

Students can get excited about Formal Methods: a model-driven course on Petri-Nets, Metamodels and Graph Grammars

Pieter Van Gorp

Hans Schippers*

Serge Demeyer

Dirk Janssens

Department of Mathematics and Computer Science

University of Antwerp

{Pieter.VanGorp, Hans.Schippers, Serge.Demeyer, Dirk.Janssens}@ua.ac.be

Abstract

Formal Methods have always been controversial. In spite of the fact that the disbelief about their usefulness has been corrected by a growing number of applications and even more publications, it remains a challenge to demonstrate the strengths and weaknesses of formal methods within the time constraints of a typical semester course. This paper reports on a new course at the University of Antwerp in which the introduction of a new formalism yields a better understanding of previously taught ones. While the exercises are designed to reveal the limitations of the formalisms used, students remain convinced that their formal models have more value than conventional source code.

1 Introduction

Formal Methods have been praised for facilitating the detection of inconsistencies and/or inaccuracies early in the development process. Still, formality is often falsely associated with the exhaustive specification of proofs in special symbols that are hard to read by most stakeholders. Therefore, they are often believed to be very costly and thus only applicable to very specific applications, such as embedded spacecraft software [10, 3]. To prevent the further spread of such myths in industry, software engineering students at the University of Antwerp are confronted with formal modeling languages in a variety of settings.

On the one hand, providing correctness proofs is part of courses on mathematics, databases, computer arithmetics and computability. On the other hand, several courses illustrate that modeling a system before its implementation assists in the early detection of misunderstandings. For example, already after a minimalistic introduction to the Unified Modeling Language (UML), student teams have lively

discussions to reach a consensus on the structure of a domain model before constructing their first distributed network application. Similar discussions are held when student teams are instructed to construct a small compiler. Within this compiler project, students are not allowed to implement a semantical analyzer and code generator using the implicit abstract syntax tree from the parser generation framework directly. Instead, they are required to model a more abstract representation of the source language as a class diagram. The instructor verifies that students do not model implementation-specific concepts and ensures they do not abuse UML compositions and/or association cardinalities. While this exercise does significantly improve the students' knowledge of the UML, the learning curve is rather steep. Therefore, one can hardly expect that students fully appreciate the added value of their modeling task.

Recently, the introduction of a new undergraduate course indicated that this investment in precise modeling did result in more appreciation of further courses in formal methods in general and a model-driven engineering approach specifically. This paper presents that course, called "*Formal Techniques in Software Engineering*", in more detail, using the following structure: Section 2 presents the role, objectives, structure and examination form of the course; Section 3 presents some of the artifacts to be developed by the students and indicates their relation and educational value; Section 4 summarizes the lessons learned after the first year while the final section concludes this paper.

2 Course Design

This section first describes the role of the course within the curriculum. Secondly, it discusses how the students' background and the complexity of supportive tools affects the objectives. Then, a description of the course structure and examination form clarifies how competences are transferred to the students and how students are evaluated.

*Research Assistant of the Research Foundation, Flanders (FWO)

2.1 Role within the Curriculum

As stated in the introduction, modeling is an integral part of the computer science curriculum at the University of Antwerp: after an absolutely fundamental course on discrete mathematics, several courses rely on formal models with laws to derive properties of the domain or system under study.

For example, instead of focussing an introductory “databases” course on querying concrete databases with SQL, systematic procedures are taught to transfer an Entity/Relationship model into a relational model. Moreover, the course illustrates how the formal nature of the latter model allows one to normalize databases automatically. Similarly, a course on computer arithmetics relies on a model of the standard for floating point arithmetics to reason about the correctness of floating point implementations.

While other courses use languages such as Z, B, SDL and statecharts, they leave the definitions of the involved models and languages implicit. In the third and final year of the bachelor program, the new course aims to teach students how different formal techniques relate to one another instead of leaving them isolated within the other, individual, courses. Primarily, students are taught that in model-driven software engineering, software is developed in different languages, at different levels of abstraction and that there are good reasons to do so. Secondly, the course illustrates how metamodeling and model transformation can be used to maintain the consistency between the models expressed in these languages.

2.2 Course Prerequisites and Objectives

This section describes the prerequisites and objectives that were defined when the course was first planned.

Students can enroll in the course provided they have practical programming experience, practical experience in the use of UML class diagrams and a solid understanding of the formal foundations of computer science (logic, formal languages). In practice, this expertise was provided by courses from the first two years of the bachelor program.

Officially, the expected learning outcomes were initially defined as follows:

“Based on formal specifications (logical specifications, statecharts, Petri-Nets) the student should be able to build models expressing the intended functionality of a system, to analyse and to verify these models, and to generate a working implementation from them.”

These objectives were designed with the AndroMDA code generators in mind [2]. Using this code generator from

UML diagrams to Java web applications would bring students in contact with:

- conceptual modeling with UML class diagrams,
- query definition with a subset of the Object Constraint Language (OCL),
- user interface flow modeling with use cases and activity diagrams.

However, supervision of another undergraduate course (in which students have to build a large software system within two semesters) indicated that the use of AndroMDA required a significant learning curve for setting up a correct modeling and build environment. Moreover, despite the significant amount of code generation, students are still required to master the underlying J2EE technologies. Therefore, the use of AndroMDA would put too much weight on the use of a code generator and leave too little more room for teaching how the modeling languages used (class diagrams, activity diagrams, ...), are defined and kept consistent. Since the novelty of model-driven engineering does not consist of using code generation as such, but of adapting code generation environments (by using standard model and code transformation languages), the course objectives were informally adjusted to:

“The student should be able to express the intended functionality of a system from different viewpoints in different formalisms (Petri-Nets, Graph Grammars) and ensure particular properties (boundedness, consistency, ...) of such models. The student should use state-of-the-art transformation techniques (model animation, model translation and code generation) to integrate distinct models and relate them to a complete implementation. The student should experiment with metamodeling in this context, and acquire an understanding of the benefits and limitations of the 4-layer meta model architecture.”

After considering a combination of the DiaMeta [13] and Tiger [9] meta-case tools with the MoTMoT [14] model transformation tool, the AToM³ tool was selected because of its completeness. AToM³ offers mechanisms for metamodeling, concrete syntax definition, model transformation and code generation in a self-contained, Python based, environment [6].

2.3 Course Structure

The course takes place in the last semester of the bachelor program. It divides six European Credits (ECTS [4]) across seven theoretical sessions and eleven lab sessions of

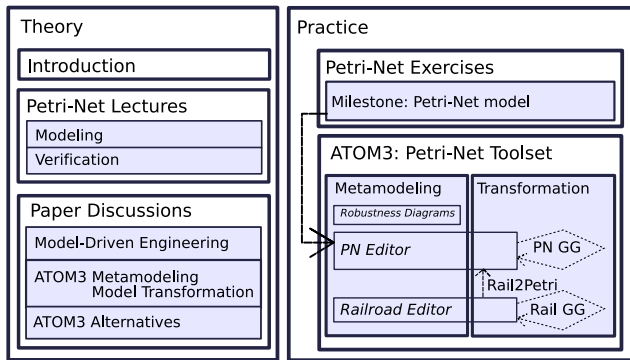


Figure 1. Structure of the Course

two hours per session. The course structure is visualized by Figure 1.

The first part of the theoretical sessions consist of lectures whereas the latter sessions are more interactive: in these sessions, the lecturer leads a discussion on a set of selected papers. The first lecture introduces the students to the Model Driven Engineering paradigm. Based on a concise, yet sufficiently complete textbook [11], the lecturer clarifies the definitions of a model, modeling language, meta-model, transformation definition and a transformation language. The following three sessions introduce the students to the most commonly used variants of the Petri-Net language. After explaining the role of invariants in the analysis of Condition/Event nets, the language is extended to Place/Transition nets with weights and capacities. A number of examples illustrate how the underlying formalism allows one to reason about deadlocks and other properties.

The first three lab sessions make sure that students master Petri-Nets both for modeling and verification. Since students are already heavily loaded with project work from other courses in their final year of the bachelor program, all exercises are made in the controlled context of the lab sessions. In the last Petri-Net session, students are given an assignment that will serve for evaluating their modeling ability. More interestingly, it serves as input for a Petri-Net editor that will be constructed in the second series of practical lab sessions.

The second and final part of the lab sessions distinguish the course from a conventional course on formal methods. In each of these eight lab sessions, students are given a well-defined assignment contributing to the construction of a Petri-Net based toolset using the ATOM³ tool.

Within the first session from this series, the teaching assistant demonstrates the core features of this tool. Moreover, the students are given an idea of what kind of toolset they will construct in the upcoming weeks. In the remainder of the first session, students are already creating their first metamodel, using the Entity/Relationship language.

This first metamodeling exercise consists of the definition of the Robustness Modeling language [16]. This language consists of only five modeling constructs: Actors, Boundaries (Interfaces), Controls (Processes), Entities (Domain elements) and Use Cases. The language is selected because it is related to design rules that students have encountered in other software engineering courses. More specifically, the language includes well-formedness rules stating, for example, that user interface elements should not access persistent elements directly. Students realize that it is better to enforce these constraints by means of a specific modeling language than to check them in implementations based on a general purpose programming language. Moreover, students become familiar with metamodeling. At the end of the first session, students hand in their E/R model defining the Robustness Modeling language along with a sample Robustness Diagram that they created with the generated editor.

In the second ATOM³ session, an editor for Condition/Event and Place/Transition Nets with weights and capacities is constructed. The third section is the first hands-on introduction to model transformation: in this session, students are instructed to define a graph grammar realizing the Petri-Net transition semantics. Again, the introduction from the teaching assistant starts from a demonstration of the result that needs to be achieved. More specifically, a correct grammar is executed on a Petri-Net model of the dining philosophers problem. Students see how different rules are combined to detect whether a transition is enabled and whether tokens have already been moved from input places to output places. To illustrate that the presented techniques are not specific to Petri-Nets, a model of a railroad track is presented and a graph grammar is used to move trains across the tracks. These demonstrations are followed by a brief introduction to the core graph grammatical concepts, like the left- and right-hand side of a rule, the node identification mechanism and rule priorities.

At the end of the third session, students hand in their graph grammar for Condition/Event nets. In session four, students generalize this grammar to the Place/Transition variant of Petri-Nets (allowing more than one token per place). The grammar should handle weights on arcs from places to transitions properly. Although this seems like a minor conceptual extension of the Petri-Net variant, the grammar needs to realize a loop over a number of its rules. Since ATOM³ only supports priorities as a control flow structure, the realization of such a loop is not straightforward. In the fifth session, the grammar is generalized one last time by supporting weights on arcs from transitions to places and capacities on places too. At the end of the fifth session, students should be able to animate a Petri-Net model for access control that is part of the theoretical course material on Petri-Nets.

In session six, students return to metamodeling. They are instructed to model the “Railroad” language that was briefly demonstrated in session three. Since this language contains a set of related concepts (trains move across straight tracks in the same way as they drive through a station), it motivates the need for inheritance in the metamodeling language. Therefore, the Railroad language needs to be modeled with class diagrams instead of with Entity/Relationship diagrams. In sessions seven and eight, the AToM³ lab is concluded with a transformation from Railroad models to Petri-Nets. This assignment challenges students to combine all competences acquired so far. Additionally, they are brought in contact with the additional techniques required to define an exogenous transformation (see Section 3.3). For example, this transformation between different languages introduces students to the need for (and nature of) traceability mechanisms.

The discussions from the four final theoretical sessions are based on three to four papers per session. To prepare for such a paper session, the students should read the papers and answer a few questions. In the lecture itself, the students debate the strong and weak points of a given paper, which results in quite vivid discussions. The goal during these debates is not that one student wins the debate, but rather that all students see the merits and differences in each approach. The final discussion session is based on two papers that generally classify today’s model transformation approaches and two papers that each present an AToM³ alternative in detail. During the debate, students are asked which aspects of their transformations they found difficult to express with AToM³. Moreover, students need to assess what features of other graph transformation languages address these difficulties.

2.4 Examination Form

The course evaluation consists of two parts: permanent evaluation, and an oral exam with a written preparation.

First of all, the solutions to the lab exercises are taken into account, which test one of the course objectives, namely “*The student should use state-of-the-art transformation techniques (model animation, model translation and code generation) to integrate distinct models and relate them to a complete implementation.*” This part of the examination is managed by electronic submissions to the university’s electronic learning platform [17]. Since the AToM³ lab sessions build incrementally upon one another, the teaching assistants have prepared partial solutions corresponding to all AToM³ related deadlines. This can help students that have failed to meet one deadline in catching up for the next deadline. In practice, students have enthusiastically completed some incomplete exercises at home. The e-learning system has been useful to collect all AToM³ ar-

tifacts within one web-based system. Moreover, it provides a comprehensive overview of the deadlines that have been met by each individual student. However, teaching assistants have selectively relaxed the firm deadlines by allowing e-mail submissions too. This was desirable when students encountered unexpected problems that were due only to unpredictable behavior of the AToM³ tool.

Secondly, at the end of the semester the students must pass an oral exam with a written preparation. During this exam, the remaining course objectives are tested, namely “*The student should be able to express the intended functionality of a system from different viewpoints in different formalisms (Petri-Nets, Graph Grammars) and ensure particular properties (boundedness, consistency, ...) of such models.*” and “*The student should ... acquire an understanding of the benefits and limitations of the 4-layer meta model architecture.*” The former is tested with an exercise in Petri-Net modeling and verification while the latter is tested with a discussion on two papers selected from the list read during the paper sessions.

3 Course Artifacts

This section presents some of the models, metamodels and transformations that need to be developed in the new course.

3.1 Petri-Net Editor

Figure 2 displays a simplified version of one of the models used to convince students that AToM³ can be used to build powerful editors. The Petri-Net models the dining philosophers problem with six philosophers. The Petri-Net editor is developed in the second session of the AToM³ labs. For improving the readability of this paper, the token capacities and edge weights have been omitted. The model can be animated using the graph grammar that is developed in sessions three, four and five.

In Figure 2, each philosopher is represented by a pair of places that are positioned on a diameter of the circle that represents the table. The outer place from such a pair holds a token when the philosopher under consideration is thinking. Therefore, in Figure 2, all six philosophers are thinking. These philosophers are numbered clockwise and with number 1 for the two places on the top of the figure. From the twelve places representing the philosophers, six places hold a token. These six places represent the three spoons and three forks. Since they hold a token, all silverware lies on the table.

After executing some graph transformation rules (that are selected automatically from the grammar), the Petri-Net model from Figure 2 has evolved into that of Figure 3. In the latter (version of the) model, philosophers 1 and 5 are

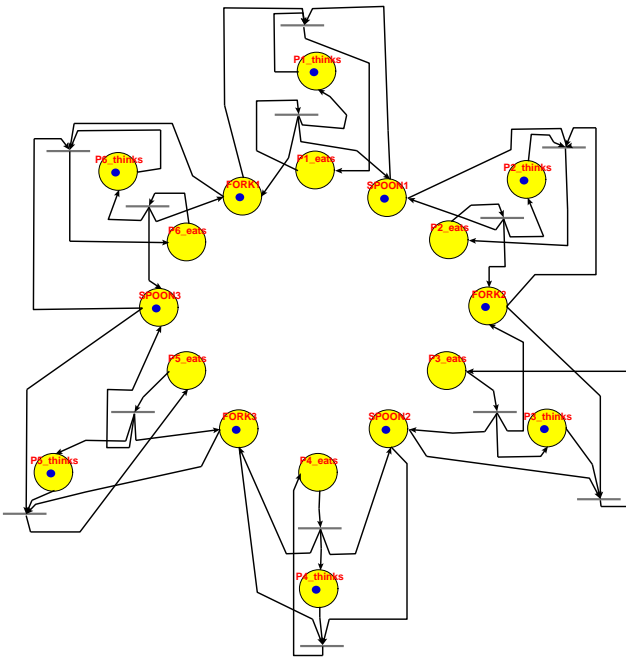


Figure 2. Petri-Net model: Dining Philosophers.

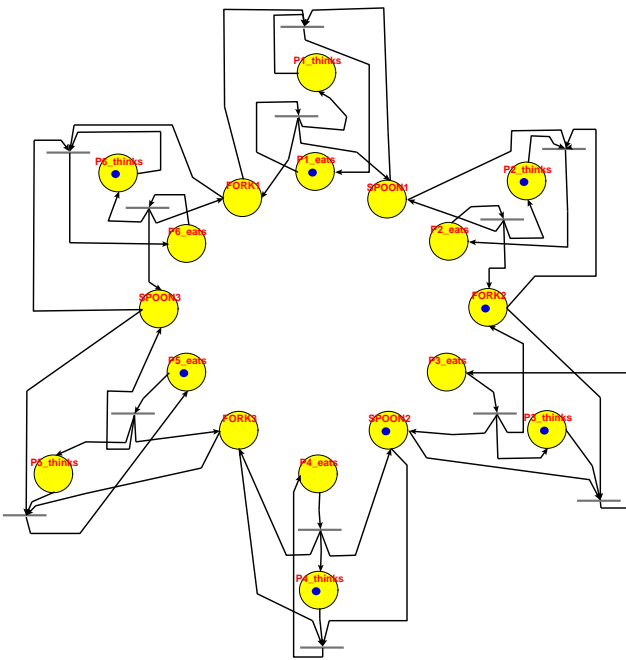


Figure 3. Model during graph grammar execution.

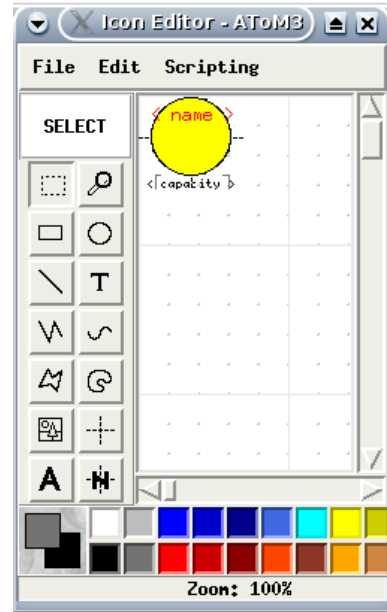


Figure 4. Editor for modeling the concrete syntax of visual language elements.

eating while the others are still thinking. Philosopher 3 can start eating since spoon 2 and fork 2 are still on the table. However, philosophers 2, 4 and 6 have to wait until their neighbours (philosophers 1 and 5) have finished eating and returned their silverware to the table.

The Petri-Nets language can be represented by a straightforward metamodel containing a *Place* class and a *Transition* class, related by an association for input places and an association for output places. Figure 4 shows the ATOM³ editor for modeling the concrete syntax of the *Place* class. This example teaches students how to use the basic constructs for representing individual language elements and for representing relations between these elements. In addition, students are confronted with the limitations of visual modeling. More specifically, the representation of tokens within a place is realized by some pragmatic programming at the level of the Python code that ATOM³ generates from the metamodel and the concrete syntax definition.

The transition semantics for Petri-Nets can be modeled by a variety of graph grammars. However, some essential rules occur in almost any solution. Figure 5 presents a snapshot of the active ATOM³ windows when such a rule is being edited. At the center of the screenshot, the graph transformation rule for incrementing output places is shown. The “order” field at the top of the dialog defines the priority of this rule and is set to 4. Rules with a lower order have a higher precedence. In this example, such rules make sure that the input places of the transition under consideration

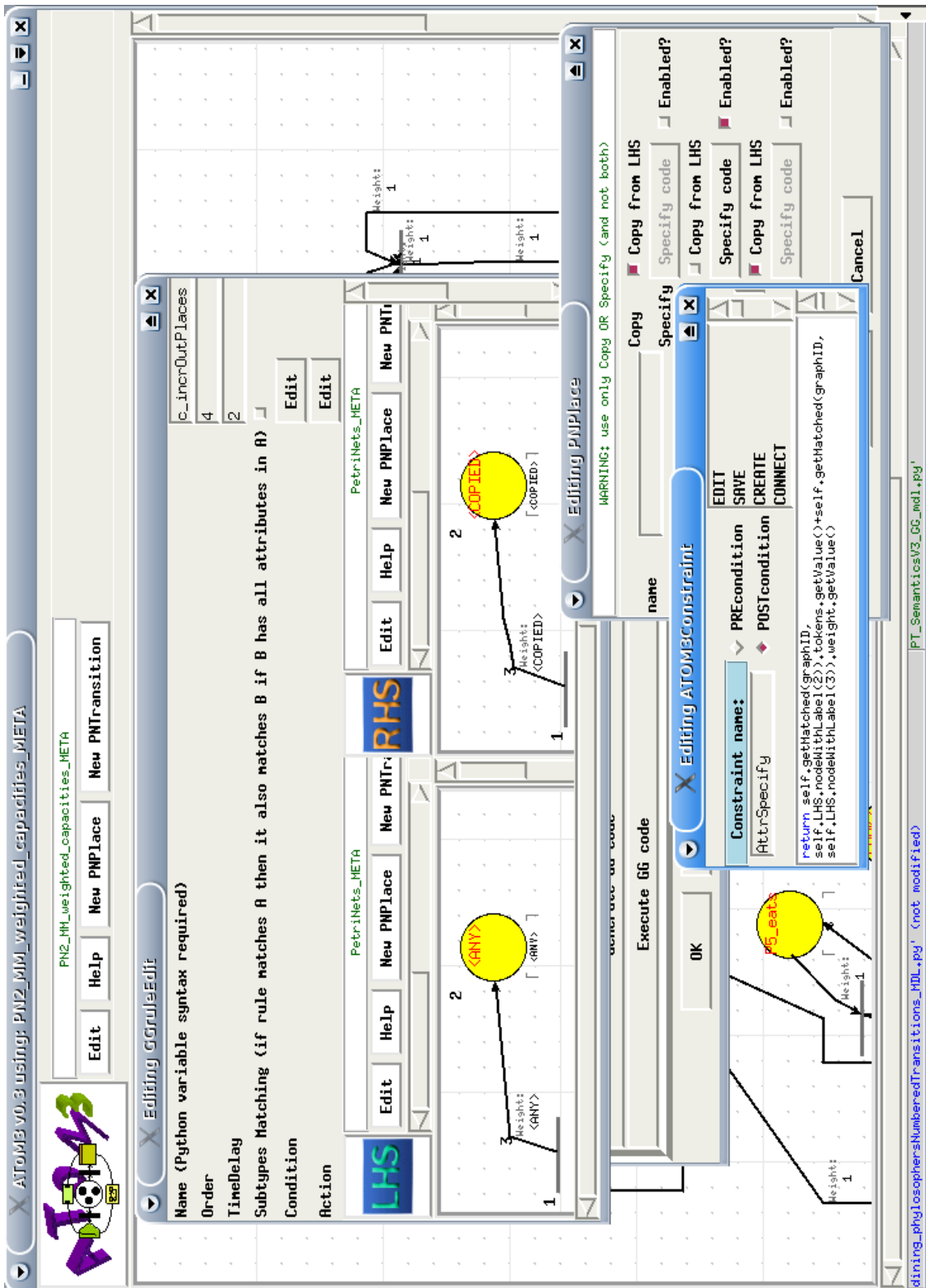


Figure 5. Screenshot of the ATOM³ dialogs for editing graph grammar rules.

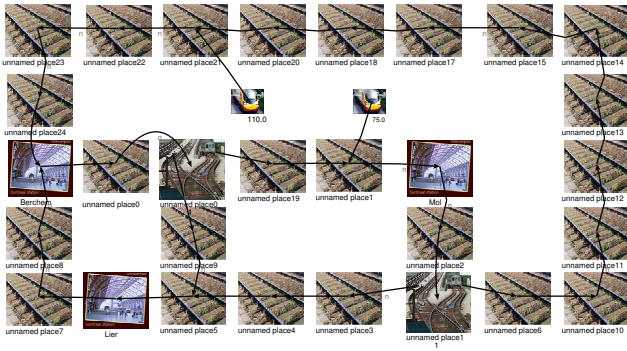


Figure 6. Sample model in the Railroad language.

hold enough tokens to enable the transition. Moreover, it is ensured that not more than one transition is enabled at a time. On the other hand, rules with a lower precedence make sure that the transition under consideration is disabled and that other rules can be enabled again.

Unlike most of today’s generic model transformation engines [12, 5], AToM³ supports the specification of rewrite rules in *concrete* syntax. The left-hand side of the rule in Figure 5 specifies that any transition, called 1, holding an edge to an output place, called 2, should be matched. The screenshot does not show that these nodes are constrained further by additional clauses from the “condition” field. The right-hand side of this rule copies the weight of the edge and the name and capacity of the output place. In contrast, the value of the “tokens” attribute from the output place is specified in Python code. The dialog at the bottom center of the screenshot shows that this value is defined by the old value of the “tokens” attribute plus the value of the weight of the incoming link. Within this dialog, the values of the Pre- and Post-condition widgets have no meaning for this example.

3.2 Railroad Editor

The Railroad language requires a somewhat more complex metamodel. In summary, it is desirable to use inheritance between classes such as *Track*, *Station* and *Fork*. Tracks and stations can be connected to exactly one next track while a fork holds a *left* and a *right* outgoing track. A Fork has an explicit property that denotes whether incoming trains will be switched to its left or right track. This property can be altered manually. Figure 6 illustrates what kind of systems can be modeled in the Railroad language. Note that the edges between the tracks, stations and forks define the path along which trains can travel the railroad.

Again, this language is supported by a graph grammar for simulation. As Figure 7 shows, this example brings students in contact with polymorphic matching: the rule for

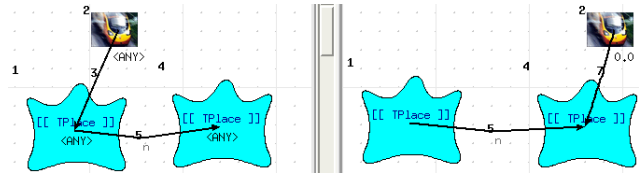


Figure 7. Polymorphic rule for moving trains.

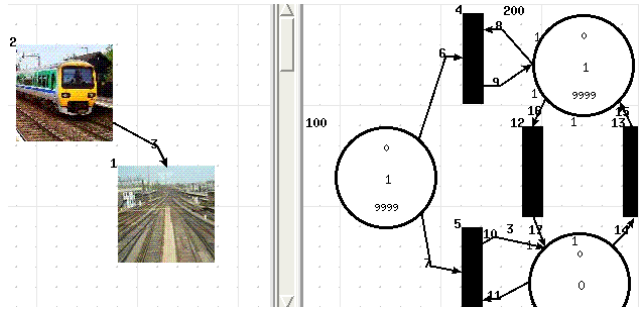


Figure 8. Translation rule for a left-going fork to a Petri-Net pattern.

moving a train from one track to the next one states that as soon as a train is located on a *Place* (a *Track* or a *Station*), it can be moved to the next track. This is realized by representing the connection with label 3 only on the left-hand side of the transformation rule and creating a new connection with label 7 in the right-hand side. All other properties are preserved.

3.3 From Railroad Models to Petri-Nets

The final course artifact is a graph grammar for translating Railroad models to Petri-Net models. Figure 8 displays a fragment from a student’s solution. The rule matches *Fork* elements that hold a train and that are configured for moving trains to the left (this property would be visible when opening a property editor for node 1). For such *Fork* elements, the translation rule generates a *Place* element that holds one token (representing the train) and two subpatterns holding a transition, two opposite connections and a place. These patterns are used to encode the behavior of the Train switch: the upper place holds a token when trains should be moved to the left and the lower place holds a token when trains need to be moved to the right. Therefore, this transformation rule generates a token in the upper place and no tokens in the lower place. A transition connects the place from the upper subpattern with the lower place and vice versa. These transitions can be used to switch the *Fork* pattern in the Petri-Net from left to right.

Other rules generate links between the transitions representing the next-track relationship. These rules need to

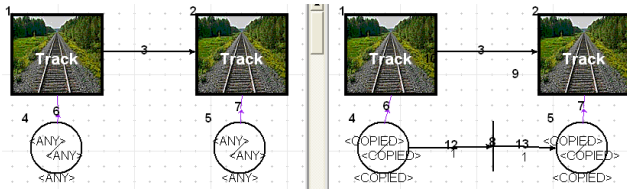


Figure 9. Navigation across traceability links.

employ some kind of traceability information. AToM³ provides “generic links” for connecting model elements from different languages. However, since the use of this built-in traceability mechanism resulted in obscure error messages, students had to design an ad-hoc solution in Python code. On the one hand, it is unfortunate that not all students have thus not been able to “model” traceability links within the rewrite rules. On the other hand, the exercise illustrated that in the final lab session, most students master the tool “internals” sufficiently to implement an alternative solution creatively.

One student did succeed to apply the built-in traceability feature of AToM³. Figure 9 for example displays his rule that properly applies generic links to generate a transition between places that correspond to two connected tracks. The left-hand side of the rule contains the two tracks, labeled 1 and 2, that are connected in the input model. The edges labeled 6 and 7 elegantly model that these tracks need to be mapped to two places, labeled 4 and 5, before this rule becomes applicable. The right-hand side of the rule specifies that between these two places a new transition, labeled 8, should be generated.

In summary, it can be stated that the AToM³ exercises have brought students in contact with the most essential techniques in model-driven development.

4 Lessons Learned

This section summarizes the lessons that were learned from designing and teaching the first edition of the course. In general, student satisfaction (as measured during a formal questionnaire) exceeds the expectations of the instructors. Nevertheless, several points for improvement are considered for future editions. The first part of this section presents the best practices that were applied in the design of the course. The second part discusses the outlook for the second edition. The section concludes by discussing the short- and long-term industrial applicability of the course’s learning outcomes.

4.1 Best Practices

The instructors of the course have paid attention to the following didactical aspects:

Feasibility Study. A key factor that contributed to the success of the new course was the use of a sufficiently tested toolsuite. All solutions to the lab sessions were prepared before planning the complete course structure. This ensured that as long as students used the evaluated AToM³ release, teaching assistants could guide them around known bugs and limitations. It should be noted that AToM³ was not just the first and best tool selected for this course. Instead, during the course preparation phase, other tools were actually found unsuitable or incomplete for use by undergraduate students.

Expert Supervision. As a constructive guide to the adoption of formal methods, Antony Hall complemented his objections against misbeliefs about formal methods with a list of hints about training in formal methods [10]. Hall essentially confirms that after theoretical training in general discrete mathematics and a specific formal language, practical workshops are indispensable. Moreover, supervision by at least one tutor is said to be essential. Hall also reports about productivity problems in the context of non-user-friendly tools. By allocating two teaching assistants during the course of all AToM³ sessions, the time that the students under supervision spent on figuring out the AToM³ user interface specifics was significantly smaller than the time the teaching assistants had spent on that issue.

Work Incrementally. By building upon the results of previous lab sessions, students have been able to construct a realistic toolsuite within a very limited time frame. The result has convinced students that the techniques they have learned can be applied on realistic problems. By defining intermediate milestones for the construction of the graph grammar for animating Petri-Nets, students have been able to demonstrate their progress at the end of each lab. This approach resulted in student satisfaction weeks before the delivery of their complete toolsuite.

Start Small. Because the Petri-Net toolsuite was used as a partial evaluation for the exam, teaching assistants had to ensure that students produced a solution as individually as possible. This was achieved by starting the AToM³ sessions with the development of the self-contained Robustness Diagram editor. Students should be – *and have been* – able to generalize their expertise from this small exercise to the larger ones.

Illustrate Applicability. Formal methods can help future system users understand what kind of system will be built by modeling functionality before it is realized. However, that requires that the formal model is made accessible to such a user [10]. Instead of claiming the applicability of Petri-Nets by means of natural language explanations, the course relied on the customer-oriented Railroad language. The translation between the Railroad and the Petri-Net languages provides concrete evidence that formal methods can be economically integrated into the requirements elicitation process.

Examples First. Another driving force for letting students define the Railroad language is that this exercise motivates the need for inheritance in the language for meta-modeling. From such examples, it becomes straightforward to motivate why the Meta Object Facility (MOF), OMG's standard language for metamodeling, resembles class diagrams.

Problems First. Before referring to the papers on more powerful model transformation languages, the lab exercises expose the limitations of the simplistic AToM³ approach. More specifically, the Petri-Net animation grammar illustrates that the execution order of graph grammar rules does not necessarily correspond to the order in which these rules are defined. By experiencing this problem in the lab sessions first, students understand why in Story Driven Modeling [7], graph transformation rules are embedded in an activity diagram.

4.2 Planned Improvements

The following list summarizes the future work on the course:

Provide Tool Feedback. AToM³ was originally developed for research purposes. Applying it in a classroom context revealed several bugs and usability issues. Constructive feedback is collected from students and teaching assistants. This feedback may influence future releases of the tool.

Consider Other Tools. Since the tool landscape is rapidly evolving, new tools may provide the metamodeling, concrete syntax definition and model transformation features required to construct an equivalent of the Petri-Net toolsuite. Therefore, the maturity of alternative tools needs to be evaluated frequently.

Provide Reusable Integration Components. The motivation of the students can be increased by providing so-called *technical projectors*: for example, one could develop a Python component that generates a file

compliant with a popular Petri-Net analysis tool from instances of an AToM³ metamodel for Petri-Nets and vice versa. Such a component would close the remaining gap between the Petri-Net verification part of the course and the model-driven engineering part, without reinventing the wheel.

Extend Railroad Case Study. A feedback loop should be developed from the analysis of a Petri-Net that is generated from a Railroad model by means of the graph grammar. By illustrating students how such analysis results should be interpreted, the case study would practically show how model-driven engineering can help improving the safety of software.

Prepare Follow-Up Courses. The curriculum at the University of Antwerp also includes two new graduate courses related to Model-Driven Engineering. These courses need to be designed such that students can apply their AToM³ expertise in tools based on actual MDA standards such as QVT [15] and industrial-strength frameworks such as the Eclipse Graphical Modeling Framework (GMF [8]).

4.3 Industrial Relevance

This section briefly discusses the short- and long-term applicability of this course in an industrial context.

In 2006, the computer science curriculum at the University of Antwerp has been completely redesigned. One novelty is that graduate students need to choose between an industrial, educational or research profile. Since the course being discussed in this paper takes place in the third year of the bachelor program, it should still fit into the three profiles. As Section 2.2 illustrates, this leads to subtle trade-offs. Coming back to the design of the course objectives, the learning outcomes that were initially planned would be directly applicable in industry: many organizations are struggling with the complexity of today's middleware and use code generators such as AndroMDA to tackle this issue.

The revisited learning outcomes may seem less applicable in the short term. However, the acquired insights are directly applicable for companies that need technical advice in the selection of an MDA tool. By gaining practical experience with model transformation, graduates should be able to look beyond marketing labels and investigate a tool's adaptability. Moreover, companies from e.g. the automotive industry demand expertise in the customization and integration of custom visual modeling tools [1, 18].

5 Conclusions

This paper presented a new course that combines traditional lectures and exercises on formal modeling and verification with a series of practice-oriented lab sessions on emerging model-driven engineering techniques. The course is focused on the definition and integration of languages rather than on the use of the languages as such. By incrementally developing an integrated case study within controlled lab sessions, students encounter problems in practice before reading and debating related papers. The permanent evaluation of the course indicated continuous student satisfaction, despite the theoretical nature of the background material and frequent failures of the supportive tool.

Acknowledgements

This work has been sponsored by the Belgian national fund for scientific research (FWO) under grant “Formal Support for the Transformation of Software Models”. The authors wish to thank Denis Dubé for providing technical support during the preparation of the AToM³ lab sessions.

References

- [1] C. Bock. Visuelle domänenspezifische sprachen - der schlüssel zur modellgetriebenen entwicklung von menschenmaschine-schnittstellen? <http://software-families.org/>, 10 2006.
- [2] M. Bohlen et al. AndroMDA Model Driven Architecture framework. <http://galaxy.andromda.org/docs-3.2/>, 2007.
- [3] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.
- [4] E. Commission. ECTS - european credit transfer and accumulation system. <http://ec.europa.eu/education/programmes/socrates/ects/>, 6 2007.
- [5] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
- [6] J. de Lara and H. Vangheluwe. Using atom³ as a meta-case tool. In *International Conference on Enterprise Information Systems*, pages 642–649, 2002.
- [7] I. Diethelm, L. Geiger, and A. Zündorf. Systematic story driven modeling, a case study (Paderborn shuttle system). In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE'04*, Edinburgh, Scotland, 2004.
- [8] Eclipse. Graphical Modeling Framework. <http://www.eclipse.org/gmf/>, 2007.
- [9] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as Eclipse plug-ins. In *ASE'05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, New York, NY, USA, 2005. ACM Press.
- [10] A. Hall. Seven myths of formal methods. *IEEE Software*, 07(5):11–19, 1990.
- [11] A. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Object Technology Series. Addison – Wesley, 2003.
- [12] T. Mens and P. Van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation (GraMoT), Satellite Event of GPCE*, 2005.
- [13] M. Minas. Generating visual editors based on Fujaba/MOFLON and DiaMeta. In H. Giese and B. Westfechtel, editors, *Proc. Fujaba Days. Technical Report tr-ri-06-275 - University of Paderborn*, pages 35–42, Bayreuth, Germany, 2006.
- [14] O. Muliawan, H. Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). <http://motmot.sourceforge.net/>, 2005.
- [15] Object Management Group. *MOF QVT Final Adopted Specification – ptc/05-11-01*, 2005. <http://www.omg.org/docs/ptc/05-11-01.pdf>.
- [16] D. Rosenberg and K. Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [17] Universiteit Antwerpen. Blackboard academic suite. <http://blackboard.ua.ac.be/>, 6 2007.
- [18] J. Weiland. Variantenkonfiguration von modellbasierter embedded automotive software. <http://software-families.org/>, 10 2006.