SPIN. This approach

is similar to ours. The major difference is that we use an intermediate formal model of workflows, dataflow networks. This allows us to implement the workflow - PROMELA transformation in two steps, each a smaller step in abstraction level.

In [1] the formal semantics of workflow models are defined by Petri nets. This approach focuses on the syntactic properties of workflows but the insertion of structural flaws, such as hanging paths resulting in unnecessary tasks is prevented by the XML validation of the above mentioned languages. The authors of [7] discuss a design procedure to transform a business model of the workflow into the IT model that requires the decisions of engineers. The IT model is represented by Communicating Nondeterministic Automata that can be analyzed with NuSMV [5] model checker. As it is shown in [10] colored Petri nets can also be used as a formal model of workflows. When modeling with dataflow networks, we also have the advantage of the formalism's compositionality.

Another approach is presented by [3] to enhance the quality of a workflow by runtime monitoring. This technique could be successfully applied in a business environment with services and processes changing frequently. Our verification method is meant to be used at design time but could also be helpful to verify the implementation of a small change in an already enacted business process.

6 Conclusion

In this paper we proposed a method to check correctness properties of workflows implemented in BPEL. Dataflow networks are used to define the formal semantics of the workflow. The BPEL model is mapped into dataflow network, the dataflow network is mapped into a PROMELA model.

The model transformations, creating the dataflow network model of the workflow and generating the PROMELA code, are implemented as graph transformations executed within the VIATRA2 (Visual Automated Transformations) framework [14]. The VIATRA2 combines the procedural and declarative programming paradigms, enabling the efficient formulation of the implementation of model transformations.

The source of the transformation illustrated in Fig. 3 contains 42 graph patterns and several ASM rules. There is a graph transformation rule for each important construct of the BPEL language. The dataflow network - PROMELA transformation consists of 32 graph patterns.

In the future we plan to support the automated generation of LTL expressions from the requirements formulated in the BPEL domain. Furthermore, the automatic back annotation of the counterexample, presented by the SPIN in case of a negative result, is also a future goal.

References

[1] Wil van der Aalst, Kees van Hee, "Workflow Management", The MIT Press 2002.

[2] A.J. Anderson, Data Flow Systems, In *Multiple Processing: A System's Overview*, ch. 10. pp. 441-488. Prentice Hall, UK, 1989.

[3] L. Baresi, S. Guinea. *Towards Dynamic Monitoring of WS-BPEL Processes*, Proceedings of the 3rd International Conference of Service-oriented Computing (ICSOC'05). Amsterdam, The Nederlands, Lecture Notes in Computer Science, volume **3826** (2005), pages 269 - 282.

[4] E. Börger and R. Stärk, "Abstract State Machines, A method for High-Level System Design and Analysis", Springer-Verlag, 2003.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, *NuSMV: a new Symbolic Model Verifier*, In Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99), number **1633** (1999) in LNCS, pages 495-499. Springer

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer, *Model-based verification of web service compositions* IEEE ASE 2003, Montreal, Canada

[7] J. Koehler, G. Tirenni, S. Kumaran, *From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods*, EDOC 2002, pages 96-106.

[8] J. Laprie, B. Randell, C. Landwehr, *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Transactions on Dependable and Secure Computing, (Vol.**1**, No.**1**) (2004) pp. 11-33

[9] S. Nakajima, *Verification of Web service flows with model-checking techniques*, presented at First International Symposium on Cyber Worlds, 2002.

[10] Y. Yang, Q. Tan, Y. Xiao, *Verifying Web Services Composition Based on Hierarchical Colored Petri Nets*, Proceedings of the first international workshop on Interoperability of heterogeneous information systems Bremen, Germany Pages: 47 - 54

[11] Specification of the Business Process Execution Language Version 1.1. 2003: ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[12] Simple Promela Interpreter: http://www.spinroot.com

[13] XML Process Definition Language Version 2.0.: http://www.wfmc.org/standards/XPDL.htm

[14] VIATRA2 Eclipse GMT subproject: http://www.eclipse.org/gmt

[15] XSD Schema of the BPEL language Version 1.1.: http://schemas.xmlsoap.org/ws/2003/03/business-process/

The York Abstract Machine

Greg Manning [1] Detlef Plump [2]

Department of Computer Science
The University of York, UK

Abstract

We introduce the York Abstract Machine (YAM) for implementing the graph programming language GP and, potentially, other graph transformation languages. The advantages of an abstract machine over a direct interpreter for graph transformation rules are better efficiency, use as a common target for compiling both future versions of GP and other languages, and portability of GP programs to different platforms.

Key words: Graph transformation; GP; abstract machines; nondeterminism; backtracking

1 Introduction

The graph programming language GP [6] consists in its core of just three constructs: application of a set of conditional rule schemata either (1) in a single step or (2) as long as possible, and (3) sequential composition of programs. This language is computationally complete (see [5]) and has a simple formal semantics. In this paper, we present a low-level abstract machine for graph transformation—the York Abstract Machine (YAM)—that will be used to implement GP.

A major advantage of a low-level abstract machine over a high-level interpreter for graph transformation rules is higher speed. The instructions of the abstract machine are typically simple stack operations which, after a GP program has been compiled, need not analyse the left- and right-hand graphs of a rule over and over again. Instead, the analysis of rules is performed once and for all when GP programs are translated into YAM bytecode.

The YAM can also serve as a common target for compilers of both future versions of GP—a language still under development—and, potentially, other graph transformation languages. Moreover, the YAM will support the portability of GP programs because they can be compiled to bytecode with any available compiler and then executed on every platform on which a YAM implementation exists.

[1] Email: gm@cs.york.ac.uk
[2] Email: det@cs.york.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

To give an example of YAM code, consider the GP program `VertexColouring` of Figure 1. The program labels all nodes of an input graph with colours (integers) such that any adjacent nodes have different colours. To achieve this, first the rule `Init` is applied as long as possible in order to label all nodes with colour 1. Then the `Inc`-rules nondeterministically increment colours until any adjacent nodes are differently coloured.

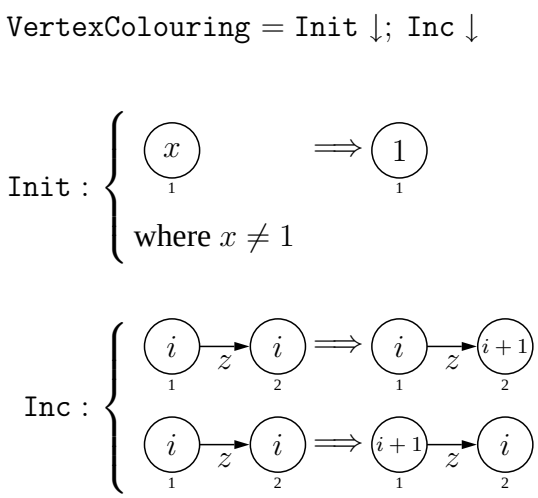


Fig. 1. GP program `VertexColouring`

A full version of (hand-compiled) YAM code for this program is presented in the appendix. Here we only consider the code for the conditional rule `Init`:

```
NA
Dup Get_node_label ATOI 1 Equals Not Assert
"1" Relabel_node
```

The purpose of these instructions can be described as follows: "Find any node, check its label is not 1, and relabel it to 1".

The first instruction, `NA`, pushes a node identifier onto the stack. If some later instruction fails and the machine starts backtracking, this instruction will subsequently retry with the next node (for some, usually random, ordering of nodes) until all nodes have been tried. The sequence from `Dup` to `Assert` checks that the label is not 1, in the following manner: `Dup` duplicates the top of stack; [3] `Get_node_label` pops a node identifier from the top of stack and pushes the label of that node (a string); `ATOI` converts the top of stack from a string to an integer; `1` pushes the integer value 1 onto the stack; `Equals` pops two values from the stack and pushes 1 if they are equal, 0 otherwise; `Not` pops the top of stack and pushes 1 if it is 0, 0 otherwise; and `Assert` pops the top of stack and starts back-

[3] Since stack instructions are almost always destructive, it is often necessary to save values by duplicating them.

tracking if it is 0. The final sequence, `"1" Relabel_node`, relabels the node with the string "1". (The identifier of the node to relabel is still on the stack, thanks to the earlier `Dup`.)

The next section describes the YAM language and its execution. Section 3 further discusses the compilation of GP to YAM Code. In Section 4 we explain the graph data-structure used to implement the YAM, Section 5 focuses on the three stacks the machine is based on. Section 6 briefly addresses the relation of the YAM to graph transformation languages other than GP, and Section 7 gives some concluding remarks and a few topics for future work.

2 The YAM language

A program in the YAM language (the assembly) is, at its most basic, a sequence of instructions separated by whitespace. It is also possible to use labels and macros, as discussed below.

A *program label* is declared in the form `LabelName:` — the name of the label followed by a colon. Whenever it is found in the source code, the integer value of the program counter at the definition is pushed onto the data stack. This value is typically then used to jump control to that program counter (via a `Jump`, `ALAP` or `Choice` instruction).

YAM code can also contain *macros* which are simply named sequences of instructions. An example of a macro definition can be found in the code for `Vertex-Colouring`:

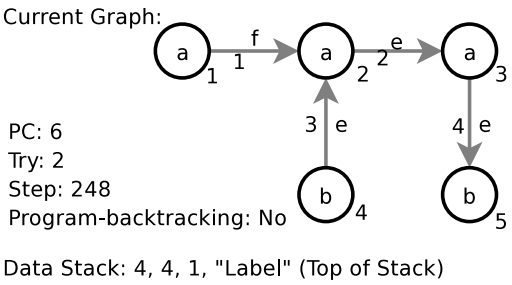
```
:EXXA NA Dup Get_node_label Swap EILA ;.
```

This is a macro named `EXXA` which finds an edge between two nodes with the same label as follows: Find any node, find its label, then find an edge from that node to a node with that label. Whenever a macro call is found in the code, it is replaced verbatim with the body of the macro (macro expansion). The *prelude* is a useful collection of macros.

The YAM instructions alter the state of the machine, which consists of a single graph, three stacks—data stack, choice stack and graph change stack, several integer values—current program counter (PC), current step, current try [4] —and a flag to control the behaviour of backtracking (explained later in this section). For example, Figure 2 shows a YAM state in the middle of a computation. There have been two changes to the graph since the machine started, represented by two frames on the graph change stack. Also, there have been two decisions made, represented by the two frames on the choice stack. The top two elements on the stack are the string "Label" and the integer 1. The next instruction for this machine might be to relabel node 1 from 'a' to 'Label'.

The graphs which the YAM uses are directed, labelled graphs. The labels of edges and nodes are strings. Parallel edges are permitted, as are self loops.

[4] The try number is the number of times the current instruction has been attempted.



Graph Change Stack: (Top of Stack)	Choice Stack: (Top of Stack)
PC:3,Step 61, Added edge 4.	PC:2,Step 60, try 1, Data Stack: 4
PC:0, Step 0, Added node 5.	PC:1,Step 59, try 3, Data Stack:

Fig. 2. An example of a YAM state

There are currently three types of data that can go on the data stack: strings, integers and boolean values. Booleans are represented by integers: non-zero integers represent logical 'true' and zero represents 'false'. Strings can be converted to integers and integers to strings using the instructions `ATOI` and `ITOA`, respectively.

There are currently about 70 instructions which the YAM can execute. [5] They fall into four main categories:

(i) Data stack-only instructions such as `Add`, `Multiply`, `Swap` and `Duplicate`.

(ii) Control instructions, which delimit rules, explicitly represent choices or mark deterministic sections of code, such as `ALAP`, `Once` and `Cut`.

(iii) Graph query instructions such as `NA`, `EILL` and `Get_edge_start`.

(iv) Graph modification instructions such as `Add_node` and `Del_edge`.

Data stack-only instructions manipulate the data stack in a simple way, having no effect on the other stacks or the current graph. Control instructions influence the choice stack or the program counter. Graph query instructions modify the choice stack where they have returned one of many answers, and push their answer onto the top of the data stack. Graph modification instructions modify the graph change stack (recording data so that the graph change can be undone) and the current graph.

The YAM provides backtracking to implement the nondeterminism of graph transformation programs. Certain instructions involve the machine choosing which answer of several to return. If the machine gets the answer wrong in that some later part of the program fails, then the choice needs to be revisited and a different answer chosen. Stacks are used to remember these choices so that they can later be reconsidered.

There are two different types of backtracking. *Matching backtracking* is concerned with making the right choices of nodes and edges to find an individual match for a rule. *Program backtracking* deals with choosing the right rule to apply from a set, and determining when an as-long-as-possible operator has finished. The program-backtracking boolean flag in the YAM state is to record which type of backtracking is currently running.

[5] The full list of YAM instructions is available at
http://www-users.cs.york.ac.uk/~gm/YAM/instructions.html

The YAM currently has two modes of operation: *one-result* and *all-results*. In either case, the machine runs until the first result is found. If it is in one-result mode then it terminates, if it is in all-results mode then it imposes a failure and finds the next result. It continues in this way until it finally backtracks past the first choice. In such a manner, the machine will find all results in the search space, so long as there are no infinite computation paths. If there are infinite computation paths then the machine will enter such a path at some point, possibly before producing all results or any result at all.

3 Compiling GP to YAM code

Although a compiler is still under development, we can make a few general remarks about the process of compiling GP to YAM code. First, individual rules need to be translated into fragments of YAM code. These fragments can then be put together with appropriate control instructions between them to mark them as sets of rules and iterated or single-step applications.

Compiled GP programs are executed using a depth-first strategy. In order to compute a result, rules must be matched and applied until the end of the program is reached. To apply each rule, an instance of the left-hand side must be found in the current graph (this is the subgraph isomorphism problem), and any conditions of the rule must be checked. At GP to YAM compile time, this problem is broken down into smaller problems of finding individual nodes and edges with certain characteristics (such as "a node with label l" or "an edge from node 3"), and asserting conditions (such as "the label is not 1"). The correct choices (if any exist) for each of these small decisions will lead to a result for the program and current input graph. The correct choices are found by means of backtracking, in a manner very similar to that of the implementation of Prolog by the Warren Abstract Machine [1]. Failure and backtracking occurs when an instruction runs out of answers, or an `Delete_node`[6] or `Assert` instruction fails. The machine then revisits the previous choice made and resumes execution with the next option for the most recent choice.

In general, individual graph transformation rules are implemented by the following pattern of YAM instructions:

(i) Find some nodes or edges from the left-hand side of the rule.

(ii) Check they fulfill any conditions.

(iii) If a full left-hand side has not yet been found then goto (i).

(iv) Execute the changes specified by the rule.

`Assert` instructions in the condition checks make sure that only nodes and edges are selected that match the rule. Any other selection will trigger backtracking.

[6] Nodes can only be deleted if there are no edges incident to them.

4 Implementing the YAM

In our implementation of the YAM, we only store the current graph. It is the purpose of the graph change stack to recover older versions of the graph. Each frame on the graph change stack describes one change that has been made to the graph in such a way that it is possible to undo that change. When backtracking occurs, the choice stack and graph change stack can be unwound in parallel, recovering older graphs as older choices are revisited. It would be infeasible to store entire graphs for choice points, given the number of choices involved in a typical program.

The YAM calls for a graph data-structure with very quick query operations because these will typically be used many more times than update operations. Our implementation achieves this by a quite complex representation of graphs, involving node and edge structures, hashtables and ordered lists.

An edge structure consists of an identifier (a unique integer), the label of the edge, and the identifiers of the start and end node of the edge.

A node structure contains an identifier and the node's label, the indegree and outdegree of the node, and four hashtables, viz. the inedges and outedges indexed by node label and edge label. [7] In these hashtables, the keys are labels and the values are ordered lists of integers (edge identifiers). Because they are ordered, taking the intersection of two lists is a very quick operation. (For example, taking the intersection of those outedges where the target node is labelled "n" and the edge is labelled "e".)

The graph data-structure has two integer-valued functions, *next unused node identifier* and *next unused edge identifier*, which provide fresh identifiers. The structure also has four hashtables. Two tables are the actual stores for nodes and edges, they have identifiers as keys and node or edge pointers as values. The other two tables are mappings from labels to ordered lists of identifiers for nodes and edges (similar to those in the nodes).

Using this structure, graph updates are quite slow—the slowest operation is relabelling a node, which requires time proportional to the size of the neighbourhood of the node being relabelled. But instructions which query the structure of the graph are very quick, and instructions with multiple answers return their results in a fixed order for a given graph. It is possible to compute the i^{th} result of the instruction "A node with label n", or "An edge from node 1 with label e" quickly: the first example requires one hash lookup, and then i steps down the ordered list; the second requires two hash lookups (one to find the node pointer, and one to find the ordered list of edges) and then i steps down the ordered list.

[7] That is, inedges by node label, inedges by edge label, outedges by node label and outedges by edge label.

5 Stacks

All components of a YAM state other than the current graph, the program counter and the try and step numbers are maintained in stacks. There are three distinct stacks, although the data stack gets copied and saved as part of the backtracking algorithm.

5.1 Data Stack

In a manner very similar to that of the Forth language [2], the abstract machine utilises a data stack for the storage and later retrieval of simple data (integers and strings). Nearly all of the instructions operate on the data stack in some way. There are simple integer operations, such as `Add` and `Divide` (which work on the top two items of the data stack), stack manipulation operations such as `Pick` and `Drop` (which rotate, copy or destroy parts of the stack), and the graph query operations `N?` and `E???`[8] (which search the graph for particular structures). All of these operations have some effect on the data stack, they usually pop their arguments and push their results. In operation the data stack does not get very big and it is normally possible to determine its maximal size statically. This is because the only instructions with a variable effect on the stack are `Pick`, `Roll` and `UnRoll` which pop some integer i and then copy up the ith item of the stack, rotate the top i items "forwards", or rotate the top i items "backwards", respectively. These three instructions are almost always preceded with an integer (`PushI`) which controls their behaviour.

5.2 Choice Stack

Backtracking is implemented using a choice stack. Whenever an instruction returns only one of a number of possible results (such as in the `NA` "Any Node" instruction), a frame is pushed onto the choice stack describing the current state of the system. The frame contains a copy of the current data stack, the program counter, the step number and the try number. Later, if the choice needs reconsidering, the state of the machine can be restored and the next choice tried.

5.3 Graph Change Stack

The state stored in a frame on the choice stack does not save the current state of the graph. To copy and store the entire graph whenever a choice is made or reconsidered would quickly exceed the memory available to any implementation. For this reason there is the graph change stack. Whenever a graph-modifying instruction is executed, a new frame is pushed onto the graph change stack. This new frame describes exactly how to undo the changes made to the graph. When backtracking occurs, in addition to restoring program counter, data stack and try number, frames

[8] In these instructions, `?` is a wildcard for `L`, `A`, or `I`. For example the instruction `EILA` is short for "An Edge from Node with Identifier i to Label j via Any edge."

are popped off the graph stack (and the appropriate changes made to the working graph) until the step number of the top of the graph change stack is smaller than the step number of the state being restored. Hence the graph will have been restored correctly at this point in the execution history.

6 Related work

This section briefly discusses the relation of the YAM to graph transformation languages other than GP.

AGG [3] is a Java-based system for graph transformation. A distinctive feature of AGG are rules with negative application conditions, which specify a graph that must not be in the current graph in order for the rule to match. Such a forbidden subgraph can be expressed in YAM code. More challenging are AGG's attributes imported from Java which are not present in the YAM language (as they are incompatible with our leitmotiv of semantic simplicity).

A similar remark applies to the attributes imported from C in PROGRES [7]—a complex graph transformation languages with some involved features. Whilst some of PROGRES' constructs will be easily expressible in YAM code, many will not. For example, the language's type system with graph schemata and path expressions has currently no counterpart in the YAM. According to [7,8], PROGRES is compiled to bytecode of an abstract machine—but we are not aware of a description of this machine.

The FUJABA language [4] is grown from PROGRES but abandons backtracking because "extensive experiences have shown that it is seldom used". Backtracking is needed in the YAM, however, to implement GP's nondeterministic semantics (given in [6]).

7 Conclusion and future work

The implementation of GP [6] will be based on the YAM. A compiler for converting GP programs into YAM code is under development. Experiments show that the machine executes programs much quicker than earlier GP implementation attempts, which took an interpreter approach. This is because most of the rule analysis (such as determining which nodes and edges in a rule get added, deleted or renamed) is done at compile time whereas at run time, the YAM doesn't need to do any analysis.

The current implementation of the YAM is written in about 3000 lines of C code. C was chosen because it is relatively low-level and efficient, and allows to control memory management completely. As an indication of execution behaviour, running the `VertexColouring` program on a 100 node, 300 edge random graph [9] takes approximately 145000 single instruction executions (including backtracking) and 464 graph changes (all of them node renamings), and involves 1658 choice points. On a 2.4 GHz PC with 512 Mb of memory this execution takes

[9] Obtained by creating 100 nodes and adding an edge between two random nodes 300 times.

less than 0.1 seconds. Running `VertexColouring` on a 1000 node, 3000 edge random graph requires approximately 10.7 million steps and 4350 graph changes, involves 15402 choice points, and takes about 11 seconds.

The YAM makes a useful abstraction. For example, the machine was not designed with GP's while-loop in mind. However the machine needs no modification to accomodate it, as the behaviour of the while construct can be captured at compile time. We expect this to be the case for some other constructs that will be added to GP, too.

Future versions of the YAM will provide support for recursive procedures and a type system for graphs because these features will be incorporated in GP. It will also be useful to equip the YAM with a user-friendly interface and to couple the machine with an animation component showing one or all possible executions of a given program.

There is also scope for static analysis of the YAM code: each decision point can be found, and the number of choices to make at any given point can be related to the size of the current graph. Hence, if the number of iterations of rules and while-loops can be estimated or calculated, then it is possible to give time bounds for programs in terms of the size of the input graph.

In the spirit of GP having a simple semantics, it is also the topic of future work to produce a simple, formal semantics for all elements of the YAM.

References

[1] Hassan Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, 1991.

[2] Leo Brodie. *Starting Forth: an introduction to the Forth language and operating system for beginners and professionals*. Prentice Hall, 1981.

[3] Claudia Ermel, Michael Rudolf, and Gabi Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 14, pages 551–603. World Scientific, 1999.

[4] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A new graph rewrite language based on the Unified Modeling Language. In *Theory and Application of Graph Transformations (TAGT'98), Selected Papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer-Verlag, 2000.

[5] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.

[6] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer-Verlag, 2004.

[7] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.

[8] Albert Zündorf. *Programmierte Graphersetzungssysteme*. Doctoral dissertation, RWTH Aachen, Deutscher Universitäts-Verlag, 1996. In German.

Appendix: Hand-compiled code for `VertexColouring`

```
//a macro to find an edge with start and end
//node labels identical.
:EXXA NA Dup Get_node_label Swap EILA ;
Init!: InitEnd ALAP
NA
Dup Get_node_label ATOI 1 Equals Not Assert
1 ITOA Relabel_node
Init! Jump
InitEnd: Init! Cut
Inc!: End ALAP
Inc1: Inc2 Choice
EXXA
Dup Is_loop Not Assert
Get_edge_end Dup Get_node_label ATOI 1 Add ITOA Relabel_node
Inc! Jump
Inc2:
EXXA
Dup Is_loop Not Assert
Get_edge_start Dup Get_node_label ATOI 1 Add ITOA Relabel_node
Inc! Jump
End:
NoOperation
```

An Example of Cloning Graph Transformation Rules for Programming

Mark Minas [1]

Universität der Bundeswehr München
85577 Neubiberg, Germany

Berthold Hoffmann [2]

Technologiezentrum Informatik
Universität Bremen
28334 Bremen, Germany

Abstract

Graphical notations are already popular for the design of software, as witnessed by the success of the Uniform Modeling Languages (UML). In this paper, we advocate the use of graphs and graph transformation for programming graph-based systems. Our case study, the flattening of hierarchical statecharts, reveals that *cloning*, a recently proposed transformation concept, makes graph transformation rules (in the double-pushout approach) more expressive. Thus programming becomes easier, and gets along with simpler control conditions in particular.

Key words: Statecharts, Graph transformation, Clones

1 Introduction

Visual notations have always been popular for designing software, even more since the appearance of the *Unified Modeling Language* (UML), a family of mostly graph-like diagram languages. On the contrary, visual programming with graphs is still far less popular, although graphs are a convenient data structure and the computational model of graph transformation is well developed. We suspect that this is because graph transformation rules alone are not enough. For programming, transformations have to be extended by control conditions (Grace [1]), by control programs (Progres [6]), by control diagrams (Fujaba [3]), or by control predicates (DiaPlan [2], GReAT [5]),

[1] Email: Mark.Minas@unibw.de
[2] Email: hof@tzi.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and control mechanisms force the user to program in a way that is very similar to conventional languages. Graph transformation only simplifies the description of a program's basic operations, but writing down an algorithm that applies these operations in a controlled way is hardly improved by current programming languages based on graph transformation.

This paper tries to improve graph-transformation-based programming by simplifying its control mechanisms. This requires that basic graph transformation steps become more expressive. We are adopting the concept of *cloning* graph transformation rules that has been proposed recently [4]. Cloning allows to express basic program steps with a single transformation rule (scheme) which would otherwise require complicated loops and alternatives. Yet, cloning is a natural approach that does not imply additional efforts to the programmer. This is demonstrated by a case study, the flattening of hierarchical statecharts, i.e., transforming statecharts containing compound states (and-states and or-states) into flat statecharts without them.

The following section briefly introduces the idea of graph transformation with cloning before the problem of flattening statecharts is described in Sect. 3. Sect. 4 shows the proposed approach of applying graph transformation with cloning to solve this problem. The last section concludes the paper.

2 Graph transformation with cloning

This section gives only an informal overview of graph transformation with cloning. Details can be found in [4]. The idea is to specify rule schemes which represent an in general infinite number of rule instances by cloning certain subgraphs. Before describing the rule schemes, we describe the graphs that constitute their left-hand sides and right-hand sides.

A *pattern* is a graph G wherein some nodes are annotated by *cardinality variables*. Each cardinality variable y determines a subgraph G_y which consists of the nodes annotated with y (the *y-fold nodes*), their incident edges, and their adjacent nodes (the *border nodes*). A *y-clone* of pattern G is obtained by binding y to an integer value $k \geq 0$, removing all y-fold nodes and their incident edges from G, and gluing k disjoint copies of G_y to the border nodes. A pattern gets instantiated by binding each cardinality variable to a non-negative integer value and creating y-clones for each variable y. The order in which variables are cloned does not matter as cloning is commutative [4].

A *rule scheme* is a double-pushout rule where left-hand side (lhs), interface, and right-hand side (rhs) are patterns, and which uses pattern morphisms as a straight-forward extension of graph morphisms. Cardinality variables of the rhs have to occur in the lhs also. A rule instance is again a double-pushout rule. A rule is obtained by binding variables of the rhs to the same values as the ones of the lhs and instantiating lhs, interface, and rhs.

As usual, we will omit the interface when drawing a rule scheme, but indicate images of interface nodes by integer numbers. As an example, consider

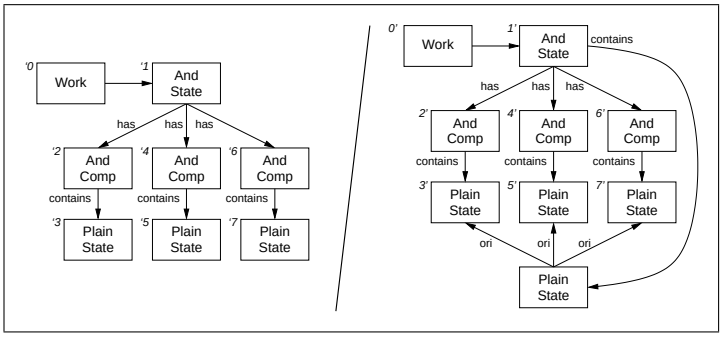
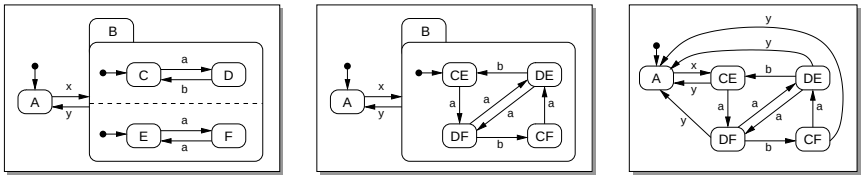


Fig. 1. Rule instance of the rule scheme shown in Fig. 6a with x bound to 3.



(a) (b) (c)

Fig. 2. A statechart diagram with an and-state (a), an equivalent statechart diagram where the and-state has been transformed into an or-state (b), and an equivalent, flat statechart diagram (c).

Fig. 6a which is shown as lhs/rhs. Interface nodes are the ones with the same number, i.e., nodes '1,'2, '3, and '4 of the lhs resp. 1', 2', 3', and 4' of the rhs. The rule scheme is annotated with cardinality variable x. x-fold nodes are visualized as a stack of nodes with an inscribed x. Fig. 1 shows an instance of this rule after creating x-clones with x bound to 3.

3 Statecharts

Statecharts are a visual language for describing behavior; under the name state diagrams, it is one of the graph-like languages of the UML. Statechart diagrams are an extension of finite state machines where transitions are annotated by events, firing conditions, and actions that have to be evaluated when a transition "fires". Hierarchical states can be used to simplify the description of complex behavior. There are two kinds of hierarchical states: An *or-state* contains a complete statechart diagram. When the or-state is active, then one of its contained states is active. Fig. 2b shows a statechart diagram with the or-state B. An *and-state* consists of several and-compartments that are separated by dashed lines. Each compartment contains a complete statechart diagram, like an or-state. When an and-state is active, all statecharts contained in its compartments are active in parallel. Fig. 2a shows a statechart diagram with such an and-state B. In order to fit the description of the flattening algorithm into this paper, we are using simplified statechart diagrams in the following. Simplified statecharts consist of a collection of plain states, and-states, or-states, and initial pseudo states. Initial pseudo states

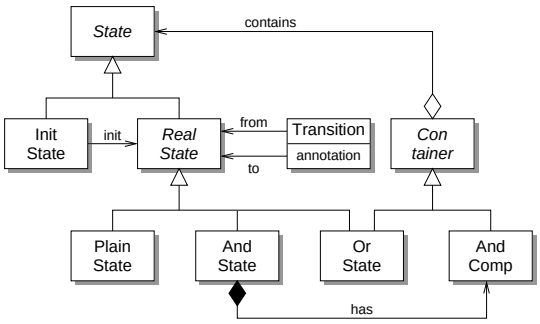


Fig. 3. Metamodel of the considered statechart diagram language.

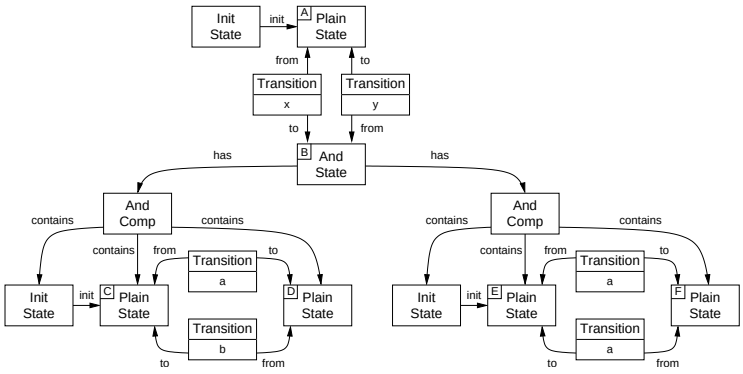


Fig. 4. Graph representation of the statechart diagram in Fig. 2a.

cannot be active; as soon as a statechart is activated, its initial pseudo state is entered and immediately left again via its only and non-annotated transition entering its connected state. Final states and history states are not used here. The annotation of transitions is also simplified: It may consist of events only; firing conditions and actions are not considered here. Also, transitions are not allowed to cross the borders of and-states and or-states. Finally, we simplify handling of conflicting transitions. UML statecharts give transitions firing from higher levels priority over those firing from lower levels. We ignore priorities and assume non-determinism for simplicity.

Fig. 3 defines the metamodel of the considered statechart diagram language as a class diagram. States are either initial pseudo states or real states, i.e., plain states, or-states, or and-states. Or-states are also containers which contain the states of the contained state diagram. And-states consist of and-compartments (*And Comp* in Fig. 3) which are also containers with contained states. Initial pseudo states are associated to their connected state, and transitions are associated to their connected states by *from* and *to* associations. Transitions have an *annotation* attribute that contains the string-valued event for this transition. Based on this metamodel, each statechart diagram can be represented as a graph. Fig. 4 shows the graph representation of the statechart diagram in Fig. 2a.

A transition fires if the state at the transition's source is active and the event is the transition's annotation. If more than one state is active within

an and-state, all outgoing transitions with corresponding annotations fire. An active state whose outgoing transition fires gets inactive, and an inactive state whose incoming transition fires gets active. If an or-state gets active, the state that is connected with the or-state's initial pseudo state gets active, too. For and-states, all of its compartments get active like or-states. Or-states and and-states of our simplified language can be left from any contained state. State B of Fig. 2a, e.g., is left as soon as a y event occurs where any of the contained states C, D, E, or F may be active.

Based on this semantics of transition firing, we can "flatten" a hierarchical statechart diagram by replacing and-states and or-states by equivalent sub-statechart diagrams. Fig. 2 shows this process: All and-states are first replaced by equivalent or-states (b), and finally, all or-states are removed (c).

Replacing an and-state is the more complicated step. The idea of this step is to turn any combination of active contained states into new states. This is similar to building product automata of finite automata. Therefore, we build the cross product of state sets for any and-compartment.[3] In Fig. 2, the contents of and-state B are replaced by the cross product $\{C, D\} \times \{E, F\}$. Transitions are considered next. This step is different from building product automata because a statechart drops an event if the event cannot be consumed by a firing transition. If more than one statechart is active in an and-state, an event can be consumed by some statecharts, but dropped by the others depending on the currently active states and their outgoing transitions. This defines the transitions that have to be created: Let $(s_1, \ldots, s_n)$ be a state from the cross product. If it is active and an event e occurs, there are some and-compartments' statecharts which consume e by a transition $s_i \xrightarrow{e} t_i$, and the other compartments' statecharts drop e, i.e., they stay in their state $s_i = t_i$. Therefore, the equivalent or-state replacing the and-state must contain a transition $(s_1, \ldots, s_n) \xrightarrow{e} (t_1, \ldots, t_n)$. Fig. 2b shows the result.

Removing an or-state is actually simple. As one of the contained state is active iff the or-state is active, we can simply drop the or-state frame, but must take care of the transitions from and to the or-state. Transitions to the or-state are redirected to the contained state that is connected to the or-state's initial pseudo state. And as the or-state can be always left by a transition from the or-state, each transition from the or-state has to be replaced by copies from each of the contained states. Fig. 2c shows the result.

The previous paragraphs have outlined the algorithm of flattening hierarchical statecharts rather coarsely. A precise algorithm in a textual or a common graph-transformation-based language would make use of an abstract representation like the one shown in Fig. 4, and one can imagine that writing it down requires several nested loops and complicated transformations. In the next section, we define the algorithm based on rule schemes that need only a very simple control structure.

[3] This is an expensive operation. A more efficient solution, e.g., is discussed in [7].

```
1  while the graph contains And State or Or State nodes do
2        // remove a bottom level and-state
3        if possible mark_bottom_level_and;
4        for all matches and_create_cross_product;
5        if possible and_create_init;
6        for all matches and_create_trans;
7        as long as possible and_clean_up;
8        // remove all or-states
9        as long as possible or_move_outgoing_trans;
10       as long as possible or_remove
11 done
```

Fig. 5. Control program for flattening statecharts

4 Flattening Statecharts by graph transformation

Sect. 2 introduced rule schemes which shall here be used for specifying the single transformation steps. They have to be combined with a control program in order to present a complete algorithm for flattening hierarchical statecharts. We do not present the control program in some existing language, but use a rather informal notation as we focus on rule schemes and how they make programming with graph transformations easier. Fig. 5 shows this control structure which calls several operations that make use of a single rule scheme. The control program, its constructs, and the rule schemes are explained in the following. Graph transformations require statecharts to be represented as graphs according to the metamodel in Fig. 3.

The control program consists of an outer loop that is repeated as long as there are still hierarchical states. The loop body consists of two parts. The first one (line 3–7) transforms a single and-state which does not contain any hierarchical sub-states (a so-called *bottom level and-state*) into an or-state. This or-state, together with all other or-states, is flattened in the second part (line 9 and 10). This procedure has to be repeated because the graph may contain several and-states, and and-states can be nested, i.e., and-states are flattened from the inside out. The operations follow the overview of the algorithm given in Sect. 3.

Lines 3–7 operate on an and-state that does not contain any hierarchical states. Rule *mark_bottom_level_and* arbitrarily selects one of them by adding a *Work* node [4] to the graph and connecting it with the corresponding *And State* node. Negative application conditions make sure that this and-state does not contain any and- or or-state. This elementary rule is not shown here for space restrictions. The control **if possible** tries to apply rule *mark_bottom_level_and*, but simply continues if the rule fails, i.e., if there is no bottom level and-state.

Fig. 6a shows the rule scheme of the operation *and_create_cross_product*

[4] Please note that the new *Work* node actually violates the metamodel shown in Fig. 3. A possible solution would be relaxing the metamodel while transforming the graph or extending the metamodel accordingly.

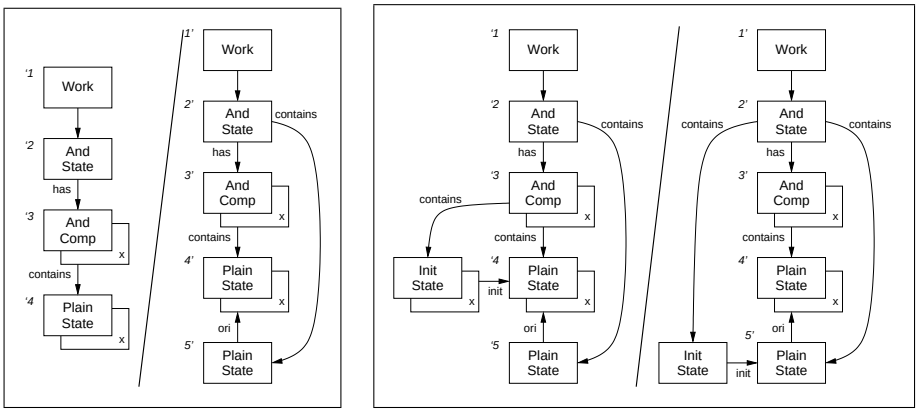


Fig. 6. Rule schemes of operations *and_create_cross_product* (a) and *and_create_init* (b).

that creates the cross product of the state sets of all and-compartments. This rule scheme can be applied to the and-state that has been selected by the previous rule. The idea of the rule scheme is to match an and-state node together with all of its and-compartments and one contained state of each compartment by a rule instantiation. Fig. 1 shows an instance of this rule scheme when binding x to 3, i.e., when the and-state has 3 and-compartments.

By applying the rule instance, a new plain state node 5' is added. Node 5' becomes a new contained node of the and-state (which later will become an or-state by operation *and_clean_up*). Moreover, 5' gets connected by *ori* edges to the original state nodes of the different and-compartments. That way, 5' represents the tuple of state nodes from the different and-compartments. However this requires that cardinality variable x is always bound to the maximum possible value when instantiating the rule. By applying this rule scheme for any possible match for the maximum value being bound to x, this rule scheme creates the complete cross product of the state sets of all and-compartments for the selected and-state. This behavior is specified by the loop control **for all matches**. Please note that this control does not repeat applying the specified rule scheme as long as possible. As the rule scheme's right-hand side contains the left-hand side, this would specify a non-terminating transformation. The loop control rather has to find for the maximum x-value all possible matches and consecutively apply the corresponding rule instance to these matches.

The previous rule instance sets up the cross product of state sets which is the new or-state's set of contained states. However, the initial pseudo state has not yet been created. This is done by operation *and_create_init* (Fig. 6b). Its left-hand side is the same as the right-hand side of the previous rule scheme, but with an additional *Init State* node and with *init* edge. Therefore, the instantiation of the left-hand side, matches the tuple of all initial pseudo states, their connected real states and the new tuple state node that has been created by operation *and_create_cross_product*. The original initial pseudo states

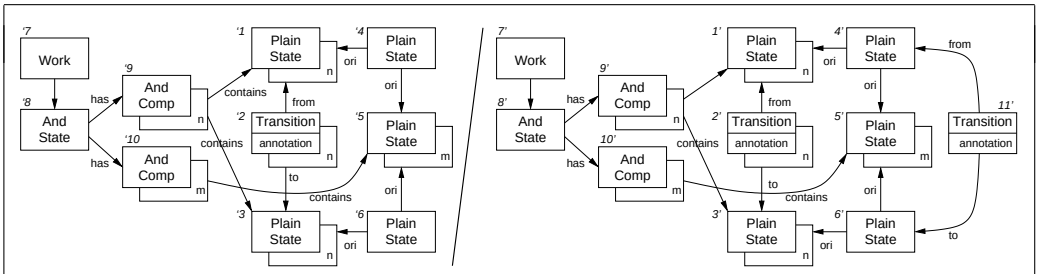


Fig. 7. Rule scheme of operation *and_create_trans*

get removed, and a new initial pseudo state gets created. Again, cardinality variable x must be bound to the maximum possible value. The control **if possible** is needed as this rule fails if there is no bottom level and-state.

The next operation has to create transitions between the new states. Fig. 7 shows the corresponding rule scheme. The left-hand side represents two tuple states ‘4 and ‘6 that represent the states ‘1 and ‘5, resp. ‘3 and ‘5. Cloned nodes ‘1 and ‘3 together with ‘2 are those states s_i resp. t_i together with their connecting transition that – as described in Sect. 3 – consume an event by firing the corresponding transitions. As a consequence, this rule scheme has to require as an application condition that the transitions that are matched by the instance of the left-hand side must have the same value of their annotation attribute. [5] Node ‘5 represents the states of those and-compartments that do not consume the corresponding event. Again, cardinality variable n has to be bound to the maximum possible value such that the application condition is satisfied. Afterwards, m has to be bound to the maximum possible value. This requirement is specified by introducing an ordering on the cardinality variables, here $n \prec m$, telling whose value has to be maximized first. When applying an instance of this rule, a new transition is added between the tuple states 4’ and 6’. This new transition has to get the same annotation value as the one of all matched transitions of the left-hand side. The loop control **for all matches** takes care of creating all possible transitions.

The “internals” of the new or-state have been completely created by line 4–6. However, the remaining and-compartments, the original transitions and state nodes as well as the connection edges together with the *Work* node have to be removed. The *And State* node, moreover, has to be replaced by an *Or State* node. This is performed by the operation *and_clean_up* in line 7. *And_clean_up* is actually a set of three straight-forward rule schemes that has been omitted here for space restrictions.

[5] This application condition in Fig. 7 uses *card* as a function that obtains the set cardinality of the multiset of all ‘2.annotation values. The action uses *select* which selects a value from a multiset.

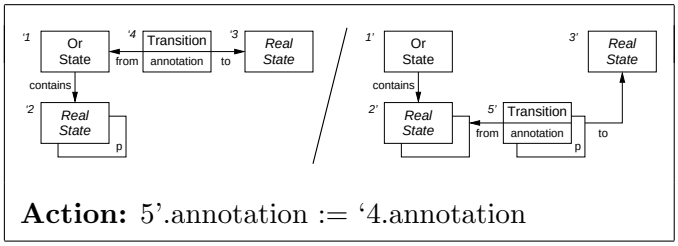


Fig. 8. Rule scheme of operation *or_move_outgoing_trans*

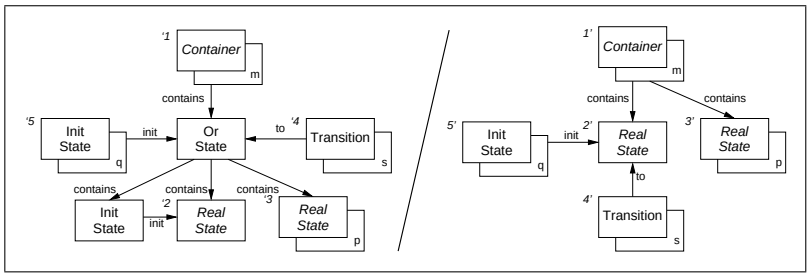


Fig. 9. Rule scheme of operation *or_remove*

The remaining two operations with loop controls (Fig. 5) flatten all or-states of the graph. This applies in particular to the or-state that just has been created from a bottom level and-state. As described in Sect. 3, we have to add copies of transitions leaving an or-state to each of the contained states. This is done by the rule scheme shown in Fig. 8 which is applied with maximum p value to each match of the left-hand side. Please note that as many copies of transition '4 are created as there are states contained in the or-state. All of these new transitions have to get assigned the same annotation values as the match of '4 (c.f. the action in Fig. 8).

Fig. 9 shows the final rule scheme which redirects all incoming transitions to the state that has been connected to the or-state's initial pseudo state, removes this pseudo state and the or-state frame, and uses the containing state – if any – of the previous or-state as the container of the states that have been contained by the or-state. Node '5 with cardinality variable q represents a potential initial pseudo state that has been connected to the or-state. This transition has to be redirected, too.

Please note that the graph does not contain any *Or State* nodes after processing line 10. However, new *Or State* nodes are created in the next loops if there are still *And State* nodes in the graph.

5 Conclusions

In this paper, we have considered the flattening of hierarchical statecharts as a case study for programming based on graph transformation. Space restrictions did only allow to treat simplified statecharts. The missing concepts, like history states, final states, firing conditions, and transition actions as well as enter and exit actions of hierarchical states, can be added in a straight-forward

way. A complete specification will be provided in the future.

The case study revealed that instantiation of rule schemes by cloning makes graph transformation more expressive; it simplifies programming by graph transformation as control programs become simpler. We hope that this concept will support graph transformations to be better accepted for programming tasks in the future.

The expansion of variables to graphs proposed in [4] makes graph transformation still more expressive. But this is beyond the scope of this paper.

Control programs have been presented quite informally in this paper. We are currently combining rule instantiation by cloning with the proposed graph-transformation-based programming language DiaPlan [2] (which already supports variable expansion) and its control structures. This will allow to discuss control structures more thoroughly.

References

[1] Andries, M., G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr and G. Taentzer, *Graph transformation for specification and programming*, Science of Computer Programming **34** (1999), pp. 1–54.

[2] Drewes, F., B. Hoffmann, R. Klein and M. Minas, *Rule-based programming with DiaPlan*, ENTCS **127** (2004), pp. 15–26, Proc. Int. Workshop on Graph-Based Tools (GraBaTs'04). Rome (Italy), Oct. 2, 2004.

[3] Fischer, T., J. Niere, L. Turunski and A. Zündorf, *Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java*, in: H. Ehrig et al., editors, *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, LNCS 1764 (2000), pp. 296–309.

[4] Hoffmann, B., D. Janssens and N. Van Eetvelde, *Cloning and expanding graph transformation rules for refactoring*, in: *Proc. Int. Workshop on Graph and Model Transformation*, Tallinn (Estonia), Sept. 28, 2005, to appear in ENTCS.

[5] Karsai, G., A. Agrawal, F. Shi and J. Sprinkle, *On the use of graph transformation in the formal specification of model interpreters*, Journal of Universal Computer Science **9** (2003), pp. 1296–1321.

[6] Schürr, A., A. Winter and A. Zündorf, *The Progres approach: Language and environment*, in: G. Engels et al., eds., *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, World Scientific, Singapore, 1999, pp. 487–550.

[7] Wasowski, A., *Flattening statecharts without explosions*, in: *LCTES '04: Proc. 2004 ACM SIGPLAN/SIGBED Conf. on Languages, compilers, and tools for embedded systems* (2004), pp. 257–266.

A Rule-Based, Integrated Modelling Approach for Object-Oriented Systems

Benjamin Braatz [1]

Insitut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany

Abstract

In this paper an integrated modelling approach for object-oriented systems is proposed. The integrated language consists of three layers. On the first layer UML class diagrams are used to define the structure of the modelled systems and OCL expressions specify queries, which do not modify the object configuration. On the second layer transformation rules model local state modifications of the system. On the third layer Nassi-Shneiderman diagrams describe complex control flows built over the rules and queries on the lower layers. The proposed integrated language is evaluated by a running example on modelling doubly linked lists and the mergesort algorithm.

Key words: Rule-Based Transformation, Nassi-Shneiderman Diagrams, UML, Integration

1 Introduction and Related Work

The Unified Modeling Language (UML) [10] has become the pre-dominant modelling language in object-oriented software development. The behavioural techniques provided by the UML, however, do not contain a method for the declarative, rule-based specification of modifications on object structures. Moreover, the interconnection between different behavioural techniques is treated rather superficially in the UML specification, because the UML tries to permit as many usage and interconnection scenarios as possible.

In this paper an integrated modelling approach is proposed, which tries to eliminate these deficiencies by giving a layered collection of specification techniques with a clear separation of concerns and well-defined interconnections. The proposed integrated language is organised in three constitutive layers.

On the first layer UML class diagrams are used to specify the structure of the system. Expressions of the Object Constraint Language (OCL) [9]

[1] bbraatz@cs.tu-berlin.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

specify the behaviour of query operations, which do not change the object configuration of the system. This layer is introduced in Sect. 2.

On the second layer local state changes are modelled using a variant of single-pushout graph transformation rules [1] tailored to the UML on this layer. This layer is described in Sect. 3. Object-Based Graph Grammars [11] and Object-Oriented Graph Grammars [7,6] are other approaches using graph transformation rules to specify the behaviour of object-oriented systems. They are, however, designed as self-contained specification techniques without relation to the UML.

On the third layer the assembly of the queries and rules from the previous layers into complex control flows is achieved by structured flowcharts [8], which are also known as Nassi-Shneiderman diagrams. The third layer is explored in Sect. 4. Structured flowcharts are favoured over e. g. UML activity diagrams in this paper, because we believe that they provide a viable alternative for the visual modelling of control flows. On the one hand, they are close to the structure of the constructs in most contemporary, imperative programming languages, which could lead to a broader acceptance among programmers and software designers. On the other hand, the separation of concerns realised by the layered, integrated language facilitates the compactness and comprehensibility of the flowcharts making them easier to grasp than the source code of a programming language. Different approaches to control the application of graph transformation systems are proposed in the literature. Transformation Units [4] are one of the most prominent examples. They provide an abstract framework for the definition of control structures over transformation rules, where the structured flowcharts of this paper could probably be integrated into that framework as a sophisticated specification language for control structures.

These layers are integrated in the sense that the interconnections between the different layers are precisely defined: The queries on Layer 1 are only allowed to call other queries on Layer 1. The transformation rules on Layer 2 may use OCL expressions on Layer 1 in their attribute specifications. The structured flowcharts on Layer 3 can use OCL experessions on Layer 1 and call arbitrary other operations, which may be specified by transformation rules on Layer 2 or other flowcharts on Layer 3. The integrated language is intended to be constructive in the sense that the behaviour of a system can (and should be) completely described using the sublanguages on the appropriate layers.

The Fujaba Tool Suite [3] uses Story Diagrams [2], which are a combination of activity and collaboration diagrams, to specify transformations on object-oriented systems. The Fujaba approach is very similar to the one proposed in this paper. In contrast to Fujaba, which employs Java source code for the specification of low-level expressions, we use OCL expressions, which are on the one hand already integrated into the UML family of languages, on the other hand they aid in keeping the approach platform independent. Another difference is the strict separation of concerns with transformation rules and flowcharts specified in self-contained diagrams, respectively, where

Fujaba uses the integrated Story Diagrams. The separation of concerns eases the reuse of transformation rules in different flowcharts and of flowcharts in other flowcharts. The choice of visualisation techniques for rules and control flow is the last main difference. Fujaba uses collaboration diagrams to visualise rules, while in this paper a seperate visualisation of left- and right-hand-side is chosen, because the collaboration visualisation cannot adequately capture the change of attribute values and the specification of negative application conditions. In Fujaba UML activity diagrams are used for the control flow, where our reasons for choosing structured flowcharts have already been given above.

The semantics of the UML languages is – sometimes deliberately – left ambiguous and only given in natural language. In order to allow features like precise reasoning and code generation, a more restricted and formal approach has to be considered. Therefore, the integrated modelling language proposed in this paper is designed to allow the definition of a precise semantics.

It is possible to use graph transformation systems also as the semantic domain of object-oriented modelling techniques. This is done in [13,5], where UML class, object, state, collaboration, and use case diagrams are translated into graph transformation systems. This approach is complementary to the one in this paper, where graph transformations are used as an additional modelling technique on the syntactical level.

2 UML Foundations

In this paper we use UML class diagrams [10] to specify the class structure of the modelled system. Additionally, the behaviour of query operations, which do not change the object configuration of the system, is specified by OCL constraints [9], which are guaranteed to have no side effects on the system state. Note, that we do not employ OCL invariants and pre-post-conditions in this paper. Those constraints will be considered in future work on verification, shortly discussed in Sect. 5, where they will play the role of descriptive specifications against which the constructive models defined in this paper should be checked.

The two subsequent layers will be designed to closely match and reuse the concepts of class diagrams and OCL constraints. For example, the self and return parameters of OCL have counterparts in both rules and flowcharts. Additionally, rules and flowcharts also require OCL expressions in various locations.

As a running example we specify doubly linked lists, whose elements contain an integer as key and a string as data content. The class diagram of the example is depicted in Fig. 1. The abstract class ListItem is used as an abstraction of the common characteristics of lists and the elements in the lists. The objects of the List class serve as sentinels for the list, such that the next item is the first element of the list and the previous is the last one. This approach

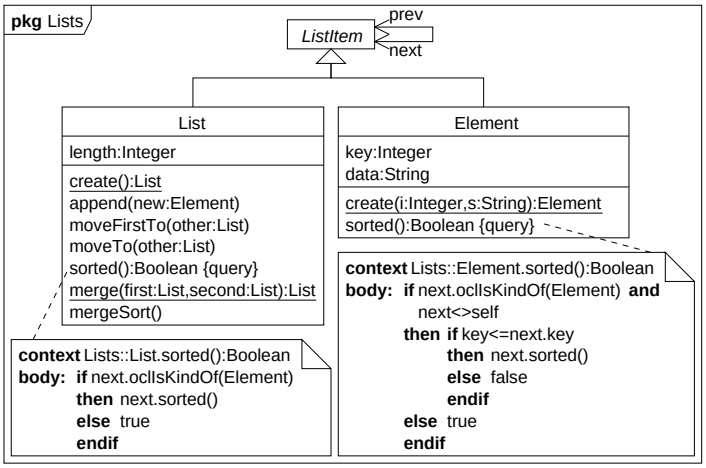


Fig. 1. Class diagram of the example

ensures that we do not have to deal with undefined pointers. Moreover, it allows to check if a list is non-empty and if an element has another successor by calls to the builtin OCL property oclIsKindOf(Element) on the next link. Note, that this class diagram would also allow object structures with multiple instances of List in a list, but the operations – namely the structure modifications specified by the rules in the next section – do not allow the creation of such senseless structures.

The underlined operations List.create, Element.create, and List.merge are static operations, which are called in the context of the class instead of a particular object of the class. The operations List.sorted and Element.sorted are annotated with a query property string expressing that they are not allowed to change the configuration of objects and attributes in the system.

The sorted queries are specified by the OCL constraints in Fig. 1. The List.sorted query returns true for an empty list and calls the Element.sorted query on its first element otherwise. The Element.sorted query returns true if the element is the last in the list, i. e. the next link points to the containing instance of List and not to another instance of Element, or the element is not contained in a list at all, i. e. the next link points to the element itself. If there is another element in the list, the keys are compared and false is returned if they are not in the correct order. Otherwise the query is called recursively on the next element.

3 Transformation Rules

On the second layer of the integrated modelling approach we will use transformation rules to describe local state changes in object configurations. These rules are a variant of single-pushout graph transformation rules [1].

A rule is given by a left- and a right-hand-side consisting of instance specifications. The left-hand-side (LHS) is connected to the right-hand-side (RHS) by a partial, injective mapping. Since the application of a rule should be de-

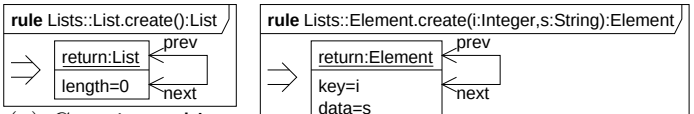
termined by the parameters given to the operation, we require the LHS to be uniquely navigable from the instance specifications representing the parameters, whose type is a class. A special parameter self is available in non-static operations to denote the object on which the operation is called. If the operation has a return parameter, the special parameter return has to be used on the RHS to designate the output of the rule. For attributes the instance specifications on the LHS may declare OCL constraints, which have to hold for the rule to be applicable. The instance specifications on the RHS may then specify the new values of attributes by OCL constraints, which may similarly to post-conditions use the @pre operator to access the attribute values before the rule application. The OCL attribute constraints may also use any data type parameters given to the operation.

Given a match of the LHS in an object configuration, the application of the rule can be constructed by removing the parts of the LHS not mapped to the RHS and adding the parts of the RHS, which do not have a preimage in the LHS. Since we assume an execution environment with garbage collection, we will use only rules, which are non-deleting on objects. Matches may in general be non-injective, but if there are contradicting attribute constraints for objects identified by the match, then the rule is not applicable. Operation invocations leading to a non-applicable rule should result in some kind of error handling, e. g. by throwing an exception, but the integration of exception handling is outside the scope of this paper and is left as future work.

In addition to the LHS and RHS, negative application conditions (NAC) may be defined for a rule. Such conditions are defined as non-injective extensions of the LHS, where non-injectivity is used to forbid the identification of certain elements by the match and extensions are used to forbid auxiliary object structures. If the NAC can be matched compatibly with the match of the LHS, then the rule is not applicable.

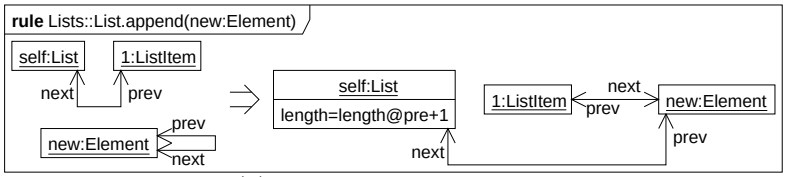
It may be argued that rules, which are allowed to manipulate all structures navigable from the called object and the call's parameters, contradict the object-oriented paradigm of encapsulation of object behaviour, but for the modelling of complex structure changes it seems appropriate to specify them as a rule operating under the control of one of the participating objects rather than dividing the operation, which logically belongs together, into operations on the different objects. In a more elaborated framework, which takes into account visibility and accessibility constraints to facilitate object encapsulation, these visibilities and accessibilities would of course have to be respected by the rules.

For our running example, the transformation rules in Fig. 2 can be used to create lists and elements. The List.create rule in Fig. 2(a) creates an empty list, while the Element.create rule in Fig. 2(b) creates an element containing the given integer as the key and the given string as the data content. The created element is not contained in a list, which is expressed by the next and previous links pointing to the element itself. Since these operations are both

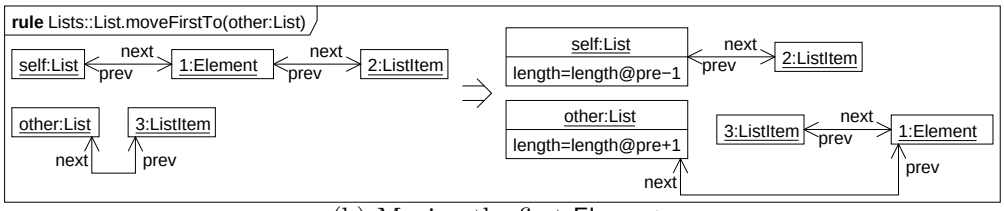


(a) Creating a List (b) Creating an Element

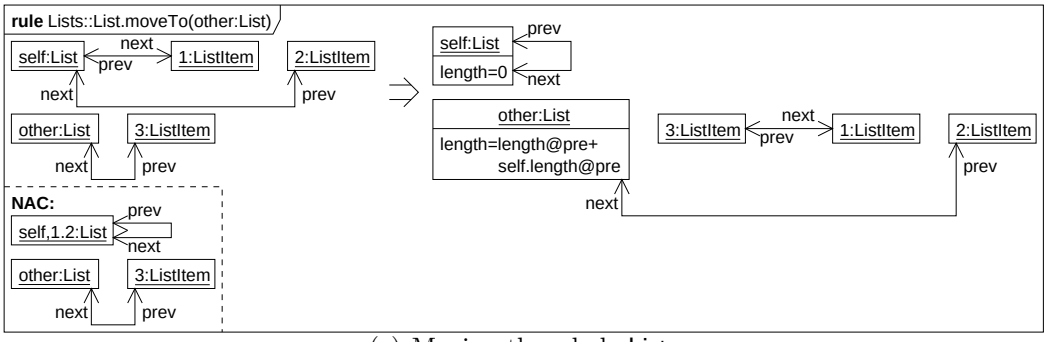
Fig. 2. Transformation rules for creating instances of ListItem



(a) Appending an Element



(b) Moving the first Element

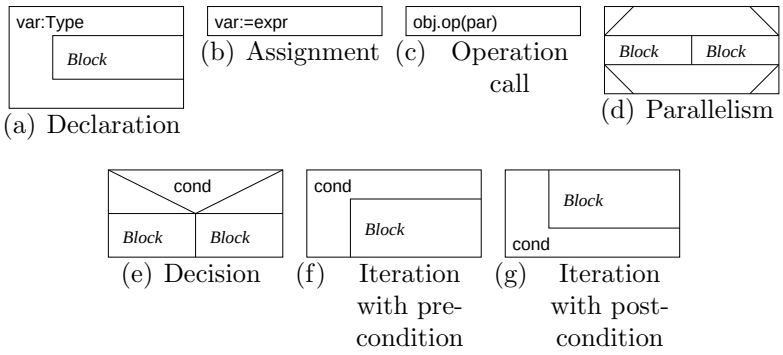


(c) Moving the whole List

Fig. 3. Transformation rules for modifying a List

static, there is no self instance in the LHS. The return parameter is used to transmit the created instances as operation results to the caller.

The rules in Fig. 3 modify a given list. The List.append rule in Fig. 3(a) appends a given element to the end of the list. The rule is not applicapble if new is already contained in another list, because the previous and next links are required to point to new itself. The List.moveFirstTo rule in Fig. 3(b) moves the first element of a list to the end of another list. This rule is not applicable if the self list is empty, because then there is no match for 1:Element, and if the caller tries to move to the same list, because the conflicting attribute specifications for self and other prohibit their identification. The List.moveTo rule in Fig. 3(c) moves a whole list to another empty list. Again, the rule is not applicable, when self and other refer to the same object, because of the conflicting attribute specifications. Additionally, the given NAC forbids



(a) Declaration (b) Assignment (c) Operation call (d) Parallelism

(e) Decision (f) Iteration with pre-condition (g) Iteration with post-condition

Fig. 4. Symbols used in flowcharts

the application on an empty list, because the application would lead to an ill-formed object configuration with the self list contained in the other list.

4 Structured Flowcharts

In this section we use structured flowcharts [8] as defined by Nassi and Shneiderman to describe control flows. The flowcharts are built over the queries and rules defined in the previous sections. Because the details of state changes are delegated to the rule-based operation specifications, the control flows remain concise and comprehensible.

Flowcharts are constructed using the block symbols in Fig. 4, where these blocks can be sequentially composed and recursively inserted for the *Block* nonterminals. A variable declaration shown in Fig. 4(a) consists of a previously undeclared variable var and a type Type. The scope of the variable is the contained block. Values can be assigned to variables by an assignment as shown in Fig. 4(b), where the variable var can be a previously declared variable or the special variable result. Note that neither the parameters of the operation including self, nor attributes may appear on the left side of an assignment, because modifications of the object structure should be specified by rules not by direct assignments. The expression expr with corresponding return type is constructed similar to OCL expressions, but it is, in contrast to OCL, also allowed to contain calls to non-query operations. Operations without return parameter can be called with the block in Fig. 4(c), where obj is a navigation path from a parameter or variable to an object, op is an operation of the class of that object, and par are parameters for the operation. Control flows, which can be executed parallely independent, can be expressed by the block in Fig. 4(d). A decision as in Fig. 4(e) is given by an OCL expression cond with Boolean return type, which is constructed over the parameters and previously declared and defined variables. If the query evaluates to true, the left block is executed, if it evaluates to false, the right block is chosen. The iteration with precondition in Fig. 4(f) corresponds to a while-loop in common programming languages. While the Boolean query cond evaluates to true, the block is executed. Conversely, the iteration with postcondition in Fig. 4(g)

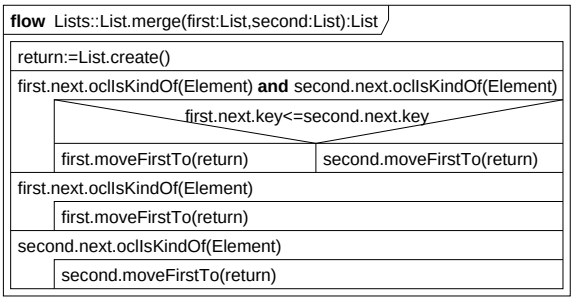


Fig. 5. Flowchart of merging two sorted lists

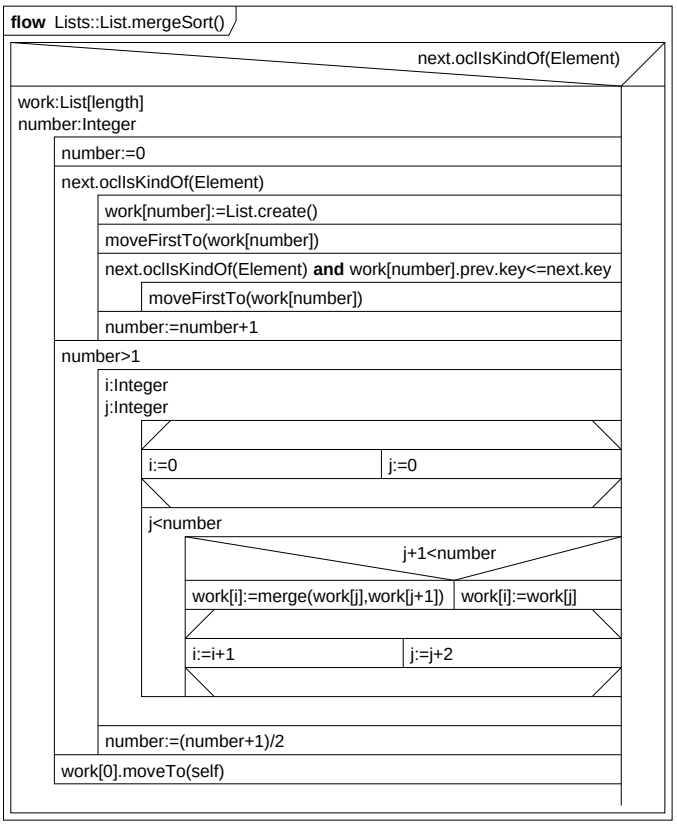


Fig. 6. Flowchart of the Natural Mergesort algorithm

corresponds to a repeat-loop, where cond is evaluated for the first time after the first iteration.

For our example of doubly linked lists, we will specify the Natural Mergesort algorithm [12], which utilises sublists that are already sorted to optimise the performance of the sorting procedure. In Fig. 5 we first specify an auxiliary operation for merging two already sorted lists. As long as both lists contain elements, the first elements are compared and the smaller one is moved to the resulting list. When one of the lists becomes empty, the rest of the other one is moved to the result and the operation terminates.

The Natural Mergesort algorithm itself is specified in Fig. 6. If the list is empty, nothing is done, otherwise the main part of the algorithm is started,

which consists of two parts. First, the list is broken up into already sorted sublists, which are stored in the work array of lists, where this array is declared to contain at most length lists. The actual number of sorted lists is stored in the number variable. Then, these lists are merged pairwise, which halves the number of sorted lists in each pass. This is done until only one list remains and this list is moved to the self list.

5 Summary and Future Work

In this paper an integrated modelling approach for object-oriented systems has been developed, which is organised in three constitutive layers. The first layer employs OCL to yield a functional description of query operations without side effects. On the second layer transformation rules are used to define the behaviour of operations, which change the object configuration locally. Finally, structured flowcharts are used on the third layer to specify complex control flows.

Interesting lines of future work include the extension of the presented approach with respect to further structural and behavioural aspects of the UML, like multiplicities, visibilities, signal and exception handling, and redefinition. The presented languages should be aligned with the UML metamodel by giving metamodels for the three layers. The definition of the abstract syntax by a graph grammar could complement the metamodel, where this also permits the use of graph transformation rules for refinement, refactoring, code generation, and other model transformations.

One of the main motivations for the work in this paper is to define a fully formalized object-oriented modelling technique. Hence, the languages will also be given an integrated formal semantics, which will then be used to facilitate formal verification. For the purpose of verification the constructive modelling techniques presented here will be complemented by descriptive specification techniques like OCL invariants and pre- and post-conditions or UML sequence diagrams. The system described by the constructive techniques can then be verified against the properties required by the descriptive techniques using a variety of verification methods.

Acknowledgement

I would like to thank Gabi Taentzer, Hartmut Ehrig, Rayo Kniep, and the anonymous referees for their valuable comments on previous versions of this paper.

References

[1] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, *Algebraic approaches to graph transformation – Part II: Single*

pushout approach and comparison with double pushout approach, in: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, World Scientific, 1997 pp. 247–312.

[2] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story Diagrams: A new graph rewrite language based on the Unified Modeling Language and Java*, in: *Theory and Application of Graph Transformations, TAGT '98*, LNCS **1764**, Springer, 2000 pp. 296–309.

[3] *Fujaba Homepage*, http://www.fujaba.de/.

[4] Kuske, S., "Transformation Units – A Structuring Principle for Graph Transformation Systems," Ph.D. thesis, Universität Bremen, Bremen, Germany (2000).

[5] Kuske, S., M. Gogolla, R. Kollmann and H.-J. Kreowski, *An integrated semantics for UML class, object, and state diagrams based on graph transformation*, in: *Integrated Formal Methods, IFM 2002*, LNCS **2335**, Springer, 2002 pp. 11–28.

[6] Lüdtke Ferreira, A. P., "Object-Oriented Graph Grammars," Ph.D. thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil (2005).

[7] Lüdtke Ferreira, A. P. and L. Ribeiro, *Towards object-oriented graphs and grammars*, in: *Formal Methods for Open Object-Based Distributed Systems, FMOODS 2003*, LNCS **2884**, Springer, 2003 pp. 16–31.

[8] Nassi, I. and B. Shneiderman, *Flowchart techniques for structured programming*, ACM SIGPLAN Notices **8** (1973), pp. 12–26, http://www.geocities.com/SiliconValley/Way/4748/nsd.html.

[9] Object Management Group, "OCL Specification, Version 2.0," (2005), http://www.omg.org/cgi-bin/doc?ptc/05-06-06.

[10] Object Management Group, "UML Superstructure Specification, v2.0," (2005), http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[11] Ribeiro, L., F. L. Dotti and R. Bardohl, *A formal framework for the development of concurrent object-based systems*, in: *Formal Methods in Software and Systems Modeling*, LNCS **3393**, Springer, 2005 pp. 385–401.

[12] Rolfe, T. J., *List processing: Sort again, naturally*, ACM SIGCSE Bulletin **37** (2005), pp. 46–48, http://penguin.ewu.edu/~trolfe/NaturalMerge/NatMerge.html.

[13] Ziemann, P., K. Hölscher and M. Gogolla, *From UML models to graph transformation systems*, in: *Visual Languages and Formal Methods, VLFM 2004*, ENTCS **127**, Elsevier, 2005 pp. 17–33.

A typed attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams

Frank Hermann [1,2] Hartmut Ehrig [3] Gabriele Taentzer [4]

Research Group TFS, Faculty IV
Technical University of Berlin
Berlin, Germany

Abstract

According to the UML Standard 2.0 class and sequence diagrams are defined in a descriptive way by a MOF meta-model and semi-formal constraints. This paper presents a formal and constructive definition of the abstract syntax of UML class and sequence diagrams based on the well-defined theory of typed attributed graph transformation with inheritance and application conditions. The generated language covers all important features of these parts of UML diagrams and is shown to satisfy all of the corresponding constraints by construction. An explicit model transformation demonstrates the close correspondence between the graph grammar and the MOF definition of UML class and sequence diagrams. The graph grammar is validated by well-established benchmarks showing that all important features of the MOF definition of UML are covered.

This formal constructive syntax definition of UML class and sequence diagrams is the basis for syntax directed editing, formal analysis, formal operational and denotational semantics and correctness of model transformations.

Key words: graph transformation, typed, attributed, inheritance, UML, sequence diagrams, class diagrams, abstract syntax

1 Introduction

Meta-modeling of visual languages, particularly the UML [10] defined by MOF [9], facilitates the definition of general structure elements and relations on

[1] Email: frank@cs.tu-berlin.de
[2] Supported by the German Research Society (DFG)
[3] Email: ehrig@cs.tu-berlin.de
[4] Email: gabi@cs.tu-berlin.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the one hand and the implementation of specific properties by constraints restricting the amount of valid instances on the other hand.

Due to the non-constructive nature of the MOF approach, i.e. there is no systematic method to generate all language elements, there exist well-known limitations, which are avoided by a constructive approach. Using typed attributed graph transformation with node type inheritance and application conditions as in [6] and [7] for defining a visual language allows the construction of elements of the language by applying rules of the corresponding graph grammar. The concept of inheritance allows creating an abstract rule, which defines an equivalent set of concrete rules, and therefore notably reduces the total amount of rules. The graph grammar GG_{CSD} for class and sequence diagrams, defined in [12], additionally uses a simple version of transformation units in the sense of [16] allowing to specify the construction of complex elements. This constructive definition shall not replace the original one, but build up a formal basis for certain applications.

Proving the correctness of GG_{CSD} relating the original specification of UML is not possible, because most of the constraints of the original definition of sequence diagrams are only informal. In contrast the formal definition eliminates some problems in the original definition (see 3.4). The explicit model transformation in Subsection 3.2 demonstrates the close correspondence to UML defined with MOF. Restrictions by multiplicities and constraints are already followed and argued at the corresponding rules.

A related approach for defining visual languages constructively is realized in [17] via an EBNF grammar. The application to UML is shortly sketched but not executed till now to our knowledge. In contrast to our visual specification this textual form includes many similarities to Java code as even the authors mention (p. 140). Previous applications of graph transformation describing the abstract syntax of UML diagrams used very simplified and restricted versions of the diagram types. The correspondence between the meta model for class diagrams and an implicit type graph is sketched in [15], but does not take advantage of a graph grammar to create the example diagrams needed for the described transformation. GG_{CSD} supports all important features of the current UML specification for class and sequence diagrams. Moreover an extension of the graph grammar to state machines was finalized in December 2005.

While UML class diagrams are widely known, the current version of UML sequence diagrams, which are special UML interactions and correspond to Life Sequence Charts as in [3], contain new and revolutionary features. Combined fragments as in Figure 1 offer the possibility to use control structures for managing the message flow in a sequence diagram. This leads to a compact notation for complex behaviors. The shown example specifies that a student is assigned to a class, if all previous costs were paid by him. Therefore the two scenarios of having a balanced account or having an unbalanced one are covered in one diagram by using the operator “opt” with its condition. Ad-

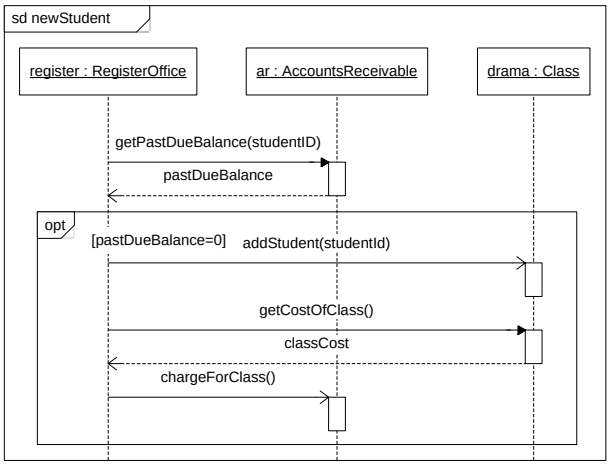


Fig. 1. Example of an UML Sequence Diagram (in [2] p. 9 fig. 9)

ditionally a variety of other operators together with multiple operands are available offering for example to specify parallel or alternative operands. A further new feature is the reuse of existing sequence diagrams in other sequence diagrams. Messages may cross the border of a used sequence diagram and lead into the using one. More details on sequence diagrams can be found in Chapter 14 of [10].

Implemented features of the grammar are validated by benchmarks as described in Subsection 3.3. Example diagrams in concrete syntax, originating from the IBM Rational Library [2] as shown in Figure 1, were recreated by applying the necessary rules leading to a graph representing the abstract syntax of the diagrams.

In a further step transformations into semantic domains shall be possible including operational and denotational semantics. These semantic representations may allow detecting internal and viewpoint conflicts as well as simulating the modeled system. Alternatively to sequence diagrams a specification by message sequence diagrams (MSCs) describes sequences of messages between objects. A formal semantics for MSCs was defined by Petri nets in [13] and allows simulation as well as analysis. Simulation and analysis of the graph grammar GG_{CSD} is possible using AGG (URL: http://tfs.cs.tu-berlin.de/agg/), a development environment for graph transformation systems, where transformation units can be simulated by using the command line input.

2 Graph Grammar for Class and Sequence Diagrams

The graph grammar GG_{CSD} for class and sequence diagrams generates instances of the corresponding parts of UML. It is defined by typed attributed graphs in the sense of [7], which integrate the graph structure and the attributes, which are elements of an algebra. Graph morphisms deliver the basis for typing and the definition of rules and transformations. All graphs of a language are typed over a given type graph via a type morphism. Rules

$(r : L \leftarrow K \rightarrow R)$ are specified using the double pushout approach, where L defines the pattern, that shall be found in a graph, K shows all remaining elements after deleting some elements of L, and finally R contains all preserved plus added elements. Application conditions in positive, negative, and general form restrict the application of a rule to graphs, which either have to contain a demanded pattern or are not allowed to. A rule is applicable, if the match from L to the graph G fulfills the gluing condition and all application conditions. The type graph includes an inheritance graph, which defines all generalization relations between the node types. This leads to a more compact definition of rules, as one abstract rule specifies a set of corresponding rules for all specialized node types. A language *Lang* is then defined by a type graph TG with inheritance, a start graph $S \in Lang$, and a set of abstract rules. Its elements are generated by applying rules to S and the relationship between graph grammar languages with abstract rules and inheritance on the one hand and with concrete rules on the other hand is used in the sense of [1]. Using transformation units [16] for creating complex language elements by a graph grammar is defined as controlled graph grammar in [12] and replaces the set of rules by a set of transformation units and the start graph by a set of start graphs.

2.1 Class Diagrams

The general structure of class and sequence diagrams is defined by the type graph TG_{CSD}. Figure 2 shows the important parts of it for class diagrams containing classes, their features, associations and inheritance relations. The gray marked node ConnectableElement connects this type graph component with the main part for sequence diagrams in Figure 5. A simple version of

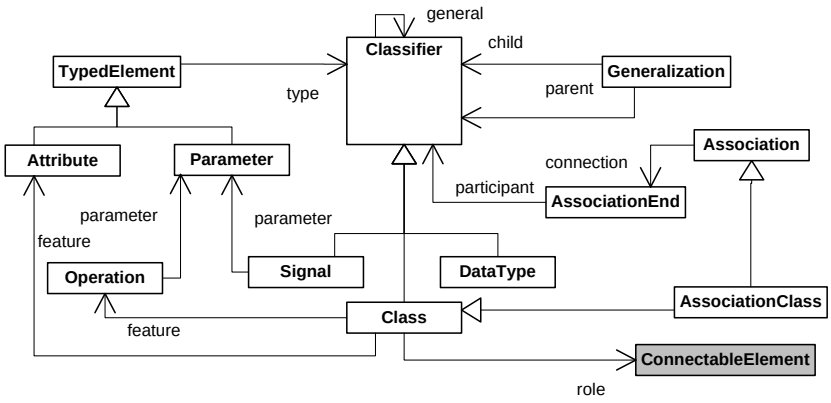


Fig. 2. Part of the type graph TG_{CSD}: main elements of class diagrams

transformation units of [16] combines different rules and imported units with the control structures ";" for sequential application and "!" to demand, that a rule or unit has to be applied as long as possible. For example the simple transformation unit "InsertGeneralization()" specifies, that a class transmits its features to another class and is shown in Figure 3. It imports the rules

"Generalization()" and "General()". After creating a new generalization the second rule is applied as long as possible to achieve again a transitive closed structure.

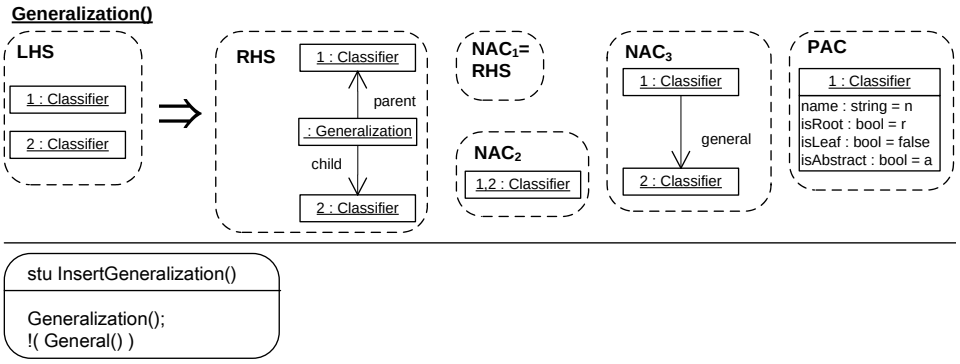


Fig. 3. Simple transformation unit for inserting a generalization

The node type Generalization connects a parent node with its child and is created via the rule "Generalization()" in Figure 3. As the inheritance relation shall be acyclic, a generalization relation in the opposite direction is strictly prohibited by the negative application condition NAC_3, where application conditions are used in the sense of [5]. The positive application condition PAC ensures, that the super class is not a leaf - a class, which is not allowed to transmit to further classes. Prevention of a double defined connection or a generalization link from one class to itself is handled by the other conditions NAC_1 and NAC_2. As Classifier is a generalization of Class, Datatype, and Signal this abstract rule implies nine concrete rules for each combination of the specializations.

Edges of type general supply the transitive generalization relation of all inheritance connections. These edges are created via the rule "General()" in Figure 4, where the positive application condition PAC_2 is used for inserting transitive links. Parallel edges are prevented by NAC and the condition PC allows to generate an edge because of a direct connection or a transitive one.

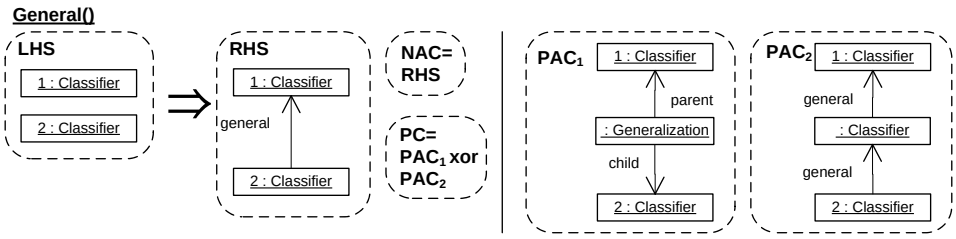


Fig. 4. Rule for creating transitive generalization relations

The simple transformation unit "InsertGeneralization()" in Figure 3 combines the two rules and allows multiple inheritance without cycles. The acyclic structure is demanded by the following constraint for Classifiers in the UML specification. It is mentioned exemplary to show how we argue that our graph grammar generates well-formed instances only.

```
[2] Generalization hierarchies must be directed and acyclic.
A classifier cannot be both a transitively general and transitively
specific classifier of the same classifier.
not self.allParents()->includes(self)
```

2.2 Sequence Diagrams

The main part of the type graph for sequence diagrams is shown in Figure 5, where arrows with closed arrow heads define inheritance realtions. Interacting objects are specified as ConnectableElements, which are already contained in the previous shown type graph component for class diagrams in Figure 2, and represents a role of a Classifier. A Lifeline is connected to anchor points of type OccurrenceSpecification on which elements like Messages can be attached. CombinedFragments are container structures to define control structures, like alternatives, loops, and parallel regions. Their content is structured in InteractionOperands, whose choice may be restricted by Constraints.

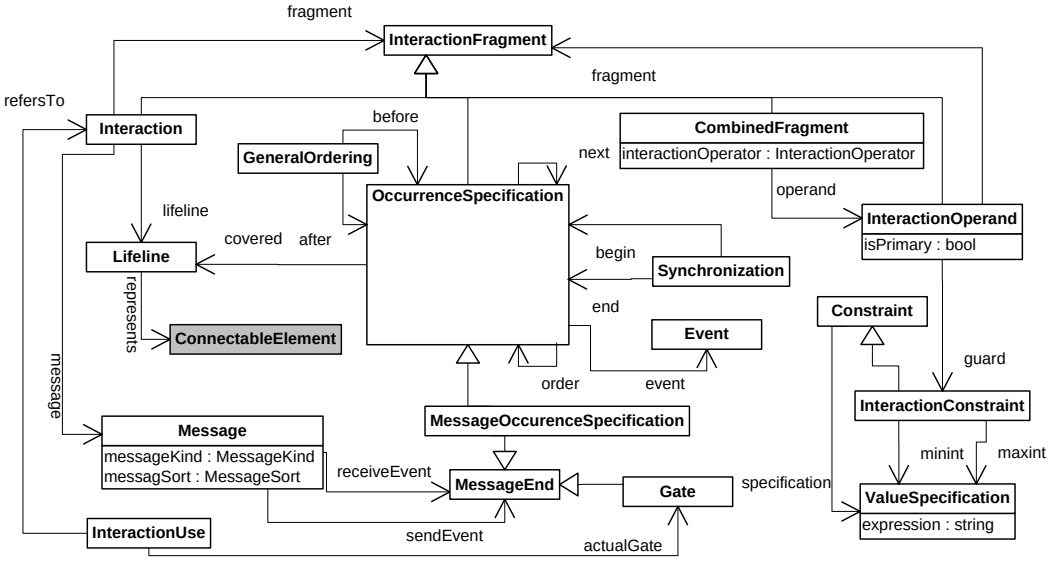


Fig. 5. Part of the type graph TG_{CSD}: main elements of sequence diagrams

Messages of sequence diagrams may be sent synchronously implying that the sender is not allowed to send other messages before receiving a reply. But the UML specification for interactions does not define a relation between these two message types. For this reason the language L_{CSD} additionally includes the node type Synchronization, which marks the beginning and the end of a synchronized interval. InteracitionUses allow to reuse existing sequence diagrams.

GG_{CSD} is fully presented by the two components GG_{CD} and GG_{SD} in [12] and contains more than 70 rules. Figure 6 shows a simple rule, which creates a Lifeline for an object and connects it to the ConnectableElement specifying the role this object executes in the interaction. Additionally it is linked to the enclosing interaction and a first anchor point is inserted. The negative

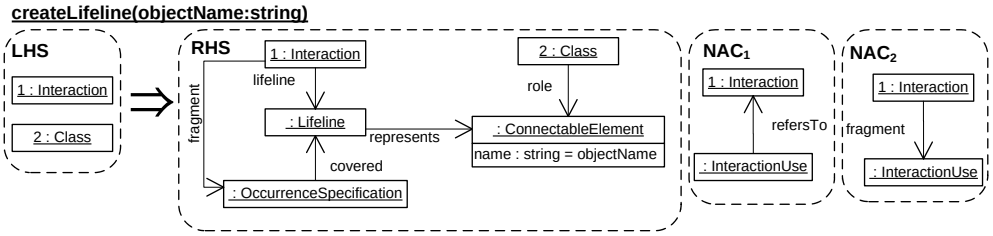


Fig. 6. Rule for creating a lifeline

application conditions NAC_1 and NAC_2 prevent an application of the rule, if the interaction is connected to an interaction use. They ensure, that hierarchical structured interactions remain consistent. This restriction in the order of the editing steps could be eliminated by a transformation unit including more complex rules.

3 Validation of the Graph Grammar

3.1 Testifying Multiplicity, OCL and General Constraints

Multiplicity constraints are respected by the rules of the graph grammar, which is argued at each relevant part of the UML meta-model in Chapter 3 of [12]. The implementation of the multiplicities into the rules is handled mainly by application conditions and well-formedness rules are also argued to be valid, independently of their formulation by natural language only or OCL.

3.2 Model Transformation: $L_{CSD} \rightarrow UML$

The abstract syntax of class and sequence diagrams is defined by GG_{CSD} and strongly corresponds to the definition of UML. As the rules of the graph grammar follow the UML well-formdeness rules, which was described before, the model transformation from each element of L_{CSD} to the corresponding diagram in UML syntax is simple and short. Some additional elements are deleted and bidirectional edges, which are redundant in their grade of information, are added and it is shown, that the transformation terminates and is confluent. The validation of the existing OCL constraints by a formal transformation and check will be available in the future.

3.3 Validation by Benchmarks

To show the coverage of UML features by GG_{CSD} common examples have been selected and its abstract syntax was generated by the grammar. The examples mainly belong to a paper of the IBM Rational Library [2] and are therefore independent benchmarks. They are concretely presented in Chapter 7 of [12] including the shown example in Figure 1 and the sequence of applied rules leading to the instance is given for every diagram. Covered features are for instance InteractionUses to reuse existing sequence diagrams an concurrent

ExecutionSpecifications for defining that an object calls a method which calls a subroutine. Scenarios with parallel or alternatively occuring fragments are other examples.

3.4 Eliminated Problems

The UML specification contains some inconsistencies and mistaken definitions. For example the following constraint occurring on page 476 in [10] is equivalent to true:

```
[2] The selector for a Lifeline must only be specified if the referenced
Part is multivalued.
(self.selector->isEmpty() implies not self.represents.isMultivalued()) or
(not self.selector->isEmpty() implies self.represents.isMultivalued())
```

Instead of the junction "or" it should contain "and". Furthermore the specification of arguments for messages and interaction uses in the meta model is inconsistent. On the one hand a "ValueSpecification" is possible, on the other hand an "Action". GG_{CSD} defines typed Elements as possible argument for both, including the specializations: "ValueSpecification", "Parameter", and "Attribute". A last example is the gap of information for the relation of a synchronous message and its reply mentioned in Subsection 2.2. All detected problems are solved in the graph grammar. Besides changing the text of OCL constraints also some connections and nodes in the meta-model had to be rearranged or inserted to cover the information of a diagram correctly.

4 Future Work and Conclusion

The abstract syntax of a visual model specifies all its semantic relevant properties in a very granular structured way leaving out all layout information. L_{CSD} with its non-descriptive but constructive definition GG_{CSD} offers possibilities to generate well defined specifications of UML in abstract syntax, which can be used directly in the following ways.

4.1 Model Transformation

The generated graphs by GG_{CSD} provide a formal basis to define transformations from L_{CSD} to some target language L_2 using graph transformations as described in [4]. As the source elements were created constructively no constraints have to be checked to ensure the syntactic correctness. Therefore the grammar can also be used for automatic generation of test cases used for model transformations from sequence diagrams.

4.2 Semantics, Simulation, and Animation

A formal semantics of L_{CSD} is planned to be applied, for example using Object-Oriented Transformation Systems (OOTS), where OOTS are an object-oriented variant of transformation systems of [11,14]. Simulation of a specification can

be realized by a transformation to an operational semantics, which also allows animation. All or a selection of possible sequences, defined by sequence diagrams, can be tested to show on the one hand the behavior of the modeled component and on the other hand liveliness, safety, and security properties.

4.3 Editor

In a next step the grammar shall be extended to deliver enough editing rules to automatically generate a syntax directed editor. The TIGER project [8] develops an Eclipse plug-in, which allows defining a graph grammar, connecting the abstract syntax with concrete layout information and generating a syntax directed editor for the language as new Eclipse plug-in. This editor can be used for modeling in the common concrete syntax but generating automatically the precise structured abstract syntax.

References

[1] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, *Proc. Fundamental Aspects of Software Engineering 2004*, volume 2984. Springer LNCS, 2004.

[2] Donald Bell. "UML's Sequence Diagram". URL: http://www-128.ibm.com/developerworks/rational/library/3101.html, 4(th) August 2005.

[3] Werner Damm and David Harel. *"LSCs: Breathing Life into Message Sequence Charts"*. Number 19(1):45-80 in Formal Methods in System Design. 2001.

[4] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 214–228. Springer Verlag, 2005.

[5] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, LNCS 3256, pages 287–303, Rome, Italy, October 2004. Springer.

[6] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in TCS. Springer, 2006. to appear.

[7] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*. LNCS 3256, Springer, 2004.

[8] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as eclipse plug-ins. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, Long Beach, California, USA, 2005.

[9] Object Managment Group et al. *Meta Object Facility (MOF) 2.0 Core Specification, available specification (ptc/04-10-15).* Object Managment Group, August 2005. URL: http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-15.pdf.

[10] Object Managment Group et al. *"Unified Modeling Language: Superstructure version 2.0, Specification (formal/05-07-04)".* Object Managment Group, August 2005. URL: http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf.

[11] M. Grosse-Rhode. Semantic Integration of Heterogeneous Software Specifications. In W. Brauer, G. Rozenberg, and A. Salomaa, editors, *Monographs in Theoretical Computer Science, An EATCS Series*, 2003.

[12] Frank Hermann. "Typed Attributed Graph Grammar for Syntax Directed Editing of UML Sequence Diagrams". diploma thesis, Technical University of Berlin, Department for Computer Science, 2005. URL: http://tfs.cs.tu-berlin.de/Diplomarbeiten/TFSdipl/05-F-Hermann.pdf.

[13] Olaf Kluge. *Compositional Semantics for Message Sequence Charts based on Petri Nets.* PhD dissertation, Technical University of Berlin, Department of Electrical Engineering and Computer Science, May 2002.

[14] Andreas Kniep. "Object-Oriented Transformation Systems". diploma thesis, Technical University of Berlin, Department for Computer Science, 2005.

[15] Oliver Köth and Mark Minas. Abstraction in Graph-Transformation Based Diagram Editors. In *Graph Transformation and Visual Modeling Techniques - GT-VMT 2001*, volume 50 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[16] Sabine Kuske. *Transformation Units - A Structuring Principle for Graph Transformation Systems.* PhD dissertation, University of Bremen, Department of Mathematics and Computer Science, February 2000.

[17] Yong Xia. *A Language Definition Method for Visual Specification Languages.* PhD dissertation, University of Zürich, Department of Economics, January 2005. URL: http://www.ifi.unizh.ch/ifiadmin/staff/rofrei/Dissertationen/Jahr_2005/thesis_xia.pdf.