Application to NWO-EW Open Competition 2004, Second Round

1a Project Title: Graphs for Software Language Definitions

1b Project Acronym: GRASLAND

1c Principal Investigator: Dr. ir. A. Rensink, University of Twente

2a Summary. In the context of the MDA (Model Driven Architecture) methodology for designing maintainable software systems, model transformation is a central concept. Models are used to describe the system in all phases of development and on various levels of abstraction; they are specified in diverse (modeling and programming) *software languages* (SLs). Model transformations typically introduce concrete, implementation specific details.

Such transformations are intended to be *correctness preserving*: they should not introduce errors or essential changes. This, however, can be guaranteed only if the meaning of the SLs involved is defined with sufficient precision. Unfortunately, this is often lacking: many SLs have a well-defined syntax but only an informal semantics. A primary reason for this is that MDA does not include a general method for easily and consistently defining the subtler aspects of SLs, such as their semantics.

The purpose of this project is to define a *meta-language* in which all aspects of SLs, besides their concrete syntax, can be defined in a consistent manner. As a common formal foundation of this meta-language we propose *graphs* and *graph transformations*, which we believe to be powerful enough to capture all relevant SL aspects. This meta-language will enable us to provide semantic definitions of the source and target SLs involved in a given model transformation on a compatible basis; this in turn will enable us to precisely formulate and check the requirement of correctness preservation. We believe these abilities to be essential in realizing the full potential of MDA.

2b Lekensamenvatting. In de context van MDA (Model-Driven Architecture), een tamelijk recente methodologie voor het ontwerpen van goed onderhoudbare software-systemen vaarvoor een groeiende belangstelling bestaat, neemt het transformeren van modellen een centrale plaats in. Het gaat hierbij om modellen die het te ontwikkelen software-systeem in een bepaalde ontwerpfase en op een bepaald abstractienivau beschrijven; hun transformatie betreft bijvoorbeeld het toevoegen van concrete, implementatiespecifieke details.

De bedoeling is dat zulke modeltransformaties “correctheidbewarend” zijn, dat wil zeggen geen fouten of wezenlijke veranderingen in het gemodelleerde systeem introduceren. Zekerheid hieromtrent kan echter alleen bestaan als de betekenis van de gebruikte modeltalen voldoende duidelijk gedefinieerd is. Helaas ontbreekt het hier in de praktijk vaak aan: veel modeltalen hebben wel een goedgedefinieerde syntax maar slechts een informele semantiek. Als reden hiervoor is aan te voeren dat er in de context van MDA geen algemeen geaccepteerde methode bestaat om subtielere taalaspecten, zoals semantiek, op eenvoudige en consistente manier te definiëren.

Het doel van dit project is de definitie van een “meta-taal” voor het definiëren van softwarebeschrijvingstalen; een taal dus waarin (naast de concrete syntax) ook andere aspecten van model- en programmeertalen gedefineerd kunnen worden. Aangezien de basis voor de semantiek van alle aldus gespecificeerde modeltalen hiermee gelijk komt te liggen, maakt dit het mogelijk om de eis dat modeltransformaties correctheidbewarend moeten zijn formeel vast te leggen, en deze eigenschap voor gegeven transformaties formeel te bewijzen. Dit is een noodzakelijke stap in het realiseren van de potentiële mogelijkheden van MDA.

3a) Informaticadisciplines:

3. Software Engineering: 3.2 (specificatiemethoden), 3.5 (interoperabiliteit), 3.6 (ontwikkeltools)
6. Fundamenten: 6.3 (semantiek), 6.5 (formele methoden)

3b) NOAG-i thema’s: Software Engineering (SE), Algorithms and Formal Methods (AFM)

4 Composition of the Research Team. The researchers that will be involved in this project are listed in the following table. Prof. dr. ir. M. Akşit will act as promotor of the OiO.

Name (title)	Position	Expertise	Institute	fte
–	OiO (this project)	–	Univ. Twente	1.0
M. Akşit (prof.dr.ir.)	HL Software Engineering	Software engineering	Univ. Twente	0.0
A. Rensink (dr.ir.)	UHD Software Engineering	Graph transformation	Univ. Twente	0.1
A. Kleppe (drs.)	Consultant	Model-Driven Architecture	Klasse Objecten	0.1

5 Research School. The Software Engineering chair is a member of the research school: Instituut voor Programmatuurkunde en Algoritmiek (IPA).

6 Description of the Proposed Research

6.1 Context of the Proposed Research

A widely recognized proposal for combating the maintenance and evolution problems faced in software engineering is the *model driven approach*, brought to the world's attention by the OMG's Model Driven Architecture (MDA) framework [28]. MDA builds upon and greatly extends UML [46]; its cornerstones are meta-modeling and model transformation.

Key concepts in MDA are: *model*, *language*, *model transformation*, and *platform*. The relationships between these concepts are shown in Figure 1 (taken with permission from Fig. 8-8 of [28]).

Within MDA, a model is defined as a description of a software system written in a certain language, preferably a well-defined language (see Problem Description for our definition of a well-defined language). A model *transformation* converts one model into another model. Often the source

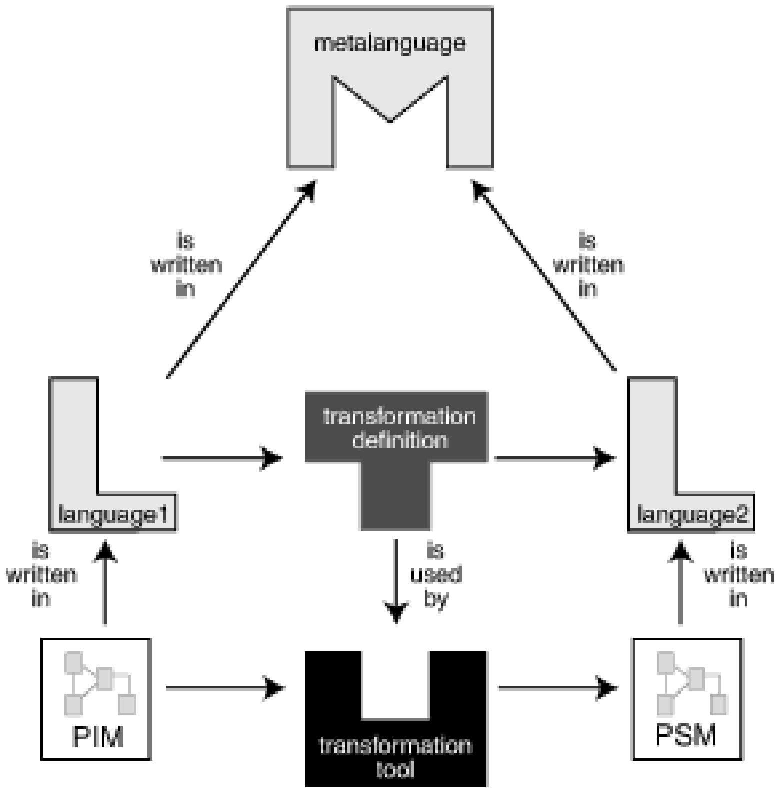

Figure 1: Concepts and relationships in the MDA framework

model is a so-called Platform Independent Model (PIM), whereas the target model is a Platform Specific Model (PSM). The concept of platform in this context can be described as the specific software and hardware that constitute the execution environment of the software system. Model transformations are automated and carried out by transformation tools. The input for the tools are transformation definitions.

Although MDA has received a warm welcome by the software developing community, currently the MDA vision cannot be implemented to its full extent, for the following reasons (among others):

1. UML, which is the language that is most often used for writing PIMs, lacks coherence and a well-defined semantics (see, e.g., [36, 22, 18, 12]).
2. The language in which model transformations should be written is not yet standardized. The OMG standardization process of what is now called the Query-View-Transformation (QVT, [34]) is still on its way, and the timing of its completion is uncertain.
3. The languages commonly used for writing PSMs, which are typically ordinary programming languages and extensions thereof, are usually not defined in a manner suited to build transformation definitions. Often a BNF grammar and a compiler are the only definitions provided.

This project addresses the third of these problems (and thereby implicitly part of the first, since UML is one of the SLs in question).

6.2 Problem Description

In the MDA Guide Version 1.0 ([28]) the concept of model transformation is defined as follows:

> *Model transformation is the process of converting one model to another model of the same system.*

The fact that input and output models should describe *the same system* implies that model transformations are to preserve the meaning of the model(s) being transformed (see also [24]). For the meaning of the models we have to look at the semantics of the languages in which they are described. From now on we call these *Software Languages* (SLs), be it for implementation (i.e., programming languages) or specification.

The semantics of current SLs are often imprecise or even completely lacking (except through the “tool semantics” provided by the compiler). Therefore it is difficult to determine with any degree of rigor whether a given model transformation really meets the above criterion. We hold that for building useful model transformations, it is imperative that the SLs in which the source and target model are written are *well-defined*. We call a language well-defined when the following aspects are clear and unambiguous:

- The concrete syntax: i.e., the notation used for writing models;
- The abstract syntax: i.e., the concepts available for constructing models;
- The static semantics: i.e., the elements and conceptual relationships of the systems described by the models;
- The dynamic semantics: i.e., the behavior of the systems described by the models, as can be examined by the changes between snapshots taken at subsequent points in time;
- The relationships between these four aspects.

As remarked above, many SLs used in the context of MDA lack one of more of these aspects.[1] Especially the requirement that static and dynamic semantics are closely related and should not be defined separately is important for SLs. Software is by definition dynamic. Even simple data storage systems have an important dynamic side (the so-called CRUD functions). A definition of an SL therefore cannot be complete without a definition of the dynamic semantics.

The established manner for defining concrete syntax is (E)BNF. The OMG standard called Meta-Object Facility [33] has been proposed as a means for defining abstract syntax. What is lacking, however, is a general way to define the relationship between concrete and abstract syntax, for instance along the lines of attribute grammars. Even less clear in the context of MDA is a convenient means for consistently defining static and (operational) dynamic semantics, in a way that is compatible with the abstract syntax — even though the essential theory has been known for some time (e.g., [48, 31]).

Note that it is not even necessarily the case that there is only one possible semantics to a given (concrete and abstract) syntax. A case in point are UML statecharts, for which a large number of semantics definitions have been published (e.g., [27]). Due to this plethora, if the intended semantics is not explicitly given it can still be unclear whether a given system built from a statechart indeed conforms to its intended specification.

What MDA needs in order to fulfill its promise is a special-purpose *language definition language* (LDL) for defining all four SL aspects in a compatible framework; in other words, a meta-language in which multiple SLs can be defined so as to uphold the well-definedness requirements stated above. The goal of this project is the definition of such a meta-language.

6.3 Approach

In this section we propose a solution strategy to the problem described above, based on two key principles. (We use L to denote an arbitrary given SL and p for a "program" [model, instance] of L.)

1. We explicitly recognize, for each of the SL aspects Asp identified above, on the one hand the meta-model Asp_L defining that aspect for L, and on the other hand the *process* of constructing model instances $\mathsf{Asp}(p)$ of Asp_L for concrete programs p of L.
2. We propose to use *graphs* and *graph grammars* to define the models, meta-models (interpreted as type graphs) and transformation processes for all SL aspects.

Example. *We demonstrate these principles using a running example. We define a toy Software Language* XL, *consisting just of label declarations and goto statements, with the following grammar:*

$$\begin{array}{rcl}
\mathsf{Prog} & ::= & \mathsf{Stat}\ `\texttt{;}'\ \mathsf{Prog} \quad | \quad \mathsf{Stat}\ `\texttt{.}' \quad . \\
\mathsf{Stat} & ::= & \mathsf{Label} \quad | \quad \mathsf{GoTo} \quad . \\
\mathsf{Label} & ::= & \mathsf{Id} \quad . \\
\mathsf{GoTo} & ::= & \mathsf{go}\ \mathsf{Id} \quad .
\end{array}$$

At a statement of the form go x*, the system jumps non-deterministically to any of the places where the label* x *is declared. The following is our running example program* xp *in this language:*

```
a; go a; a.
```

Concrete Syntax. The concrete syntax of L is typically given in EBNF, which is a string grammar. The grammar gives rise to a parser for L which can construct a syntax tree $\mathsf{CoSy}(p)$ from syntactically

[1]It should be remarked that there is no universally agreed-upon terminology: although we find it useful to distinguish these aspects for modeling purposes, others prefer to call the first three *syntax*, and only the fourth *semantics*; see, e.g., [29].

correct instances p of L. The structure of the syntax trees is implicit in the rules of the grammar: the nodes are terminals and non-terminals, and the branching structure is derived from the placement of the corresponding textual fragments within the original p.[2]

For the purpose of this project we require an explicit specification of the meta-model CoSy_L of the concrete syntax trees of L, in addition to the rules of the grammar; moreover, the grammar is taken as a special case of a graph grammar.

Example. *The first column of Figure 2 shows the concrete syntax meta-model* $\mathsf{CoSy}_{\mathsf{XL}}$ *and the concrete syntax tree* $\mathsf{CoSy}(\mathsf{xp})$. *Note that we have made the ordering of the children in the syntax tree explicit.*

Abstract Syntax. As stated in the previous section, in the context of MDA the abstract syntax for a given SL L is typically defined through a (possibly annotated) diagrammatic meta-model; in the setting of this project, this corresponds to a type graph AbSy_L. Instances $\mathsf{AbSy}(p)$ (for p a program of L) are to be derived from the corresponding concrete syntax tree $\mathsf{CoSy}(p)$; the derivation can be defined again through a graph transformation from CoSy_L-typed graphs to AbSy_L-typed graphs. What happens during this transformation is obviously specific to L, but typically it will involve the merging and removal of redundant syntactic elements, as well as name resolution.

Example. *The abstract syntax of* XL *rearranges the statement types in an abstraction hierarchy and unifies all label identifiers. The resulting type graph* $\mathsf{AbSy}_{\mathsf{XL}}$ *as well as the example instance* $\mathsf{AbSy}(\mathsf{xp})$

[2]What we describe may be called a *concrete syntax tree*; it is actually more common to work with so-called *abstract syntax trees*, in which those syntactic elements that were merely there to ease the job of the parser have been removed. This is in fact a step towards our *abstract syntax graphs* described below. To avoid confusion we concentrate on concrete syntax trees in this proposal.

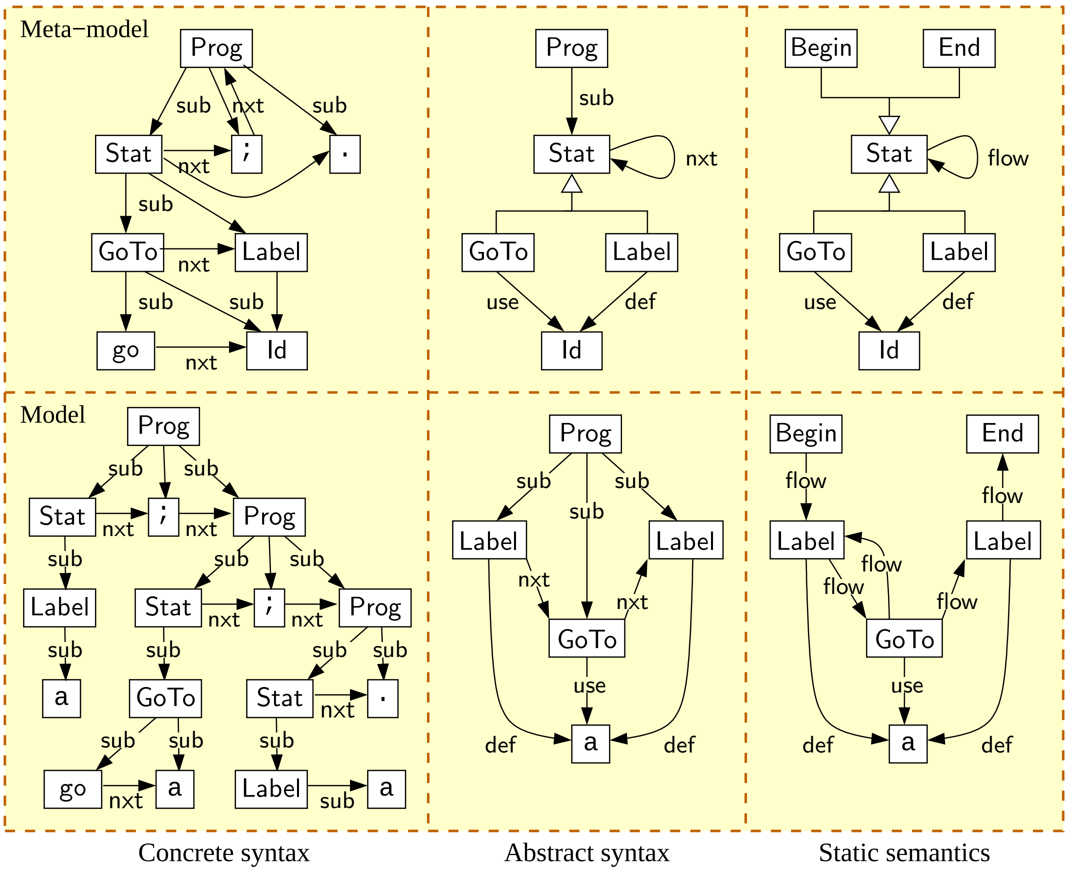

Figure 2: Example meta-models and models for the program xp: *“*`a; go a; a.`*”*

are shown in the second column of Figure 2. The transformation from the concrete to the abstract syntax involves removing the intermediate Stat*-nodes (while maintaining the* nxt*-ordering) and merging* Id*-nodes with identical names.*

Static Semantics. The static semantics of a program is typically expressed using a collection of models, including, for instance, flow, data, call and dependency graphs. In this project we take the position that these can fruitfully be combined into a single model, possibly at the loss of ready visualization but certainly at the gain of consistency. The separate models can be recaptured as *views* upon this combined model. Again, the static semantics of a language L can be specified through a meta-model, StSe_L; instances can be derived from the abstract syntax graphs through graph transformation.

Example. *The static semantics of* XL *adds* Begin- *and* End*-nodes and control flow to the abstract syntax graph, as mentioned above. The meta-model* $\mathsf{StSe}_{\mathsf{XL}}$ *and example model* StSe(xp) *are given in the third column of Figure 2. The transformation rule for a* Label*-statement just converts the outgoing* nxt*-edge into a* flow*-edge; the rule for a* GoTo*-statement creates a* flow*-edge to all* Label*-statements* def*ining the label identifier* used *by the* GoTo*.*

Dynamic Semantics. In contrast to the aspects discussed so far, the dynamics of a software system will be described not by one single model, but by a collection of snapshots with a transition relation between them. Each of these snapshots can be modeled as an extension of the static semantics model with run-time information; in language implementation terms, the run-time information consists of (abstractions of) the stack and heap of a program. Together this gives rise to a state-transition system, which can be seen as the behavioral model of the system.

Once more, we can capture the structure of the snapshots in a meta-model, DySe_L. It will generally extend the static semantics (StSe_L) with the types of run-time information mentioned above. Transformation rules in this context play a different role than before: rather than collectively describing the construction of one model for a given program p from another (concrete to abstract syntax to static semantics), now each rule will describe a (small) step in the execution of p; in other words, each rule acts as a graph-based virtual machine instruction.

Example. *Figure 3 shows the meta-model* $\mathsf{DySe}_{\mathsf{XL}}$ *(which adds a node type* VM *for the "virtual machine" executing a given program and an edge type* pc *for the "program counter" pointing to the statement under execution) as well as the transition system modeling the behavior of* xp*.*

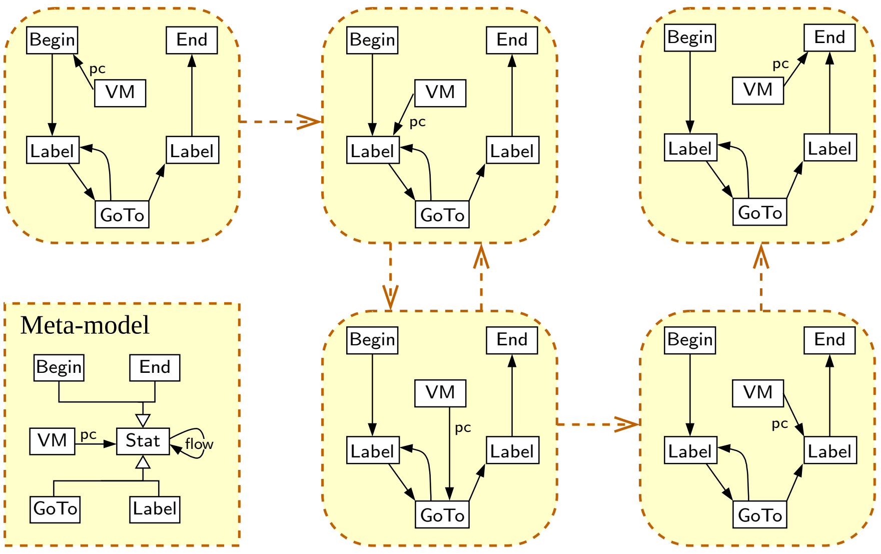

Figure 3: Dynamic semantics of the example program

Meta-language As discussed and illustrated above, the key idea is to describe all essential elements of SL definitions using (type) graphs, and the relationship between these elements using graph transformations. Thus, graphs and graph transformations are the core of the meta-language (the LDL) which this project is out to define. We rely on the extensive research in graph transformations to provide us with results on simulation and verification and algorithms for automatic parsing, synthesis and transformation of models. This establishes a common basis for a more precise definition of language aspects, and thereby a foundation on which a theory of correctness-preserving, intra- or inter-language model transformations can be built — thus addressing, and hopefully closing, one of the main gaps in the road towards realizing the benefits of the MDA approach.

6.4 Project Objectives

We define the following steps/milestones in the project:

1. A case study consisting of a *programming* language definition, addressing all aspects mentioned above, for a realistic fragment of an existing programming language, such as Java. (Preliminary work in this direction has been done in two MSc theses: see [8, 23].)
2. A case study consisting of a *specification* language definition, addressing all aspects mentioned above; for instance, UML state diagrams. The intention here is *not* to define a new semantics but to formulate one of the existing semantic definitions in terms of graphs and transformations, along the lines described above; see, e.g., [47] for the feasibility of such a semantics.
3. An initial definition of an LDL in which at least the language concepts identified in the case studies can be specified. This requires an understanding of these concepts on an abstract enough level. At this stage, especially the abstract syntax and semantics of the LDL are of interest.
4. A concrete syntax for the LDL, once the initial definition (previous step) is stable.
5. In parallel with two the steps above and partly driving them, the (re)formulation of the case study language definitions (steps 1 and 2) in terms of the LDL being developed.
6. The development of tools for compiling LDL "programs" (i.e., SL specifications), which means the creation of the meta-models and graph transformation systems.
7. The complete definition of the case study languages (steps 1 and 2) in the LDL.

All in all, the outcome of the project is a language definition language with formal semantics and (light-weight) tool support, and the formal definition of a realistic programming and a specification language in the LDL.

6.5 Other Related Work

This project has clear connections with work in graph rewriting and other approaches to the specification and verification of behavioral semantics, as well as the development of formal foundations for UML and MDA.

Graph rewriting. An authorative and comprehensive overview of the theory and practice of graph rewriting can be found in [45, 15]. Graph rewriting has been applied before in models of computation; see, e.g., [7, 16, 14, 11]. Our own project GROOVE [20, 4] investigates the combination of graph rewriting and verification techniques (in particular, model checking).

Within software engineering there are several currently very active areas of application of graph rewriting, such as software evolution (e.g., [30]) and UML model and meta-model semantics (e.g., [19, 21, 26, 47]).

Behavioral semantics Both process algebra ([10]) and programming language theory (e.g., [48]) have developed a rich theory of behavioral semantics, from which we intend to benefit in this project. In fact the graph transformation semantics we intend to pursue is obviously a form of operational semantics, and it seems natural to incorporate a form of SOS (Structural Operational Semantics, see [6]) into our LDL. Another connection exists to *term graph rewriting* [9], which is indeed a technique for graph transformation-based operational semantics of functional languages.

Formal foundations for UML. There are many research groups investigating semantics for UML; e.g., the aforementioned statechart semantics [27] and graph rewriting approaches [26], but also the work carried out in the pUML group [17, 25, 37]. The difference with this project is that we are not interested in any particular semantics, but with the means to define them systematically for a given SL.

6.6 Embedding within own existing activities

Software Engineering chair. Within the software engineering group, a number of topics are being studied, some pertaining to various stages of the software development process, others to the process itself. The principal themes are (i) *Aspect-Oriented Software Composition*, (ii) *Synthesis Based Software Architecture Design*, (iii) *Soft Computing applied to Software Development*, and (iv) *Design for Correctness*. The current proposal falls into (iv) and (to a lesser degree) in (ii).

Klasse Objecten. The core activities of Klasse Objecten are consultancy and training in software design through modeling, using UML, OCL and MDA. From 1997 Klasse Objecten has been actively involved in creating the UML and OCL standards. The knowledge gained by this participation is crucial for the company. In the coming years the subject of MDA will become more and more important to (clients of) Klasse Objecten. Therefore it is necessary for Klasse Objecten to be involved in the development of stable foundations for the application of MDA.

Principal investigator. Arend Rensink moved to the area of graph transformation in 2001, after having done research in process algebra-related fields for close to a decade. His particular interest is the application of graphs as software models and graph transformations to describe the dynamics of those models; this raises opportunities for software modeling and verification that are new in the context of the (otherwise quite well-established) area of graph transformations. Apart from a number of conference publications (see reference section), the research has resulted in the development of a tool environment called GROOVE [38], to be developed further in an NWO project of the same name [20] started recently. The current project dovetails onto GROOVE very nicely, since the LDL to be developed here will allow to provide graph transformation semantics to SLs that can then be fed into the GROOVE tool.

7 Work Programme

Planning. The tasks described below refer to the enumeration in the project objectives (Section 6.4).

Tasks	1	2	3	4	5	6	7
Year 1	X	X	X				
Year 2			X	X	X	X	
Year 3					X	X	X
Year 4	— Consolidation and PhD thesis —						

Education. It is anticipated that the OiO takes part in the regular IPA courses. Depending on the background of the OiO, he will also participate in a selection of (international) Master courses.

9 Literature

Key publications of the research team

[1] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? on the automated verification of linked list structures. In *The 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Lecture Notes in Computer Science. Springer-Verlag, 2004. To appear.

[2] A. Kleppe, J. Warmer, , and W. Bast. *MDA Explained; The Model Driven Architecture: Practice and Promise*. Object Technology Series. Addison-Wesley Pub. Co., 2003.

[3] A. Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *Programming Languages and Systems — European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 2004.

[4] A. Rensink. 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)*, volume 3063 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[5] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Model Ready for MDA*. Object Technology Series. Addison-Wesley Pub. Co., second edition, 2003.

Other references

[6] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In Bergstra et al. [10], chapter 3, pages 197–292.

[7] M. Andries, 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*, 34:1–54, 1999.

[8] M. Arends. Graph grammars for Java bytecode simulation. Master's thesis, University of Twente, 2003.

[9] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe, Volume I*, volume 259 of *Lecture Notes in Computer Science*, pages 142–158. Springer-Verlag, 1987.

[10] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[11] A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating java into graph transformation systems. In Parisi-Presicce et al. [35].

[12] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal? UML shortcomings for coping with round-trip engineering. In R. France and B. Rumpe, editors, *Proc. UML '99 —- The Unified Modeling Language: Beyond the Standard*, volume 1732 of *Lecture Notes in Computer Science*, pages 630–644. Springer-Verlag, 1999.

[13] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer Academic Publishers, 2002.

[14] F. L. Dotti, L. Foss, L. Ribeiro, and O. M. dos Santos. Verification of distributed object-based systems. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, 2003.

[15] 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.

[16] H. Ehrig and G. Taentzer. Computing by graph transformation, a survey and annotated bibliography. *Bull. Eur. Ass. Theoret. Comput. Sci.*, 59:182–226, 1996.

[17] A. S. Evans and S. Kent. Meta-modelling semantics of UML: the pUML approach. In *Proc. UML '99 —- The Unified Modeling Language: Beyond the Standard*, volume 1732 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[18] H. Giese, J. Graf, and G. Wirtz. Closing the gap between object-oriented modeling of structure and behaviour. In R. France and B. Rumpe, editors, *Proc. UML '99 —- The Unified Modeling Language: Beyond the Standard*, volume 1732 of *Lecture Notes in Computer Science*, pages 534–549. Springer-Verlag, 1999.

[19] M. Gogolla, P. Ziemann, and S. Kuske. Towards an integrated graph based semantics for UML. In P. Bottoni and M. Minas, editors, *Proc. ICGT Workshop on Graph Transformation and Visual Modeling Techniques*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B.V., Oct. 2002.

[20] Groove: Graphs for object-oriented verification, 2004. Research project funded by the Dutch NWO, Grant 612.000.314.

[21] E. Guerra and J. de Lara. Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In Parisi-Presicce et al. [35].

[22] B. Henderson Sellers and F. Barbier. Black and white diamonds. In R. France and B. Rumpe, editors, *Proc. UML '99 —- The Unified Modeling Language: Beyond the Standard*, volume 1732 of *Lecture Notes in Computer Science*, pages 550–565. Springer-Verlag, 1999.

[23] H. Kastenberg. Software metrics as class graph properties. Master's thesis, Department of Computer Science, University of Twente, July 2004.

[24] A. Kleppe and J. Warmer. Do MDA transformations preserve meaning? an investigation into preserving semantics. In *MDA Workshop, York*, Nov. 2003.

[25] A. Kleppe and J. Warmer. Unification of static and dynamic semantics of UML: A study in redefining the semantics of UML using the pUML OO meta modelling approach. Technical report, Klasse Objecten, 2003. Available from `http://www.klasse.nl`.

[26] S. Kuska, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In M. Butler, L. Petre, and K. Sere, editors, *IFM 2002*, volume 2235 of *Lecture Notes in Computer Science*, pages 11–28. Springer-Verlag, 2002.

[27] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of uml statecharts diagrams. In *The 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems*. Kluwer Academic Publishers, 1999.

[28] MDA guide version 1.0.1, June 2003. Available from `http://www.uml.org`.

[29] B. Meek. The static semantics file. *ACM Sigplan Notices*, 25:33–42, Apr. 1990.

[30] T. Mens. Conditional graph rewriting as a domain-independent formalism for software evolution. In Nagl et al. [32], pages 127–143.

[31] T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors. *On Extracting Static Semantics*, volume 2566 of *Lecture Notes in Computer Science*. Springer, 2002.

[32] M. Nagl, A. Schürr, and M. Münch, editors. *Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[33] OMG. Meta object facility (MOF) specification, Apr. 2002. See `http://www.uml.org`.

[34] MOF 2.0 Query / Views / Transformations request for proposals, Apr. 2004. Available from `http://www.uml.org`.

[35] F. Parisi-Presicce, P. Bottoni, and G. Engels, editors. *Second International Conference on Graph Transformation*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[36] D. C. Petriu and Y. Sun. Consistent behaviour representation in activity and sequence diagrams. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000 — The Unified Modeling Language: Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*, pages 369–382. Springer-Verlag, 2000.

[37] The precise uml group. See `http://www.cs.york.ac.uk/puml/`.

[38] A. Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars. Available at `http://www.cs.utwente.nl/˜groove`, 2003.

[39] A. Rensink. A logic of local graph shapes. CTIT Technical Report TR–CTIT–03–35, Faculty of Informatics, University of Twente, Aug. 2003.

[40] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Proceedings of the* 3rd *Workshop on Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

[41] A. Rensink. Agtive'03: Summary from the outside in. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3063 of *Lecture Notes in Computer Science*, pages 486–488. Springer-Verlag, 2004.

[42] A. Rensink. Representing first-order logic using graphs. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *International Conference on Graph Transformations*, Lecture Notes in Computer Science. Springer-Verlag, 2004. To appear.

[43] A. Rensink. Time and space issues in the generation of graph transition systems. In *International Workshop on Graph-Based Tools (GraBaTs)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2004. To appear.

[44] A. Rensink, Á. Schmidt, , and D. Varró. Model checking graph transformations: A comparison of two approaches. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *International Conference on Graph Transformations*, Lecture Notes in Computer Science. Springer-Verlag, 2004. To appear.

[45] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, Singapore, 1997.

[46] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley Pub. Co., 1999.

[47] D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 2003.

[48] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. The MIT Press, 1993.