Teaching Computer Language Handling - From Compiler Theory to Meta-modelling

Terje Gjøsæter and Andreas Prinz

Faculty of Engineering and Science
University of Agder
Serviceboks 509, NO-4898 Grimstad, Norway
{terje.gjosater,andreas.prinz}@uia.no

Abstract. Most universities teach computer language handling by mainly focussing on compiler theory, although MDA (model-driven architecture) and meta-modelling are increasingly important in the software industry as well as in computer science. In this article, we investigate how traditional compiler theory compares to meta-modelling with regard to formally defining the different aspects of a language, and how we can expand the focus in computer language handling courses to also include meta-model-based approaches. We give an outline of a computer language handling course that covers both paradigms, and share some experiences from running a course based on this outline at the University of Agder.

1 Introduction

Although MDA (model-driven architecture) and meta-modelling is increasingly important in the software industry as well as in computer science, many universities still teach language handling with the main focus on compiler theory. For example, in the Norwegian universities, we have found that there is a strong emphasis on compiler theory (CT) and little or no focus on meta-modelling (MM) in most available computer language handling courses (see Table 1).

Compiler theory has traditionally had its strength in defining optimised compilers for large textual general purpose languages. On the other hand, the focus among language designers is shifting towards creating small domain specific languages (DSLs) [1]. 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.

MDA may have some advantages when it comes to defining these types of languages. An important aspect of MDA 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

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 446–460, 2011.

Table 1. Courses available at Norwegian universities related to computer language handling. The information is collected from course catalogues and course descriptions.

University	Course Name	ECTS	MM	CT	Notes
Bachelor level courses:					
Norwegian U. of Sci. and Tech.	TDT4165 Programming languages	7,5		x	Languages and language implementation
U. of Oslo	INF3110 Programming languages	10		x	Language description
Master level courses:					
U. of Agder	IKT415-C System development with generative programming	5	x	x	Recently revised to also include MM
Norwegian U. of Sci. and Tech.	TDT4205 Compilers	7,5		x	Compiler construction
U. of Oslo	INF5110 Compiler techniques	10	x	x	CT is main focus but MM is mentioned
U. of Bergen	INF225 Introduction to program translation	10		x	Last held in 2005, compiler focussed
No courses currently available:					
U. of Stavanger	N/A				No courses available
U. of Tromsø	N/A				No courses available

generators. Meta-model-based tools are typically based on these principles, but there are also grammar-based tools available that take a similar approach, such as LISA [2].

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 compare a compiler-theory-based approach with a meta-model-based approach to master level courses in computer language handling, and to examine how that type of courses can be modified from a focus on traditional compiler theory to also cover meta-model-based approaches, tools and technologies. We wish to emphasise that the goal of this paper is not to come up with clear-cut statements about which is better of grammar-based and meta-model-based language definition technologies, but rather to find out which technologies are adequate and suitable for which aspects of a language definition, and how both approaches can be included 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 and 3, are introductory sections enumerating the main elements we want to cover in courses in compiler theory and meta-modelling, respectively. Each of these language elements, or language *aspects*, are handled in the following sections; *structure / abstract syntax* in Section 4, *constraints / static semantics* in Section 5, *presentation / concrete*

syntax in Section 6 and *behaviour / dynamic semantics* in Section 7. For each of the language aspects described in sections 4 to 7, we have subsections covering the following topics:

- A general introduction to this aspect.
- Some important issues related to teaching this aspect from the perspective of compiler theory.
- Some important issues related to teaching this aspect from the perspective of meta-modelling, including a selection of meta-model-based tools and technologies that can be used to illustrate the theory of this aspect.
- A comparison of the two approaches from the perspective of teaching this aspect.

In Section 8, we propose an outline of a unified computer language handling course, covering meta-modelling as well as compiler theory. Finally, we summarise our findings in Section 9.

2 A Compiler Theory Curriculum

From Figure 1, we see the basic flow of the main elements of a compiler, and this can also serve as the sequence of main topics for a series of lectures in compiler-construction-based language handling.

Concrete syntax including scanner and parser parts of the compiler, and symbol table generation.
Static semantics including type checking and logical constraints.
Abstract syntax including intermediate code generation and building abstract syntax trees.

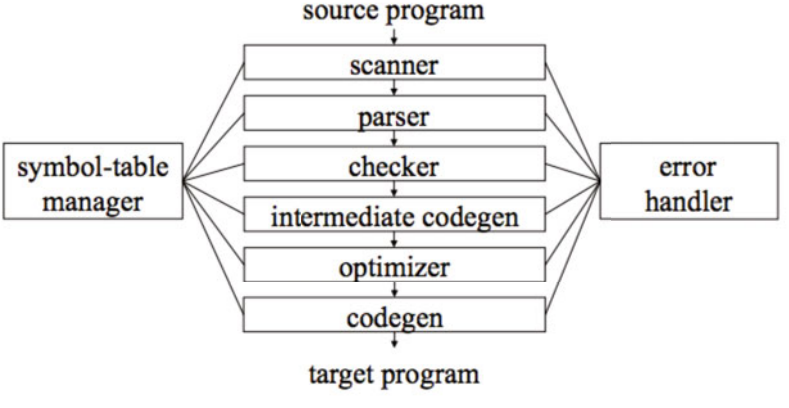

Fig. 1. Elements of a compiler

Translational semantics including optimisation, code generation and error handling.
Execution semantics including run time environments.

In traditional compiler technology, languages are defined by a concrete syntax, an abstract syntax and semantics. Concrete syntax can be precisely defined in BNF/EBNF, and compiler-compiler and parser generator tools like lex and yacc may be used to generate the parser. EBNF can also be used for defining the abstract syntax, however in practice abstract syntax is often automatically derived from the concrete syntax. Although there are well established methods for specifying the formal semantics for a language, in practice, semantics is often not formally defined but developed in an ad-hoc fashion [3].

3 Metamodelling - A Curriculum Based on Aspects of a Programming Language

In [4], a language definition is said to consist of the following aspects: *Structure*, *Constraints*, *Presentation* and *Behaviour* (see Figure 2).

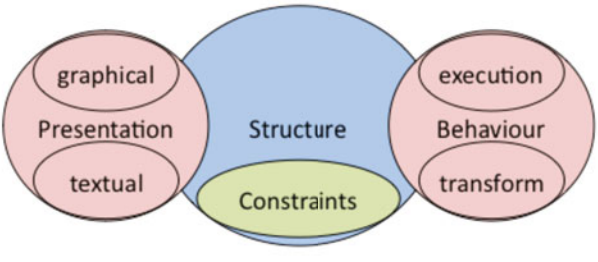

Fig. 2. Aspects of a computer language description

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 presentations 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. Traditional grammar-based compiler tools tend to force the focus to the presentation of the language rather than its structure. On the other hand, 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. It is feasible to build a series of lectures in computer language handling on a running example using Eclipse/EMF-based [5] plug-ins and frameworks, to illustrate all aspects of a meta-model-based language definition.

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 [3].

Because of this difference in main focus between traditional compiler technology and meta-modelling, it also seems reasonable to let this be reflected in the teaching of these topics. When teaching compiler theory, it is common to start with parsing and grammars, and then later move into abstract syntax and finally semantics and code generation. However, when teaching meta-model-based language design, it is essential to start with teaching how to create a well-formed abstract structure, instead of initially focussing on the presentation of the language. Based on the Structure lecture, should follow lectures on Constraints, Textual and Graphical Presentation, and finally lectures on Transformation (Model-to-Model and Model-to-Text) and Execution.

4 Structure / Abstract Syntax

4.1 Definition

The *structure* of a language specifies what the instances of the language are; it identifies the meaningful components of each language construct [6] and relates them to each other. Based on the language structure definition, a language instance can normally be represented as a tree or a graph. To describe the structure of a computer language therefore means to describe graph structures: what types of nodes exist, and how they can be connected.

There are different levels of expressiveness used in different contexts; grammars, meta-models, database schema descriptions, RDF schemata, and XML schemata are all examples of different ways to express structure.

4.2 Topics and Issues for the Compiler Theory Lecture

Compiler technology commonly uses context-free grammars to define structure. Abstract syntax is in most cases quite similar to the concrete syntax, with some

redundancy removed. Most popular computer languages are grammar-based and do not have a separately described abstract syntax definition, but rely on the concrete syntax.

Grammars can be used to define abstract syntax trees as data structures for language instances [7]. An extended form of context-free grammars are attribute grammars; they define attributed abstract syntax trees. Attributes can help to realise the other aspects of the language: constraints, presentation and behaviour. Attributes can also function as additional connections that turn the abstract syntax tree into a real graph.

The lecture on abstract syntax should include an introduction to grammars and common language structures including regular languages, automata, context free languages, parse trees, abstract syntax trees and attribute grammars.

4.3 Topics and Issues for the Meta-modelling Lecture

While simple grammars define a tree-structure, meta-models are capable of defining a graph. Meta-modelling uses UML's structure modelling constructs to model the structure of languages. A meta-model only defines an abstract structure for a language, because it just describes what the language concepts are and not how their are written or drawn. A meta-model introduces classifiers like classes and associations, for all the constructs in a language. Associations are used to define how instances of these classes, i.e. instances of language constructs, relate to each other.

Meta-modelling allows to modularise, reuse, and combine whole languages or single language constructs. To achieve this, meta-modelling uses object-oriented UML notions like packages and imports, class inheritance, and feature refinement. Object-orientation does not only help with the meta-model design, but also for the design of other definitions and tools based on the meta-model. The graphical nature of meta-models can also facilitate understanding of the structure compared to a textual presentation of structure as is commonly used with grammars.

There are different standards and recommendations for meta-modelling with different complexity and expressiveness. The most famous dialects are MOF 1.x [8], EMF/Ecore [5], and CMOF [9]. The simplicity of EMF/Ecore and EMOF makes it easy to align it to the Java programming language. This fact and EMFs tight integration into Eclipse [10], make it today's most popular meta-modelling language.

Around meta-modelling (especially EMF and Eclipse), many tools and frameworks have been created to easily describe and execute tasks like: persistent language instances in data-bases, validation of language instances, model transformation, execution of language instances, and providing different forms of model editors. Meta-modelling fuelled the vision of creating domain specific languages including comprehensive tools with little resources. Meta-modelling and repositories are the bases for many existing domain specific language and UML case tools.

One weak point with meta-models, is that it is still not well understood what criteria to use to evaluate the quality of a meta-model, and what properties are important and desirable.

4.4 Comparison Related to Teaching

We note that structure can be handled fine with both approaches, but while the structure is usually the starting point when defining a language with meta-model-based technologies, there is less emphasis on this aspect in traditional compiler theory. Therefore, 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. It should also be noted that the simple tree structures generated from simple grammars are easier to understand for students then the more complex graphs typically formed by meta-models and attribute grammars.

5 Constraints / Static Semantics

5.1 Definition

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 [11].

There is an overlap between the structure and constraint aspects of a language. Some language features may obviously belong to one of them, but many features could belong to either of them, depending on choice or on the expressiveness of the technology used to define the structure.

5.2 Topics and Issues for the Compiler Theory Lecture

What we want to express here are logical rules (static semantic conditions) related to elements of the language structure. Often, these constraints are expressed in code, and in some cases in a logic language. These logical rules may be attached to attributes in an attribute grammar. This lecture should also include an introduction to type systems and type checking.

5.3 Topics and Issues for the Meta-modelling Lecture

While meta-models are constructive definitions of what objects a language instance can consist of, constraints allow to narrow down the possible instances of a meta-model class. A meta-model constraint is thereby always written in the

context of a class, and only constrains the set of possible instances (objects) of this class. A constraint forms a logical expression. It takes an instance of the context class as input and evaluates to a boolean value, assessing the instance as either a valid, or an invalid instance. Only the models that exclusively consist of valid objects are valid language instances.

In meta-modelling, the most common technology for expressing constraints is the Object Constraint Language, OCL. OCL is designed to present the expressiveness of predicate logic, in a programming language like syntax. Related tools allow to check whole models or single objects, based on the constraints associated with the model's meta-model classes. Language tools based on meta-models, usually do not check a model within a separate tool, but are integrated into model editors. Model editors check single objects and can display invalid objects to the user.

5.4 Comparison Related to Teaching

We see that constraints can be handled fine with both approaches. There is often more emphasis on explicitly defined constraints in meta-model-based development. Teaching constraints will fit naturally as an extension to lectures about structure, and can be illustrated by creating and adding logical expressions to an example grammar and OCL constraints to an example meta-model.

6 Presentation / Concrete Syntax

6.1 Definition

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.

Similarly, in a graphical language, the presentation will express what different symbols are used in the language, and how they can be connected and modified to form a meaningful unit in the language. The *presentation of graphical languages* can be defined in two ways:

The "constructive" way is generator-based, using graph grammars.
The "direct" way may describe a model for the graph.

In addition to defining the graph structure, we may wish to define attributes such as location, shape, and colour of the different elements in the graph.

Describing language structures separately from the language's notation(s), allows us to define multiple notations for the same language, and allows for arbitrary kinds of notations, graphical (diagrams and other variants) as well as textual.

There are two major ways to *connect the presentation to the structure* of a language;

The "constructive" way is done by defining a transformation between presentation and structure.

The "abstract" way is based on pattern matching, showing how elements of the presentation are connected to the structure elements.

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, executable output text in the form of machine code or byte code can also be considered a presentation of a language instance. Code generators have to support a one-way connection from the structure to a presentation of the code to be generated.

6.2 Topics and Issues for the Compiler Theory Lecture

The presentation of a language is in traditional compiler theory called *concrete syntax*. Context free grammars for the concrete syntax of a programming language are often written in BNF (Backus–Naur form) or EBNF (Extended BNF) notation, and most popular parser generators use grammars based on an (E)BNF-like syntax.

This lecture should include a basic introduction to different grammar types such as LL, LR, SLR, LALR; and also parse tables, canonical sets, first-follow sets, conflicts (shift-reduce and reduce-reduce), conflict resolution, and mapping between concrete and abstract syntax. For graphical concrete syntax, a brief introduction to graph grammars should be included. Symbol table management and error handling may also fit into this lecture.

6.3 Topics and Issues for the Meta-modelling Lecture

Meta-models describe language structures with classes and associations resulting in language instances that are graphs, rather than the tree structures normally generated by grammars. Therefore, a meta-model is a suitable basis for defining graphical notations.

In general, notations are described in a separate formalism. Textual notation for example can be described with context-free grammars, graphical notations can be described in their own meta-model for shapes and connections. A third model then defines a mapping between the meta-model elements and the notation definition's elements.

A formalism to define a certain kind of notation consist of a notation definition language and a mapping definition language. Existing formalism for notations definition are usually embedded in frameworks and tools that realise

them. Existing formalisms for graphical notations and textual notations allow to automatically create graphical editors and feature-rich text editors, including error annotations (for validations), code-completion, name-resolution, syntax highlighting, etc.

Graph grammars have also been suggested for use in creating modelling tools. See for example [12] for more information on graph grammars.

Frameworks for textual notations can be divided into tools like XText [13], which actually provides editors solely based on language definitions consisting of grammars, and frameworks like TCS [14], TEF [15] and EMFText [16], which combine meta-models and grammars. XText allows to define a language syntax and implicitly a language structure based on a grammar-like definition. XText generates a meta-model and a textual editor for this meta-model from this definition. The editor continuously generates a meta-model instance, by parsing the text entered by the user. The other frameworks allow to provide a grammar and grammar-to-meta-model mapping based on already existing meta-models. They generate editors that allow creation of meta-model instances, by parsing the user text. These frameworks use techniques similar to those of attribute grammars to handle non-containment model structures and provide automatic support for resolving named references based on these techniques.

One well-known framework for graphical notations is GMF [17]. It features a language to define graphical notations, including different shapes, shape containment, connections, and labels for different elements. GMF allows to define simple mappings between meta-model elements and the elements of a graphical notation. GMF generates Eclipse and GEF-based [18] editors from these definitions. This is fully functional for simple language notations, and can be enriched by manually altering the generated code.

6.4 Comparison Related to Teaching

While textual presentation may be a natural starting point in compiler theory based teaching, it is more suitable to let lectures on presentation build on a foundation of basic structure. In this part, the students should get practice in defining grammars for simple languages as well as deriving languages from grammars. 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.

7 Behaviour / Dynamic Semantics

7.1 Definition

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 [6]:

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 [11], 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.

7.2 Topics and Issues for the Compiler Theory Lecture

Semantics has traditionally been an area that is much less formalised than the structure or abstract syntax of a language. In most cases, the semantics has been described in plain English, or by reference implementation of a compiler or interpreter for the language. However, for attribute grammars, it is quite common to attach more formal semantic rules to the attributes.

7.3 Topics and Issues for the Meta-modelling Lecture

Similar to grammars, in many cases, the semantics has been described in plain English, or by reference implementation.

While the focus in traditional compiler theory teaching often is on code generation, it is natural in a course focussing on meta-model-based technologies to cover model-to-model transformations as well as model-to-text transformations. For the former, transformation languages like QVT or ATL can be used to create example transformations on the structure of the running EMF-based example, and for the latter, JET [19], Acceleo [20] or XPand [13] can be used to generate textual code.

There is plenty of academic work that suggests the definition of operational semantics based on a meta-model. These approaches create state-transitions systems to describe behaviour along Plotkin's classical operational semantics [21]. These systems are based on meta-models as a definition for the set of states, and depending on the approach use transformations based on graph transformations

[22], or some form of action languages, like UML Activities [23], or ASM [11] to define possible state transitions. The Eclipse plugin EProvide [24], provides support for developing visual debuggers and interpreters based on operational semantics defined in ASM, QVT/Relations, Java, Prolog or Scheme.

7.4 Comparison Related to Teaching

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. Model-to-model transformations tend to be better supported by meta-model-based tools and technologies. Model-to-text transformation is adequately supported by both approaches. On the other hand, execution is not well supported in any of the two approaches.

For illustrating the theory in this lecture, we may want to give the structure of our running example Model-to-Model transformations using QVT or ATL, and Model-to-Text with for example JET or XPand. It may also be useful to demonstrate operational semantics with ASM-based semantics in EProvide.

8 A Computer Language Handling Course Outline

From the ideas developed in the previous sections, we have defined the following course outline:

Level: MSc.
Prerequisites: Object oriented programming, UML modelling.
Credits: 5 ECTS.
Literature: Aho, Lam, Sethi, Ullman: Compilers (2nd ed.)[25]; Clark, Sammut, Willans et. al.: Applied Metamodeling (2nd ed.) [26].
Form: 7 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, EMF/Ecore, abstract syntax, attribute grammars.
Part 3 - Constraints: Semantic analysis, type systems, static and dynamic checks, type safety, logical constraints, OCL.
Part 4 - Textual presentation: Syntax analysis, top-down and bottom-up parsing, lexical analysis, mapping, symbol tables, error handling, TEF, EMF-Text.
Part 5 - Graphical presentation: Graphical languages, graph grammars, GMF.
Part 6 - Transformation behaviour: Transformation, code generation, intermediate code, optimisation, handling of generated code, JET, QVT.

Part 7 - Execution behaviour: Semantics, interpreters, runtime environments, storage allocation, activation records, parameter passing, dynamic binding, ASM, EProvide.
Part 8 - Summary: Repetition of the most important topics of the course.

In the related project course, the students have a choice of different projects building on this course.

8.1 Experiences

The course has been implemented at the University of Agder in the spring term of 2010. After running the 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.

9 Conclusions

When it comes to teaching of computer language handling, we conclude that although traditional compiler theory should still play an important role, there is also a need for a stronger focus on meta-model based technologies. Although the two paradigms are emphasising different aspects of language definition, we have shown that it is still possible to cover all important language aspects relatively well with either paradigm.

It is possible to build a series of lectures in computer language handling where both compiler-theory-based and meta-model-based approaches are covered. The meta-model-based approach can be illustrated by running examples based on Eclipse/EMF and other Eclipse-based plug-ins and frameworks, to cover all aspects of a language definition.

We have found that it is essential to emphasise the connections and similarities between compiler-theory-based and meta-model-based approaches to language handling, and to avoid the least mature tools. It is also important to ensure that the students has a common software platform to lower the risks of unexpected bugs and problems.

References

1. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. Wiley Interscience, Hoboken (2008)
2. Henriques, P.R., Pereira, M.J., Mernik, M., Lenič, M.: Automatic generation of language-based tools. In: Second Workshop on Language Descriptions, Tools and Applications, LDTA 2002. Electronic Notes in Theoretical Computer Science, vol. 65, pp. 77–96. Elsevier Science Publishers, Amsterdam (2002)
3. Kleppe, A.: A language is more than a metamodel. In: ATEM 2007 Workshop (2007), http://megaplanet.org/atem2007/ATEM2007-18.pdf
4. Nytun, J.P., Prinz, A., Tveit, M.S.: Automatic generation of modelling tools. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 268–283. Springer, Heidelberg (2006)
5. Griffin, C.: Using EMF. Technical report, IBM: Eclipse Corner Article (2003), http://www.eclipse.org/articles/Article-UsingEMF/using-emf.html
6. Sethi, R.: Programming Languages Concepts and Constructs. Addison-Wesley, Reading (1996)
7. van Eijk, P., Belinfante, A., Eertink, H., Alblas, H.: The Term Processor Generator Kimwitu. CTIT Technical report 96-49, University of Twente (1996), http://fmt.cs.utwente.nl/kimwitu/
8. OMG Editor: Meta Object Facility (MOF) Specification. Technical report, Object Management Group (2002), http://www.omg.org/docs/formal/02-04-03.pdf
9. OMG Editor: Revised Submission to OMG RFP ad/2003-04-07: Meta Object Facility (MOF) 2.0 Core Proposal. Technical report, Object Management Group (2003), http://www.omg.org/docs/formal/06-01-01.pdf
10. d'Anjou, J., Fairbrother, S., Kehn, D., Kellermann, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley, Reading (2004)
11. Prinz, A., Scheidgen, M., Tveit, M.S.: A Model-based Standard for SDL. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 1–18. Springer, Heidelberg (2007)
12. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and tools, vol. 2. World Scientific, Singapore (1999)
13. Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., Schönbach, P., Eysholdt, M.: OpenArchitectureWare User Guide (2008), http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/index.html
14. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proceedings of the Fifth International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 249–254 (2006)
15. Scheidgen, M.: Textual Editing Framework (2008), http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/documentation.html
16. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
17. GMF developers: Eclipse Graphical Modeling Framework (2008), http://www.eclipse.org/gmf

18. GEF developers: GEF documentation (2008), http://www.eclipse.org/gef/reference/documentation.php
19. JET developers: JET Tutorial part 1 (2004), http://www.eclipse.org/articles/ArticleJET/jet_tutorial1.html
20. Musset, J., Juliot, É., Lacrampe, S.: Acceleo User Guide (2008), http://acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf.2.6edn
21. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, University of Aarhus (1981)
22. Grunske, L., Geiger, L., Zündorf, A., Eetvelde, V., Van Gorp Niels, P., Varró, D.: Using Graph Transformation for Practical Model Driven Software Engineering. In: Model Driven Software Engineering, pp. 91–118. Springer, Heidelberg (2005)
23. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007)
24. Sadilek, D.A., Wachsmuth, G.: Prototyping visual interpreters and debuggers for domain-specific modelling languages. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 63–78. Springer, Heidelberg (2008)
25. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Reading (2007)
26. Clark, T., Sammut, P., Willans, J.: Applied Metamodeling – A Foundation for Language Driven Development, 2nd edn. Ceteva (2008)