Electronic Communications of the EASST
Volume 34 (2010)

Proceedings of the 6th Educators' Symposium: Software Modeling in Education at MODELS 2010 (EduSymp 2010)

Teaching Model Driven Language Handling

Terje Gjøsæter, Andreas Prinz

10 pages

Guest Editors: Peter J. Clarke, Martina Seidl
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Teaching Model Driven Language Handling

Terje Gjøsæter[1], **Andreas Prinz**[2]

[1] terje.gjosater@uia.no
[2] andreas.prinz@uia.no
http://www.uia.no/
Faculty of Engineering and Science
University of Agder, Grimstad, Norway

Abstract: Many universities teach computer language handling by mainly focussing on compiler theory, although MDD (model-driven development) and meta-modelling are increasingly important in the software industry as well as in computer science. In this article, we share some experiences from teaching a course in computer language handling where the focus is on MDD principles. We discuss the choice of tools and technologies used in demonstrations and exercises, and also give a brief glimpse of a prototype for a simple meta-model-based language handling tool that is currently being designed and considered for future use in teaching.

Keywords: MDD, meta-modelling, language specification, teaching

1 Introduction

MDD (model-driven development) and meta-modelling is increasingly important in the software industry as well as in computer science. However, many universities still teach language handling with the main focus on compiler theory. For example, in the Norwegian universities, there is a strong emphasis on compiler theory and little or no focus on meta-modelling in most of the available computer language handling courses [GP11].

The focus among language designers is shifting towards creating small domain specific languages (DSLs) [KT08]. These languages may have a graphical or textual presentation (concrete syntax), and they are often based on existing languages and may be preprocessed / embedded / transformed into other languages for execution, instead of being compiled with a traditional compiler.

MDD may have some advantages when it comes to defining these types of languages. An important aspect of MDD is to provide the language designer with support for rapid development and automatic prototyping of language support tools, and allow for working on a high level of abstraction. This approach allows the language designer to focus on the language being developed, while still being able to use the definition for generating tools such as editors, validators and code generators.

It may therefore be beneficial to modify university courses in computer language handling to focus not only on compiler development, but also on meta-model-based language design and definition.

The main purpose of this article, is to share experiences from teaching meta-model-based language description, and to discuss which tools and technologies are suitable for covering the

different aspects of a language definition when teaching computer language handling.

The article is based on literature study, language specifications, and the authors' own experiences with tools, language descriptions as well as teaching of both compiler theory and meta-modelling.

The rest of the article is organised as follows: Section 2 gives an overview of the different aspects of a meta-model-based language specification and outlines a course teaching these principles. Section 3 discusses issues related to choice of tools and technologies for use in teaching the different language aspects. Finally, we summarise our findings in Section 4.

2 Teaching Meta-model-based Language Handling

2.1 Overview

When a course in compiler theory was modified to also cover meta-modelling, it became clear that we needed to get a common understanding between the two paradigms; which parts of a compiler description correspond to which parts of a meta-model-based language description. In [NPT06], a language definition is said to consist of the following aspects: *Structure*, *Constraints*, *Presentation* and *Behaviour* (see Figure 1).

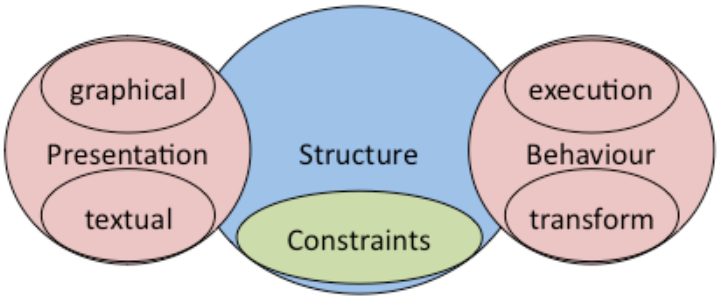

Figure 1: Aspects of a computer language description

Structure defines the constructs of a language and how they are related.

Constraints bring additional constraints on the structure of the language, beyond what is feasible to express in the structure itself.

Presentation defines how instances of the language are represented. This can be the definition of a graphical or textual concrete language syntax.

Behaviour explains the semantics of the language. This can be a transformation into another language (denotational or translational semantics), or it defines the execution of language instances (operational semantics). Another type of semantics is axiomatic semantics, that

gives meaning to phrases of a language by describing the logical axioms that apply to them.

These aspects are not always as strictly separated as they seem in the illustration; constraints are shown as overlapping with structure, since constraints interact closely with the structure-related technologies in building up (and restricting) the structure of the language. However, constraints can also be used for defining restrictions for presentation as well as behaviour.

The structure is the core of the language; it contains the concepts that should be part of the language, and the relations between them. A meta-model-based approach to language design facilitates a focus on the structure. Starting from a well-defined language structure, it is convenient to define one or more textual and/or graphical presentations for the language, as well as to define code generation into executable target languages such as Java.

Meta-models define the structure and constraints of a language. For a complete language definition, it is also necessary to define the presentation and behaviour, and relate these definitions to the meta-model, as explained in [Kle07].

2.2 A Computer Language Handling Course

As described in more detail in [GP11], we have investigated how a course that primarily focused on compiler theory could be updated to include meta-model-based approaches to language definition, and a special focus on determining the optimal abstraction level for each language aspect. Based on this, we have defined the following course outline that covers both meta-model-based as well as compiler-based approach to language definition:

Level:	MSc.
Prerequisites:	Object oriented programming, UML modelling.
Credits:	5 ECTS
Literature:	Aho, Lam, Sethi, Ullman: Compilers (2nd ed.)[ALSU07]; Clark, Sammut, Willans et. al.: Applied Metamodeling (2nd ed.) [CSW08]
Form:	8 parts; each part with lectures, practical and theoretical exercises, and an obligatory hand-in.

Part 1 - Introduction: Compilers, languages, language aspects, grammars, NFA and DFA automata, T-diagrams.

Part 2 - Structure: Models, meta-models, MDA, meta-models, abstract syntax, attribute grammars.

Part 3 - Constraints: Semantic analysis, type systems, static and dynamic checks, type safety, logical constraints.

Part 4 - Textual presentation: Syntax analysis, top-down and bottom-up parsing, lexical analysis, mapping, symbol tables, error handling, textual presentation for meta-models.

Part 5 - Graphical presentation: Graphical languages, graph grammars, graphical presentation for meta-models.

Part 6 - Transformation behaviour: Transformation, code generation, intermediate code, optimisation, handling of generated code, model-to-model and model-to-text transformations.

Part 7 - Execution behaviour: Semantics, interpreters, runtime environments, storage allocation, activation records, parameter passing, dynamic binding, operational semantics for meta-model-based languages.

Part 8 - Summary: Repetition of the most important topics of the course.

The course has been implemented at the University of Agder in the spring term of 2010. In a related project course, the students have a choice of different projects building on this course.

After running the language handling course, the following experiences were gathered:

- It is good to use a running example where aspects are added to complete a simple example language. It is also beneficial to cover all language aspects within one platform. However, students can easily be demotivated by immature tools.

- We should not try to cover too many different tools in the practical exercises, but rather concentrate on the most important ones and give the students more time to try them out for themselves by modifying and extending provided examples.

- The understanding should be strengthened by giving different perspectives on the same issues in a lecture covering both compiler theory and meta-modelling. However, the connection between the two paradigms were sometimes difficult for the students to see.

- The choice of tools and technologies for use in the course, is of big significance. We will therefore discuss some of the available tools and technologies and their suitability for use in teaching in Section3.

2.3 Finding the Correct Abstraction Level

An important part of this course concerns finding a good abstraction level in order to facilitate code generation from models. In this respect, tools for language description are used as an example. However, it is a challenge to find tools and technologies that work on a high abstraction level for each language aspect. If the abstraction level is too low, there are too many seemingly irrelevant details, that create complications and complexities that will make it more difficult for the students to get started with the tools. On the other hand, if the abstraction level is too high, it may not be possible to generate working tools from the language specification. For *Structure* and *Textual Presentation*, there are tools that operate on a suitable level of abstraction, while it is more difficult to find good abstractions for the other language aspects. We will cover some of the available tools and technologies and their suitability for use in teaching in the following section.

3 Choice of Tools and Technologies for Teaching

3.1 Overview

Immature or overly complex tools and technologies can demotivate students and in some cases even make them avoid meta-model-based projects. A former Master student has described experiences from implementing a DSL in both Eclipse with suitable plugins, and in Visual Studio with DSL tools, and concluded that Visual Studio is good on integration, documentation and ease of use, while Eclipse allows the developer to operate on a more suitable high level of abstraction and has a good selection of plug-ins to extend its functionality. However, both platforms have weaknesses when it comes to stability and user-friendliness [IGP08].

3.2 Choice of Platform

We prefer free multi-platform tools and technologies to lower cost and to enable students to install the software on their home computers. We also wish to have a collection of tools that can co-exist in one platform, such as for example Eclipse. We have also seen that the stability and user-friendliness has increased for Eclipse over the last couple of years, so we have ended up using that as our preferred platform, and testing various plug-ins to cover the different aspects of a language specification. In the following, we will give a brief overview of our choices of tools and technologies for each of the language aspects listed in Section 2.

3.3 Tools and Technologies for Teaching Structure

The *structure* of a language specifies what the instances of the language are; it identifies the meaningful components of each language construct [Set96] and relates them to each other.

There are several ways to express structure; grammars, meta-models, database schema descriptions, RDF schemata, and XML schemata are all examples of different ways to express structure. There are different standards and recommendations for defining meta-models with different complexity and expressiveness. The most famous dialects are MOF 1.x[OMG02], EMF/Ecore[SBPM08], and CMOF[OMG03]. EMF/Ecore or EMOF is a simplified version of MOF 1.x; among other simplifications, it removes associations and replaces them with pairs of class references. It seems reasonable to start a course in meta-model-based language design with an introduction to structure definition, using for example Eclipse with EMF/Ecore (preferably with a graphical Ecore editor) for demonstrating relevant examples.

3.4 Tools and Technologies for Teaching Constraints

Constraints on a language can put limitations on the structure of a well-formed instance of the language. This aspect of a language definition mostly concerns logical rules or constraints on the structure that are difficult to express directly in the structure itself. Neither meta-models nor grammars provide all the expressiveness that is needed to define the set of wanted language instances. The constraints could for example be first-order logical constraints or multiplicity constraints for elements of the structure [PST07].

In meta-modelling, the most common way to express constraints is the Object Constraint Language, OCL, which has the expressiveness of predicate logic, in a programming-language-like syntax. A lecture on constraints can be illustrated by creating and adding OCL constraints to a sample meta-model, and using an OCL toolkit such as MDT OCL or the EMF Validation Framework.

3.5 Tools and Technologies for Teaching Presentation

The *presentation* of a language describes the possible forms of a statement of the language. In the case of a textual language, it describes what words are allowed to use in the language, what words have special meaning and are reserved, and what words are possible to use for variable names. It may also describe what sequence the elements of the language may occur in; the syntactic features of the language. This is expressed in a grammar for textual languages.

We have two main approaches to creating tools for handling presentation of a language;

Parsers that have to support a one-way connection from the presentation to the corresponding structure.

Editors that have to support a two-way connection between the presentation and the corresponding structure, providing feedback from the syntax analysis in form of syntax highlighting, error messages, code completion suggestions etc.

In addition to the presentation definition, an explicit or implicit mapping is needed to connect it to the structure.

One popular framework for defining graphical notations is GMF [GMF08]. It features a language to define graphical notations, and generates Eclipse and GEF-based [GEF08] editors from these definitions. It allows for defining possible diagram elements and tool palettes, as well as explicit mapping to structure.

Frameworks for textual notations can be divided into tools like XText [EFH+08], which provides editors based on language definitions consisting of grammars, and frameworks like TCS[JBK06], TEF [Sch08] and EMFText [HJK+09], which combine meta-models and grammars. An advantage of EMFText, is that it can generate a HUTN-based (Human-Usable Textual Notation) parser and editor from an Ecore meta-model, that can be used as a starting point for developing a textual notation.

If a running meta-model-based example is used, it may be fruitful to show the students how an EMF-based example structure (with constraints) can be extended with both graphical and textual presentations, using editor generation frameworks like for example GMF for graphical editor generation and EMFText for textual editor generation.

3.6 Tools and Technologies for Teaching Behaviour

The *behaviour* of a language describes what is the actual meaning of a statement of the language.

Two main types of formal ways of defining semantics are called operational and denotational semantics [Set96]:

Denotational semantics in the strict sense, is a mapping of a source expression to an input-output function working on some mathematical entities. If we wish to include model transformations and language-to-language translations in our behaviour descriptions, we can include them in this category by applying a more broad definition of denotational semantics; namely a transformation of each phrase of the language into a phrase in some other language, often a mathematical formalism. To execute or interpret the behaviour of a statement, semantics for the target language is then needed. A denotational semantics describes an "abstract" compiler.

Operational semantics describes the execution of the language as a sequence of computational steps. You will then need to know the semantics of the interpreter. Operational semantics may be described by state transitions for an abstract machine. In [PST07], it is described how semantics for SDL are handled by Abstract State Machines (ASM). With operational semantics, a runtime environment is needed. An operational semantics describes an "abstract" interpreter.

A third type of semantics, *Axiomatic* semantics, gives meaning to phrases of a language by describing the logical axioms that apply to them. Experience shows that axiomatic semantics are extremely complex and rarely used for computer languages. For this paper we only focus on denotational and operational semantics.

We have noted that it may be challenging to teach this language aspect since most of the tools available for supporting the theory of this aspect are relatively immature and/or hard to use, particularly for execution behaviour.

Transformation languages like QVT [OMG05] or ATL [BDJ+03] can be used to create example transformations on the structure of the running EMF-based example, and for the latter, JET [JET04], Acceleo [MJL08] or XPand [EFH+08] can be used to generate textual code. The Eclipse plugin EProvide, provides support for developing visual debuggers and interpreters based on operational semantics defined in ASM, QVT/Relations, Java, Prolog or Scheme.

For illustrating the theory in this lecture, we may want to apply Model-to-Model transformations using QVT or ATL, and Model-to-Text with for example JET or XPand. We may also demonstrate operational semantics with ASM-based semantics in EProvide.

3.7 An Alternative Platform

Based on experiences from teaching, we have concluded that it may be useful to develop a very simple meta-model-based language definition platform, that attempts to remove some of the complexity of the more popular existing tools, in order to better allow the students to grasp the basic principles of meta-modelling. It should let the student operate on a suitable level of abstraction on each relevant language aspect, and facilitate making and modifying small example languages. In order to achieve this, we have started designing and prototyping a new platform named LanguageLab. It is planned to be a complete environment for experiments with modular language specification, particularly intended for use in teaching.

The following use cases will be supported:

- Edit/select language elements and put them together into a complete language (with tools)

- Create a language specification in a modular way.
- Based on an existing/predefined structure, the user can modify it to fit his needs, and a new language (with tools) is created.
- Combining pre-defined modules to create a language (with tools) supporting some required features (supported/implemented by those modules)
- A language module can cover a complete, or parts of a, language aspect. Parts can be for example: inheritance, loops, composite objects.
- A language module uses and provides interfaces for other modules that the language developer can use.
- Create a structure and connect it to a predefined execution model.
- Create an interface for a language module in order to promote it to a meta-language module that can be used for defining other language modules.

A language will consist of one or more modules that have structured elements that can be instantiated into a runtime model representing the language instance, via an interface. Each module supplies *create*, *get*, and *set* operations for each *Type* element that is accessible from the interface. If the created runtime model is intended to be used as a language module, it is possible to create an interface from the runtime model by promoting runtime model type instances to types via an optional operation in the module.

A simple prototype has been developed based on Eclipse/EMF. It supports some basic functionality, allowing us to test it by creating a state machine runtime model from a simple state machine language module (only the structure aspect is supported in the initial prototype), as shown in Figure 2.

4 Conclusions

One of the main challenges of teaching meta-model-based language handling is finding tools that are simple, on a high abstraction level, and that work well together with other tools for other language aspects. It is our impression that some of the perceived complexity of meta-modelling comes from complex tools and technologies, rather than from the principles behind them.

It is possible to build a series of lectures in meta-model-based computer language handling supported by running examples based on Eclipse/EMF and other Eclipse-based plug-ins and frameworks, to cover all aspects of a language definition.

However, we think that it may also be interesting and fruitful to develop and introduce a very simple meta-model-based language definition platform, that attempts to remove some of the complexity of the more popular existing tools, in order to better allow the student to grasp the basic principles of meta-modelling. It should let the student operate on a suitable level of abstraction on each relevant language aspect, and facilitate making and modifying small example languages.

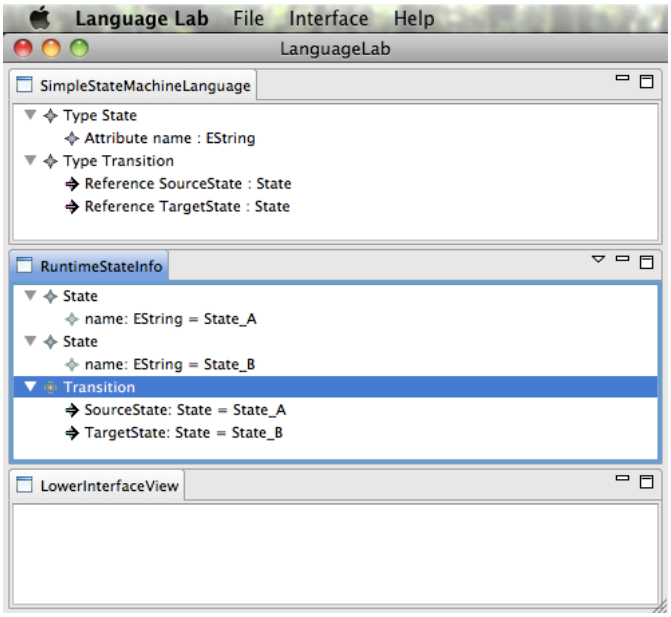

Figure 2: LanguageLab prototype

Bibliography

[ALSU07] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools, 2nd ed.* Addison-Wesley, 2007.

[BDJ+03] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, J. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *OOPSLA 2003 Workshop, Anaheim, California.* 2003.

[CSW08] T. Clark, P. Sammut, J. Willans. *Applied Metamodeling – A Foundation for Language Driven Development. Second Edition.* Ceteva, 2008.

[EFH+08] S. Efftinge, P. Friese, A. Haase, D. Hübner, C. Kadura, B. Kolb, J. Köhnlein, D. Moroff, K. Thoms, M. Völter, P. Schönbach, M. Eysholdt. OpenArchitectureWare User Guide. see also http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/index.html, 2008.

[GEF08] GEF developers. GEF documentation. see also http://www.eclipse.org/gef/reference/documentation.php, 2008.

[GMF08] GMF developers. Eclipse Graphical Modeling Framework. 2008. See also http://www.eclipse.org/gmf.

[GP11] T. Gjøsæter, A. Prinz. Teaching Computer Language Handling - From Compiler Theory to Meta-modelling. In *GTTSE 2009*. LNCS 6491, pp. 446–460. Springer, 2011.

[HJK+09] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Model Driven Architecture - Foundations and Applications*. Lecture Notes in Computer Science 5562/2009, pp. 114–129. 2009.

[IGP08] I. F. Isfeldt, T. Gjøsæter, A. Prinz. Meta-model-based implementation of Sudoku: Eclipse vs. Visual Studio. In *Norsk informatikkonferanse : NIK 2008*. Pp. 51–62. 2008.

[JBK06] F. Jouault, J. Bézivin, I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *GPCE'06: Proceedings of the fifth international conference on Generative programming and Component Engineering*. Pp. 249–254. 2006.

[JET04] JET developers. JET Tutorial part 1. See also http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, 2004.

[Kle07] *A Language is More than a Metamodel*. 2007. Available at http://megaplanet.org/atem2007/ATEM2007-18.pdf.

[KT08] S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling*. Wiley-Interscience, 2008.

[MJL08] J. Musset, É. Juliot, S. Lacrampe. Acceleo User Guide. See also http://acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf, 2.6 edition, 2008.

[NPT06] J. P. Nytun, A. Prinz, M. S. Tveit. Automatic Generation of Modelling Tools. In Rensink and Warmer (eds.), *ECMDA-FA*. Lecture Notes in Computer Science 4066, pp. 268–283. Springer, 2006.

[OMG02] OMG Editor. Meta Object Facility (MOF) Specification. Technical report, Object Management Group, 2002.

[OMG03] OMG Editor. Revised Submission to OMG RFP ad/2003-04-07: Meta Object Facility (MOF) 2.0 Core Proposal. Technical report, Object Management Group, April 2003.

[OMG05] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted Specification ptc/05-11-01. OMG document, Object Management Group, 2005.

[PST07] A. Prinz, M. Scheidgen, M. S. Tveit. A Model-based Standard for SDL. In *SDL 2007: Design for Dependable Systems*. Lecture Notes in Computer Science 4745, pp. 1–18. Springer Berlin / Heidelberg, 2007.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley Professional, second edition, 2008.

[Sch08] M. Scheidgen. Textual Editing Framework. see also http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/documentation.html, 2008.

[Set96] R. Sethi. *Programming Languages Concepts and Constructs*. Addison-Wesley, 1996.