Realia et naturalia

DISSERTATIONES
MATHEMATICAE
UNIVERSITATIS
TARTUENSIS
87

MARGUS FREUDENTHAL

Simpl: A toolkit for domain-specific language development in enterprise information systems

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
87

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
87

MARGUS FREUDENTHAL

Simpl: A toolkit for domain-specific language development in enterprise information systems

UNIVERSITY OF TARTU
PRESS
1632

Institute of Computer Science, Faculty of Mathematics and Computer Science, University of Tartu, Estonia

Dissertation is accepted for the commencement of the degree of Doctor of Philosophy (PhD) on September 2nd, 2013 by the Council of the Institute of Computer Science, University of Tartu.

Supervisor:

Prof. PhD.	Marlon Gerardo Dumas Menjivar University of Tartu Tartu, Estonia

Opponents:

Prof. PhD.	Marjan Mernik University of Maribor Maribor, Slovenia
Prof. PhD.	Kuldar Taveter Tallinn University of Technology Tallinn, Estonia

The public defense will take place on October 25th, 2013 at 16:00 in Liivi 2-403.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu.

ISSN 1024-4212
ISBN 978-9949-32-391-3 (print)
ISBN 978-9949-32-392-0 (pdf)

University of Tartu Press
www.tyk.ee

Contents

LIST OF PUBLICATIONS

1. Freudenthal, M.: Domain-Specific Languages in a Customs Information System. IEEE Software 27(2), 65–71 (Mar 2010).

2. Freudenthal, M.: Using DSLs for developing enterprise systems. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications. pp. 11:1–11:7. LDTA '10, ACM, New York, NY, USA (2010).

3. Freudenthal, M., Pugal, D.: Simpl: a Toolkit for Rapid DSL Implementation. In: Proceedings of the 12th Symposium on Programming Languages and Software Tools - SPLST'11. Institute of Cybernetics (October 2011).
   This paper describes the Simpl DSL toolkit and two experiments that were performed to evaluate the toolkit. The author contributed description of Simpl and one of the evaluations (the controlled usability study).

4. Freudenthal, M.: Implementing Oberon0 Language with Simpl DSL Tool. Tech. Rep. T-4-18, Cybernetica AS (2013), available online at `http://research.cyber.ee/`.

ABSTRACT

Domain specific languages (DSLs) are languages designed with the specific purpose of developing or configuring part of a software system using concepts that are close to those of the system's application domain. Documented benefits of DSLs include increased development productivity, flexibility and maintainability, as well as separation of business and technical aspects allowing in some cases non-technical stakeholders to closely partake in the software development process. DSLs however comes at a potentially non-negligible cost, that of creating and maintaining DSL implementations. These costs can be reduced by means of specialized tools that support the creation of parsers, analyzers, code generators, pretty-printers, and other functions associated with a DSL.

This thesis deals with the problem of enabling cost-effective DSL-based development in the context of Enterprise Information Systems (EIS). EISs are generally built using application frameworks and middleware. Accordingly, it must be possible to package the DSL implementation as a module that can be called from either the build system or from the enterprise system itself. Additionally, the DSL tool should be accessible to enterprise system developers with little or no expertise in development of programming languages and supporting tools, such as Integrated Development Environments (IDEs).

The central contribution of the thesis is Simpl, a DSL toolkit designed to fulfill a number of DSL tool requirements identified in the context of EISs. Simpl builds up on top of existing tools and programming languages, and introduces the following features: a grammar description language that supports the generation of both the parser and the data types for representing abstract syntax trees; support for lexer states that add context-sensitivity to lexer in a controlled manner; a pretty-printing library; an IDE framework; and an integration layer that combines all components into a single whole and minimizes the need for boilerplate code.

Simpl has been evaluated in a multi-pronged manner via a controlled experiment and by comparing DSL implementations in Simpl against implementations of the same DSLs using alternative tools. The evaluation demonstrates that Simpl is suitable for EIS development and offers usability that is comparable and in some cases superior to other DSL tools.

CHAPTER 1

INTRODUCTION

1.1 Problem Area

Van Deursen et al. define a Domain-specific language (DSL) as "a programming language or an executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [13]. The key characteristic of DSLs according to this definition is their *focused* expressive power. In other words, DSLs trade off broad applicability to gain power of expression in particular domain. Classical examples of DSLs are Unix Makefiles (build scripts), regular expressions (specifying text patterns), HTML (describing text layout) GraphViz (describing graphs).

The benefits of using DSLs have been widely studied in the literature. They include development productivity, flexibility, maintainability (including program comprehension), and separation of business and technical aspects. The following list provides some examples of research related to benefits of DSLs.

- Batory et al. [3] provide an experience report on the use of DSLs in conjunction with Software Product Line Engineering, to implement a command-and-control simulator. The case study demonstrated the benefits of DSLs in terms of added development productivity and flexibility with respect to an implementation in Java.
- Van Deursen et al. [88] report an experience in using a DSL in the financial engineering domain with an emphasis on the added maintainability.

- Chandra et al. [7] and Thibault et al. [87] discuss case studies where DSLs were used to implement video-device drivers and distributed cache coherence protocols respectively, demonstrating the benefits of separation of concerns between technical aspects and business logic.

- Kosar et al. [52, 51] report on series of experiments that compared program understanding of DSLs with GPL libraries. They measured three aspects of program understanding: learning the notation and meaning of programs, perceiving the meaning of a program or a language construct, and evolving and extending existing programs. The studies showed DSLs to be superior to application libraries in all the three aspects.

- Kärnä et al. [42] compared GPL programming with use of a graphical DSL tool (Metacase DSM tool) and found that the use of DSL improved developer productivity by an order of magnitude.

The main reason behind the above observed benefits is the high level of abstraction of DSL programs and closeness of mapping between DSL concepts and domain concepts. However, the benefits of using a DSL must be balanced with the costs (see [88, 60, 42] for discussion of costs associated with using DSLs). One source of costs is the creating and maintaining the DSL and its implementation. Another source of costs is integrating the part of the application developed using the DSL with the other parts developed by means of other general-purpose programming languages. The decision to use a DSL depends on the domain and the size of the application. Sprinkle et al. [82] describe characteristics of problems that can be effectively solved with DSLs. In this respect tool support for DSL creation plays a critical role as it can reduce the implementation cost and therefore make the DSL-oriented approach usable in a wider range of projects.

This thesis investigates practical issues connected to using DSLs and DSL tools for developing enterprise information systems. In order to be more concrete, we must first define what is an "enterprise information system" (EIS). Finding the authoritative definition for this term is quite difficult (if not impossible), but there seems to be a rough consensus that the EIS is a system for integrating and coordinating business processes of a (usually large) organization. From the technical point of view, the enterprise systems can be characterized by the following properties.

- They are usually implemented using object-oriented programming languages such as Java, C#, a 4GL, or a database language

(PL/SQL). In other words, the tools used do not represent cutting edge of the programming language research and are typically not considered suitable for DSL-oriented programming. The enterprise systems also make extensive use of frameworks (such as JEE and .NET) and middleware (application servers, enterprise service buses etc.).

- They may consist of a set of interconnected modules that are built upon a common architecture or on top of a packaged enterprise system such as an Enterprise Resource Planning (ERP) system.

- They tend to be shallow but wide. Although the enterprise systems are technically quite complex, involving persistence, transactions, messaging, remote procedure calls, and other architectural mechanisms, this complexity is implemented in libraries, frameworks and middleware. Thus, the application code does not contain significant amount of technical details and instead is focused on implementing the concepts, rules and processes of the organization.

The frameworks and middleware that are used to build the EIS usually contain different graphical or XML-based DSLs for configuring the components. These DSLs are often called horizontal or technical DSLs. Additionally, there are vertical or business DSLs that are concentrated on encoding business logic for a specific business domain such as accounting, banking, or medical domains. Taking advantage of the vertical DSLs is usually technically more complex because there may not exist a suitable DSL (with suitable implementation) for this particular application, and the EIS developer herself must create the necessary DSLs. This thesis focuses on the second, more business-oriented type of DSL where the DSL and the DSL-supporting components are developed together with the rest of the EIS.

Language-oriented programming is a general method that can be realized by means of DSLs embedded in a general-purpose programming language (internal DSL). In this way, the various DSLs can be integrated by means of the host programming language. However, mainstream general-purpose programming languages commonly used for developing enterprise systems do not offer features needed for creating high-quality internal DSLs (e.g. unobtrusive syntax, ability to define new control structures). In addition, internal DSLs are often difficult to use for non-programmers because the details of the host language often interfere with the DSL (for example, in the case of error messages). Therefore, this thesis focuses on external DSLs that are implemented using especially crafted parser and that can offer syntax

and semantics that does not depend on the host language. When building an external DSL, one can use either textual or graphical syntax. Both have different strengths and weaknesses and choosing between them can be a matter of taste and availability of tools, competence, etc. However, the tools for developing textual DSLs have different characteristics compared to those for developing graphical DSLs. This makes detailed technical analysis and comparison of tools a difficult task because a wide range of variations must be taken into account. In order to be able to delve into the technical details of DSL tooling, this thesis focuses on one type of DSLs, namely textual DSLs. Textual DSLs are better researched, are more popular, have better tool support, and, in the author's opinion, scale better for expressing complex relationships and behavior.

If one adopts the DSL-based development process, the costs of developing and maintaining a DSL must be low. If the cost of DSL implementation does not offset the savings gained by using the DSL, it is more economical to code the business logic in a general-purpose programming language. The DSL development can be made more efficient by taking advantage of good tool support. There exists a reasonable body of tools to assist in various aspects of creating DSLs. Examples of these tools are parser generators, code generators, transformation systems, and IDE generators. The level of sophistication varies widely, starting from simple parser generators or template engines and ending with tools that provide ability to describe type systems in a declarative manner, or to merge grammars of two (sub) languages to produce a (non-ambiguous) grammar for the composite language. However, many of otherwise functional and useful tools are not suitable for all aspects of enterprise software development because they are heavyweight and/or they impose restrictions on the development process of the overall EIS. In EIS development one cannot assume that the overall system is built around a DSL or a DSL tool. Instead, there is a need facilitate the integration between the part of the system implemented using a DSL with the part of the system developed using other means. Therefore, the DSL implementation must conform to the requirements set by the EIS and the surrounding development process instead of the other way around. In summary, it is important that a DSL tool and DSL implementations produced with the tool are lightweight and that they place no additional requirements on either the technical infrastructure and to the development process.

1.2 Problem Statement

Our overall goal is to make DSL-oriented software development in the EIS sector economically feasible. To achieve this, we aim to provide tool support for implementing DSLs in the enterprise setting. The ideal tool would be

- lightweight so that it does not interfere with the overall system;
- embeddable so that the DSL implementations and DSL programs can be integrated with the rest of the system; and
- easy to use so that both the DSL tool and DSL implementations are accessible to enterprise developers.

Introducing an additional tool to an existing project or team can often encounter difficulties. The transition can be made easier if the added tool does not introduce additional concerns into the development. The concerns can be both technical (such as need to change architecture of the overall system) or related to organization (such as changes to development process, need to retrain the developers). It is important that both the DSL tool and the DSL programs are able to seamlessly integrate with the other parts of the system and interact with the frameworks and libraries used. Ideally, the tool itself can be embedded into the bigger enterprise system. Finally, in order to be useful, the DSL tool must be usable by the enterprise developers who are generally not specialists in programming language tools. The tool should be easy to learn and allow convenient creation of both simple, one-off code generators and complex, full-featured DSLs.

1.3 Contributions

A prevalent theme of this work is that the use of DSLs in enterprise software development imposes specific requirements for the tools. We argue that most of the popular DSL tools do not satisfy these requirements. On the other hand, we demonstrate that it is possible to build a DSL toolkit that is suitable for enterprise software development and offers ease of use comparable to existing DSL tools. In particular, the contributions of this thesis are:

1. analysis of the requirements that DSL tools should satisfy so that they fit well into the overall development process;

2. analysis of state of the art of DSL tools with respect to these requirements;

3. description of Simpl: a DSL toolkit that satisfies these requirements; and

4. evaluation of the Simpl DSL tool, measuring both its suitability for the intended purpose, and its usability.

The thesis is partly based on previous publications. Contribution 1 and part of Contribution 2 are reported in reference [21]. An initial overview of the Simpl tool (Contribution 3) is given in reference [23]. This reference, co-authored with David Pugal from Cybernetica AS[1], also contains parts of contribution 4. The other parts of contribution 4 are previously unpublished. Finally, Contribution 1 is partly inspired by a case study on the application of DSLs that was published in reference [20]. This latter article is the basis for Chapter 2.

1.4 Structure of this Thesis

Chapter two reports on a case study in which DSLs were used as the backbone to develop a suite of customs information systems. It provides an overview of the role of DSLs in the development of this solution and quantifies some of the benefits derived from the use of DSLs, both during the initial development and during the ongoing maintenance phase. The chapter also describes a scenario that motivated this research.

Chapter three analyzes this motivating example and develops and spells out a set of requirements for DSL tools.

Chapter four presents overview of the state of art with regards to the previously formulated requirements.

Chapter five describes Simpl DSL tool that is targeted at enterprise software development. Simpl creates DSL implementations that can be embedded in a bigger enterprise system and offers good usability for professional developers.

Chapter six presents four studies that evaluate the usability of Simpl DSL tool and its suitability for enterprise software development. First, we reimplement two DSLs from the customs system described in chapter one. Sec-

[1]However, the author of this thesis was the main author of the paper.

ond, we implement compiler to Oberon0 programming language and compare the result with implementations created with other language tools. Third, we measure code metrics of different implementations of a fairly complex DSL. Fourth, we perform a controlled usability study where subjects implement a DSL using either Simpl or a baseline DSL tool. We measure time spent and user satisfaction.

Finally, we present our conclusions and give directions for further research.

CHAPTER 2

A CASE STUDY

2.1 Customs Engine

Since 2005, Cybernetica AS is engaged with the Estonian Tax and Customs Board in building Customs Engine (CuE) – a suite of systems for processing customs documents (customs declarations, manifests, warehousing notices etc.). Each type of document reflects a movement of goods (e.g. between a ship and a customs warehouse) and is associated with its own set of rules, regulations and procedures that need to be carried out. Given the document-centricity of the domain, we decided to decompose CuE into subsystems, such that each subsystem is responsible for processing one type of document.

CuE is developed using Java Enterprise Edition and can operate on standard Java EE application servers. Each CuE subsystem is an independent application that can be deployed and used separately. In addition to providing user interface for manipulating a particular type of document, the CuE subsystems communicate with other CuE subsystems and their counterpart systems in the European Union (EU). CuE follows principles of service-oriented architecture (SOA). The subsystems send notification messages and implement services that can be called by other applications. This principle can be illustrated with an example.

The Export Control System (ECS) manages export reports (information about exported goods that cross the EU border). In a common scenario an export declaration is lodged in Estonia, but the goods exit the EU via Latvia. The declaration is submitted to the subsystem for processing customs declarations (SAD). After completing the necessary formalities,

the SAD releases goods and notifies the ECS. The Estonian ECS subsystem creates new export report and sends it to the Latvian ECS system. When goods cross the border in Latvia, the Latvian ECS sends notification back to the Estonian ECS.

The subsystems have functional and technical similarities. In order to take advantage of these similarities, we have adopted a software product line engineering approach [62] by developing a generic platform for building customs information systems. The platform contained two types of items. First, it contained components that encapsulated certain functionality, such as risk analysis or interacting with the other EU member countries. Sometimes these components could be reused as is, other times it was necessary to configure or customize the components for each of the CuE subsystem. Second, the platform contained frameworks that encapsulated the architecture and control flow of a certain part of the system, such as document editor or document state machine.

The development of the Customs Engine relied on several DSLs. The use of DSLs was driven by the following concerns.

- Because there is significant variability between CuE subsystems, the components in the platform must be highly flexible and configurable.
- CuE implements sizable and complex legislation and working procedures that change over time. Therefore, it is important that processing rules are implemented in a way that provides a clear overview of what the system does and facilitates rule evolution. Preferably, updating the rules should not require additional programming effort nor rebuilding or restarting the system.

These concerns match quite well with the advantages gained from using the DSL approach as discussed in the introduction.

The author of this thesis was a software architect at the team who implemented the customs engine. When started designing the first subsystems, we anticipated that business logic for verifying correctness of the submitted customs documents (together with checks to external registries and calculation of some fields, such as taxes) will be complex and volatile. The complexity stems from the fact that customs is very heavily regulated in the European Union and there are extensive standards and regulations concerning each type of document. Volatility comes from two sources. First, the EU requirements were new both to us and to domain specialists in Estonian Customs[1]. This implied iterative development of the business logic

[1]We started building CuE shortly after Estonia's accession to the EU.

with continuous feedback from the users to ensure that the functionality of the system corresponds to the actual working procedures of the customs. Second, both legislation and customs working procedures change over time. This implies that histories of verification rules needed to be maintained – older documents had to be verified using rules that were in effect at the time the document was first submitted.

Our solution was to encapsulate the document verification logic into a separate component. This component receives the document to be checked (in XML format) and returns list of verification results (errors, warnings or tasks for the customs officer). The verification rules are not hard-coded in the component, instead they are programmed in a DSL and the verification component contains an implementation of the document verification DSL. Although it could have been possible to implement this component using an off-the-shelf rule engine (such as Drools[2]), we decided to implement a custom document verification language. This was driven by the following considerations.

- We predicted that the amount of verification rules will be very large (each subsystem will have its own set of verification rules). Therefore, improvements in usability of the rules can offset the costs of developing a custom verification rule language.
- Upon analyzing the rules, we discovered that most of them follow one among a few recurring patterns. We therefore decided to implement direct support for these patterns so that for simple rules, the "right thing" is done automatically.
- We wanted to have tight integration between the user interface for document creation, the verification module and the other verification steps (such as simply checking presence of mandatory fields or whether field contains the correct classificator value). This tight integration allowed the user interface to provide precise visual feedback about fields that contained verification errors or warnings.

The design of the document verification language is detailed in Section 2.2.1.

Building on this experience, we started structuring most of our main components into two layers. The higher layer contains a formal specification of what the component does, written in a high-level DSL. The lower layer is the implementation of this DSL. Our aim was to make the executable specifications in the higher layer concise, human-readable and non-technical.

[2]http://www.jboss.org/drools/

- Concise – the specification is short enough so that it is possible to read and understand it in its entirety.
- Human-readable – readable by non-technical persons, such as analysts or domain specialists.
- Non-technical – free of technical details, such as specific Java language patterns.

When trying to create comprehensive DSLs, we usually encountered some cases that were quite complicated. In general, implemented these cases required a language that had similar power to a typical GPL. Sometimes it was also necessary to access the internals of the system. Implementing these use cases would have lost most of the benefits gained by focusing on a specific domain (conciseness, readability, non-technical nature) and made the language similar to a general programming language.

Instead of trying to express all the details using a DSL, we followed a 80-20 approach and focused on brevity and clarity. We used the DSL for giving overview of the component's behavior (describing the "essence" of the component) and solved the complicated corner cases by implementing them in Java and calling the Java code from the DSL. This kept our DSLs simple and readable and enabled us to separate the "what" (the high-level logic/structure of the program) from "how" (low-level technical details). For the DSL user, the calls to Java functions were opaque building blocks that were composed using the DSL.

2.2 Domain-Specific Languages in the Customs Engine

The Customs Engine relies on a DSL for expressing document verification rules in the customs domain (namely Burula) and a number of configuration DSLs. These DSLs are outlined below.

2.2.1 Burula

Burula illustrates our approach to implementing large chunks of business logic using DSLs. Burula is the language that is used in the verification component to specify rules for verifying correctness of the documents submitted to the system. Each of the CuE subsystems contains different set

of verification rules, specific to this type of document. The input to verification process is the document in XML format and the output is a list of verification errors. Each verification error contains human-readable description of the error and location of the erroneous value.

Burula is the result of several years of iterative development. The initial versions of the document verification language were designed in *ad hoc* manner, without much advance information about the verification rules. Burula was developed based on our experiences with the previous verification languages and analysis of the rules written using the previous languages. This allowed us to optimize for the most common case.

The most common type of correctness check concerns dependencies between fields[3], following the pattern "if field X contains value 'Foo', field Y must contain value 'Bar"' (where 'Foo' and 'Bar' can be sets of possible values or simply presence/absence of a value). About 80-90% of all checks fall into this group. The rest of the verification involves comparing the submitted document with other data – previous versions of the same document, licenses, permits, various reference data, etc. This other data usually resides in other systems and must be queried via various interfaces.

When designing Burula, our intention was that analysts and domain specialists at the customs board would write and/or modify the verification rules. From this goal, we derived the following main requirements for the verification rules language.

- The rules must be structurally similar to the rules expressed in a natural language.
- The rules must contain a minimal amount of technical information. For example, referring to document fields must be possible without spelling out the exact location of fields (e.g. user should be able to use "packages" instead of "declaration.goodsItem.packages"). Additionally, iteration over repeated elements should happen transparently.
- The language must have strong static type checking.
- The rule language must enforce a good style. In particular, it should discourage writing long, complex rules with complicated boolean expressions, containing unexplained "magic" values[4].

[3]Simple "field X is required/forbidden" checks are performed in other verification steps.

[4]For examples of such rules, consider the examples in figures 2.1 and 2.2 with all the predicates in-lined into the main rule. When one adds additional conditions and nested AND or OR operations, determining conditions under which the rule applies becomes difficult.

```
predicate is-unpacked-goods
    kindOfPackages is ('NE', 'NF', 'NG')

packages must have numberOfPieces
        when is-unpacked-goods
    error "When goods are unpacked, number
        of pieces must be present"
```

Figure 2.1: Example of a simple verification rule

- Because the verification rules change more frequently than the core business logic, it must be possible to change the rules without restarting the system. Additionally, rules must be versioned and old documents must be verified using rules that were in effect at the time when the document was submitted.

Figure 2.1 contains an example of a business rule written in Burula. In a natural language, this rule can be expressed as follows:

> If the goods are unpacked (indicated by using classificator codes "NE – unpacked", "NF – unpacked, 1 item" or "NG – unpacked, several items"), then the "number of pieces" field must be filled in[5].
>
> *Remark: this check applies to all the package descriptions in all the goods items of the declaration.*

This rule takes advantage of the implicit iteration in Burula. The Burula implementation automatically iterates over all the goods items and all the package descriptions inside the goods items. It also applies the predicate *is-unpacked-goods* only to the package description that is currently examined by the rule. This example also shows how Burula tries to enforce good writing style by restricting the complexity of the rule. The condition on *kindOfPackages* field cannot be used directly as part of the rule and must instead be written as a separate predicate, ensuring that it will be named (and thus giving information about its purpose).

Figure 2.2 shows a more complicated rule. It checks whether the itinerary for the export movement contains more than one country. *XmlEcs* is the name of the root element of the XML document representing an export report. It indicates that scope of this rule is the whole document (as opposed

[5]For packed goods, the "number of packages" field is used instead.

```
predicate is-ship-supplies
    specificCircumstanceIndicator = 'A'

predicate is-postal-consignment
    specificCircumstanceIndicator = 'B'

XmlEcs lengthOf itinerary > 1
        unless is-ship-supplies
            or is-postal-consignment
    error "If the declaration does not contain
        ship supplies or postal consignment, the
        itinerary must contain at least
        two countries."
```

Figure 2.2: Example of a more complicated Burula rule

to example in Figure 2.1 where the scope is one package description). The scope of the rule affects how the Burula implementation iterates over fields with multiple values (e.g. goods items). Additionally, it is used to determine which field should be used as location of the verification error (all the generated verification errors contain pointers to the field that contains the error).

Burula is implemented as a compiler that compiles the Burula source file to Java bytecode (class files). This provides performance comparable to writing the rules in Java language. Additional benefit is improved integration with rest of the system. It is easy to call Burula programs from Java code and Java methods from Burula (for example, the *lengthOf* function in the previous example is implemented in Java). The helper function calls are performed in the same execution context as the rest of the system and the Burula rule.

The Burula compiler is embedded into the CuE system. The user can load the rules via user interface. The system then compiles the Burula programs to Java bytecode (class files) and saves them into a database from where they are loaded when the user runs the rules. Because of the dynamic loading, the verification rules can be modified without restarting the system.

The embedding of the Burula compiler into the system serves two purposes. First, it allows designated end users to change the verification rules themselves without the help of systems administrators. Second, it functions as a security measure. If the Burula programs were compiled off-line and loaded to the system as class files, then it would be possible for a malicious

(or simply careless) user to introduce arbitrary code into the system. In our case, the compiler is embedded in the system and the loaded Burula programs are restricted to a narrow set of tasks related to verification of customs documents.

The business rules are created and managed offline, using standard text editor and Subversion[6] for version management. Initially, we experimented with embedding the rule editing functionality into the CuE user interface, but it soon turned out that our web-based tools were inconvenient for this purpose and that version control was best performed using standard tools.

Burula toolset consists of the following items:

- Burula compiler and runner that is embedded into each CuE subsystem;
- Standalone compiler that can be used for testing of the verification rules. It takes as input rule file and XML input document and outputs list of verification errors;
- Eclipse-based Burula IDE (see also Section 6.1.1) that can be used to edit and manage Burula programs.

2.2.2 Configuration DSLs

The document verification part of the system contains a bulk of application-dependent business logic. For these parts, creating a complex, special-purpose DSL implementation can be justified, especially if the costs of this implementation can be shared among several subsystems built on the same platform. In addition to verification DSL we used several smaller configuration DSLs for customizing the platform components for each of the CuE subsystem. Compared to the document verification module, the other platform components had some important differences.

- The configuration DSL programs are simpler and contain far less rules than Burula programs.
- The configuration DSLs have a more technical nature. While the verification rules are almost entirely derived from external specifications and user requirements, the configuration DSLs often express technical

[6] See `http://subversion.tigris.org/`

concerns, e.g. configuration of a messaging component. Also, configuration DSL expressions are less likely to change during life of the project.

- Configuration DSL expressions are mostly written by technical people (programmers and architects) and therefore there is less emphasis on friendly syntax.

Because there are many small configuration DSLs, the cost of each individual language has to remain low. We could not afford to implement custom compilers for each of these languages. Therefore we decided to implement these languages using a template engine to generate Java code and/or XML configuration files from the DSL programs. This approach has several benefits.

- We can reuse existing Java compilers and tools, thus lowering implementation cost.
- We inherit features from Java. For example, with clever use of Java types in the generated code, it is possible to use Java type checker for verifying the correctness of a DSL program. This is done by creating a separate Java class or interface for every concept in the DSL language. These classes encapsulate the allowed operations and interactions between different concepts. Thus, an invalid DSL program will generate type-incorrect Java code that is detected by the Java compiler.
- We can embed Java statements and expressions inside DSL code. With this approach, the DSL can be relatively simple and support the most important cases. For complex corner cases, we can use Java directly.
- Input files to template engine share the same base syntax. This simplifies learning the next configuration DSL.

We reused previously developed in-house template engine called *Templater*. Templater's main strengths are good performance, support for Java language and good integration with our build system.

Figure 2.3 shows the working principle of Templater. The input file contains a program in a DSL. The template matches elements in the input file and generates parts of the output file using the data contained in the matched

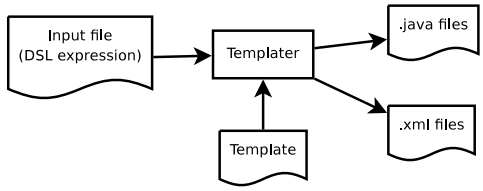


Figure 2.3: Use of Templater

input element. The input files have standard syntax that is based on S-expressions used by the Lisp family of languages [59].

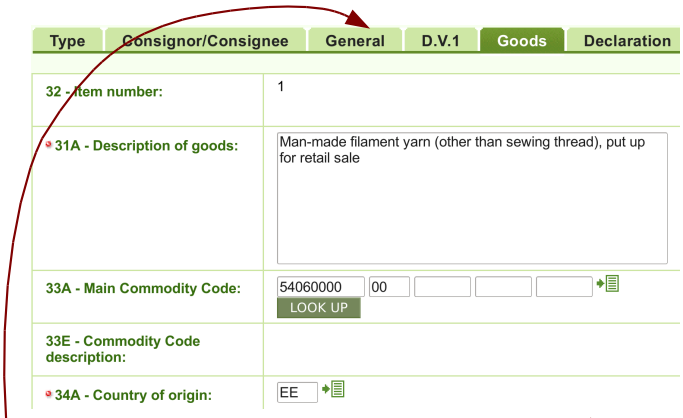
Figure 2.4 shows excerpt of configuration DSL expression that describes layout of a document editing user interface. The document editor uses tabbed, wizard-like user interface that displays document fields in a form. All the document fields are assigned an alphanumeric code (such as 10, 23A) to make it easy to refer to them in documentation. The excerpt on Figure 2.4 defines which tabs are displayed in the editor and which document fields go to which tabs.

2.3 Our Experiences

2.3.1 General Experiences

Our overall experience in using DSLs, based on four years of developing and maintaining CuE, is quite positive. CuE subsystems are quite small in code size – typically between 5000 and 25000 lines of code (including all the code written in DSLs). The platform itself is about 100000 lines of code. We attribute the small code size to extensive reuse and raised level of abstraction – parts of the system were coded in high-level DSLs.

From the software architecture standpoint, our extensive use of DSLs forced us to separate the system into a "business" part and a "technical" part and to define explicit interfaces between these parts. This has led to clear separation of concerns and enables us to change one part of the system without noticeably affecting the other part. Our experience from building CuE shows that about half of the platform components could be reused simply by instantiating platform classes and calling methods. However, the other half of the components had to be parametrized with code. The required code was often quite technical and consisted of several related classes and methods. When using DSLs, this complicated code can be




```
(tabs
   (type "Type")
   (subject "Consignor/Consignee")
   (common "General")
   (valueinfo "D.V.1")
   (goods "Goods")
   (decl "Declaration")
)

(default-tab common)

(descriptions
   ...
   (32 goods)
   (31A goods TextAreaWidget)
   ((33A 33B 33C 33CC 33D) goods CommodityCodeField )
   (33E goods)
   (34A goods)
   ...
)
```

Figure 2.4: Document editing screen and its corresponding DSL code

generated from concise and human-readable DSL program, making reuse of the platform components much easier.

Besides its impact on the technical architecture, the use of DSLs has also influenced the way we work. One notable change is that much of detailed programming work has been transferred from the programmers to the analysts. Instead of writing detailed documents about business rules or creating mock-up prototypes, the analysts often write the appropriate code themselves, using DSLs. This has led to a quicker and more iterative development cycle. The analysts can now directly try out rules and screen descriptions, receiving immediate feedback to their work. Instead of requiring detailed, up-front specifications, we can start with some initial specification (often provided by the EU), implement it and then start iterating with continuous feedback from the actual end users. An additional benefit is that the analysts are required to specify requirements in a formal language. This forces them to really think through the rules and procedures. If the analysis results were written down as human-readable documents, the errors, inconsistencies and lack of details would only be discovered during programming or even in a later phase.

The shift to using DSLs has impacted the amount of documentation produced by analysts. Because the analysts implement some of the functionality themselves, they no longer write lengthy human-language documents. This means that the programs written in DSL become (formal and executable) documentation of the system's behavior. On the positive side, this is desirable, because this representation is more clear and precise. On the negative side, the formal descriptions are not easily understood by client representatives who usually do not have technical background. Also, formal descriptions focus on the "what" or the "how" rather than the "why". Therefore, there is still a need for human-readable documentation that presents a non-technical overview of the system's functionality and a rationale of key design decisions.

One (purported) advantage of using domain-specific languages is the ability to involve domain experts in the design process of the system[88]. This was also our initial intention when building CuE. For example, Burula was specifically designed with non-technical end users in mind. Our initial plan foresaw that our analysts create the initial set of rules and from there on, the domain experts will take over the development of the rules. However, when we held training sessions for writing the rules, it became clear that this plan was not practical. The users struggled even with the very basic concepts of computer programming (such as a need to express oneself in a very formal, constrained language). It appears that even programming in

a language crafted specifically for a given domain requires formal thinking skills that the users without an IT background often do not possess[7]. We found it more cost-effective to train our own business analysts (who have more technical background than the end users) rather than training a larger pool of domain experts who did not have any IT background. However, after our own analysts had written the verification rules, the domain experts were able to read and, sometimes, even to modify them.

One practical issue that we encountered was that of tool support for our DSLs. The domain-specific languages are almost by definition not mainstream languages and therefore lack high-end tools created for widespread languages, such as Java. Because CuE contained several DSLs, we could not spend much resources on creating specialized tools for each language. In general, we added syntax highlighting support for a few common editors, but did not create IDE-style tools with full support for auto-completion, real-time syntax error detection, integrated debugging, etc. This frustrated some programmers who felt that writing DSL programs using standard text editors is not as convenient as writing the same functionality in Java, although the latter may be longer and potentially less readable. In this case, a DSL tool with good support for creating IDEs would have been very useful to alleviate these concerns.

2.3.2 DSL-Related Workload

When using DSLs, one operates under the assumption that the costs of creating the DSLs are paid back later during development and maintenance of the software. This section presents some data about costs related to maintaining the DSL programs and the DSLs themselves.

The most important DSL in CuE was Burula. The amount of effort spent on developing Burula (creating specification, programming the compiler and runtime, integrating Burula into CuE subsystems, creating toolset) was about 4 man-months. This cost is shared between the 6 subsystems of CuE. Note that Burula was built by hand without using any DSL tools. With proper tool support, the effort would have been reduced significantly.

A moderately complex CuE subsystem has about 100-150 verification rules, most of them derived from the EU specifications. For example, developing the 131 verification rules for the ECS subsystem took about 2.5 man-weeks.

[7] As a side note, we discovered that domain specialists without IT education had also difficulties when writing detailed and unambiguous natural-language specifications (such as descriptions of the verification rules).

This time was spent on learning Burula, reading and analyzing the data model and specification on data constraints, writing and testing the rules. The most complex CuE subsystem is SAD, responsible for managing import and export declarations. SAD contains about 2000 verification rules. These rules are derived from EU legislation, national regulations and working procedures of the Estonian Customs.

The technical DSLs were considerably simpler to implement. Creating a very simple DSL with code generator took about 2 man-hours.

CHAPTER 3

REQUIREMENTS

Starting from the experience reported in the previous chapter, this chapter formulates requirements for a DSL toolkit for EIS. The chapter starts with a formulation of desiderata (or goals) directly illustrated on the case study. Next, we review the typical components provided by a DSL tool. Finally, we analyze specific functional, integration and non-functional requirements and their relation to specific DSL tool components.

3.1 Goals

In the case study (Chapter 2) we identified the need for tool support for at least two kinds of DSLs. The first kind is used to generate technical, "boilerplate" code that is needed to make use of a framework. In this case, one needs to be able to implement simple parser and code generator. The program checking can be done on generated code (as it was done in the Customs Engine project). Basic IDE with syntax highlighting is needed to edit the code. It must be possible to integrate the DSL implementation with the build system so that the code generator can be called during the system build. Because there can be many technical DSLs in the system, there is need to make the cost of implementing a simple DSL-based code generator low. Creating a new DSL should take only several hours of work for a person with sufficient expertise.

The second kind is a complex DSL that is mostly aimed at non-technical people. As such, it needs to have sophisticated tooling, such as full-featured IDE and program checker. The code generator can also be nontrivial. If the DSL programs are compiled at run time, it must be possible to embed the code generator part of the DSL implementation into a bigger system.

Next, we generalize these scenarios and derive a list of general goals that a DSL toolkit should fill in order to fit well into enterprise software development niche.

- First, the DSL implementations created with the DSL tool must be able to function as part of a larger system. In other words, they should be embeddable. At minimum, it must be possible to integrate the DSL implementation into the build system of the EIS so that it is used to process the DSL programs. Ideally, it should be possible to embed the DSL implementation into the EIS itself, as this enables additional development possibilities (in the Burula example, loading the rules via the user interface). Section 3.3 provides more detailed analysis of integration and embedding scenarios.

- Enterprise software development is generally done by professional software developers who are not experts in programming language implementation and, especially, the relevant research. In order to gain acceptance of the developers, the DSL toolkit should lower the barrier of entry by minimizing the number of changes that must be made in the working environment and number of additional tools that must be installed and learned. If possible, the DSL toolkit should be based on existing tools and programming languages (learning a new programming language is often a major undertaking and may be difficult to fit this into project schedule). Since the professional developers are used to IDEs, the DSL tool should also provide an IDE.

- Following the Burula example, the DSL tool must allow implementing complex DSLs. In particular, it must be possible to implement a compiler for a DSL that has the expressive power of a typical functional or imperative language.

- Following the configuration DSL example, the DSL tool must allow implementing simple DSLs with very little effort. There should be no need to keep in sync descriptions of various aspects of the language and no need to manually write boilerplate code.

- Since the developers are accustomed to IDEs, the DSL toolkit must make implementing an IDE for the DSL simple. Especially, creating a reasonable IDE for a simple DSL should be done with little or no programming efforts. In addition, there must be a possibility to implement a full-featured IDE for a more complex DSL.

Based on these goals we derive requirements for enterprise DSL tools. Section 3.2 describes functional requirements. Sections 3.3 and 3.4 develop a list of non-functional requirements.

3.2 DSL Tool Components

Typically, a DSL implementation contains the following components, which map one-to-one to the key functional requirements of a DSL tool. Other models for decomposing the DSL implementation can be found in [96] (structure, constraints, behaviour), [49] (the extract-analyze-synthesize paradigm), [19, 94] (discussion of various DSL components), and [54] (feature model of DSL implementation).

- **Parser** is typically implemented via parser generator or parsing library. Parser generator takes as input grammar description for the DSL and generates a parser, typically implemented in a GPL such as Java. Parsing library provides functions and classes that simplify implementing the parser in a GPL. Parser libraries are often combinator libraries (such as parsec [56]).
- **Program checker** inspects the parsed program and detects errors, such as name or type errors or constraint violations. This can be programmed in GPL with the help of various libraries. Alternatively, this can be achieved by special-purpose program transformation language or, e.g., by an attribute grammar system.
- **Program transformation** transforms the AST. The transformed AST can be same language (e.g. for program simplification or optimization) or it can be different language (such as for compilation).
- **Code generator** prints the (possibly transformed) AST as text. This is useful for compilation (translating DSL to GPL or assembler) or source-to-source transformation (e.g., pretty-printing or optimization).
- **Program editor** allows editing of the DSL programs. The editor can be a part of an IDE, such as Eclipse, or it can be a standalone program.

In the simplest cases, some components can be either not present or be very simple. For example, the program checker and program transformation

components can be missing in very simple cases. Also, any text editor can be used as the program editor.

Ideally, a practical DSL toolkit should be able to include support for implementing all the necessary DSL components. Some authors [26] have argued that beyond this minimal support, DSL tools should evolve towards providing support for development of debuggers, testing engines and profilers. However, the challenges associated with making the development of such tools cost-effective are still an open research area.

3.3 Integrating DSLs into EIS

In general, all the functionality of an enterprise information system cannot be expressed by a single DSL program. The EIS contains several concerns (persistence, distribution, business logic) that each are best handled using a separate DSL. For example, the Java Enterprise platform includes several XML-based technical DSLs. Additionally, different parts of business logic (workflows, state machines, verification rules) are often best expressed using different DSLs. Following that logic, a DSL-based EIS would typically consist of components written in different DSLs and glued together by code manually written in a general-purpose language, such as Java. Therefore, one important question is how the DSL code can be called from the rest of the EIS. This section lists different scenarios for integrating the DSL implementation into the EIS.

The biggest factor influencing the options for integrating DSL code and "glue" code is whether the DSL is interpreted or compiled. In this thesis, the line between interpretation and compilation is drawn according to whether the DSL interpreter is part of the application code or part of the environment (language runtime, hardware)[1].

The integration options listed in this section mostly differ in the point of time when the DSL program is packaged/compiled and loaded into the system. From the DSL user's point of view the main difference between the options is the versioning model of the DSL program. For example, if the DSL program is packaged with the rest of the source code of the system

[1]For example, an application can be written in the Python language and the DSL program translated to Python (or Python bytecode), which is then loaded into Python runtime. We consider this approach to be compilation, because the code is interpreted by the environment (Python runtime). However, if the DSL program is translated to a form that is interpreted by the Python code in the application itself, then we consider it to be interpretation.

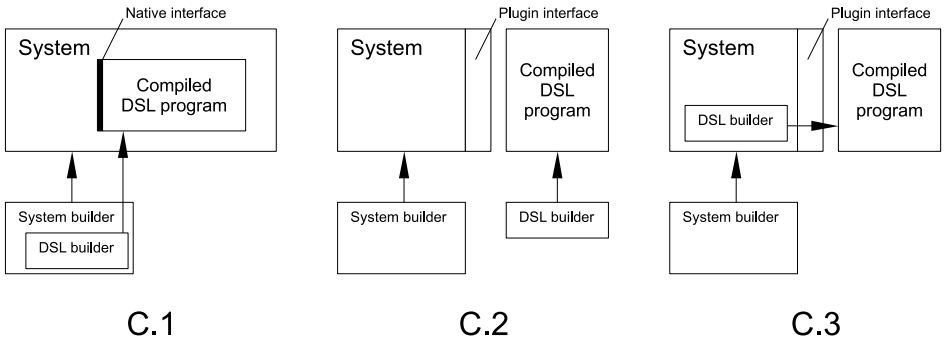


Figure 3.1: Options for deploying compiled DLSs

then the DSL program should be versioned together with the rest of the source code.

There are two principal ways of deploying DSL programs that use interpretation.

I.1. DSL program can be packaged with the application source code as a text resource. For example, this option is used in the Java Enterprise platform for various configuration and manifest files. With this approach, the life-cycle (deployment schedule, versioning policy) of the DSL program will match the life-cycle of the other application code. Changing the DSL program involves redeploying of the whole application. Therefore, this approach is mainly suitable for technical DSL programs that change with the application code.

I.2. DSL program can be loaded at runtime and stored in a file or a database. The program can either be in the source form, or it can be compiled to some kind of bytecode. The application can also provide environment for editing, testing and debugging the DSL programs. Because changing the DSL program does not involve changing or redeploying the rest of the application code, this approach is suitable for non-technical DSL programs that capture fast-changing business requirements.

For deploying compiled DSLs, there are three options for packaging and loading the DSL programs (see figure 3.1).

C.1. The DSL program can be compiled during the build process of the system. The compiled DSL program is then packaged and deployed

together with the rest of the system. Processes for changing and deploying the DSL program are exactly the same as for changing and deploying rest of the application code. The DSL program thus becomes integral component of the system that is integrated with other, possibly hand-written components. This approach is suitable for technical DSLs (essentially programming at higher level of abstraction).

C.2. The DSL program can be compiled separately and loaded into the system at run time (e.g. as a dynamically loaded plugin). The DSL program can be managed separately from the application code and can be used for describing business logic that changes more often than the application code. This option requires creating a special tool or a build system that is able to compile the DSL program and package it for deployment.

C.3. The DSL program can be loaded into the running application and compiled by the application code. The compiling can be done in several steps, for example by first generating Java source code and then invoking the Java compiler. The compiled DSL program is stored in a file or a database and loaded at run time. Using this option requires packaging the DSL compiler with the application software. As with the previous option, the life-cycle of the DSL program is not tied to the life-cycle of the application code.

The main difference between the last two options is the environment where the DSL compiler is run. The option C.3 can be somewhat more complicated to implement because the runtime environment must also contain all the development tools and libraries needed for compiling the DSL programs. Also, invoking the DSL compiler from the application code can be complicated and resource-consuming. However, embedding the DSL compiler into the application software can offer several benefits. First, it lowers the requirements for the working environment of the business engineer[2]. If the application software includes a web-based DSL editing tool, the business engineer can create DSL programs using only a web browser. Second, if the DSL program comes from untrusted sources (e.g., end users customizing their user experience) then it is possible to analyze the DSL program before compiling to ensure it meets the requirements. If the DSL program is compiled before loading into the system, then the analysis becomes more difficult as it becomes necessary to reverse engineer the compiled code.

[2]Business engineer is the person who describes solutions to business problems with DSL program. See [27] for a full taxonomy of the roles in DSL-oriented development.

3.4 Non-Functional Requirements

This section formalizes non-functional requirements that should be fulfilled by a DSL tool(kit) that aims to support most of the scenarios encountered in enterprise software development. The requirements are based on goals from Section 3.1 and integration scenarios described in Section 3.3. Each requirement is given a short code that will be used to refer to it in later text.

LANG Writing program checking and program transformation code should not require the developer to learn new non-mainstream programming language. Fulfilling this requirement makes the tool more accessible to industry developers.

SMALL Implementing small and simple languages must be as simple as possible. In particular, implementing a small language should not require writing boilerplate code and repeating same information (such as AST structure) in several places. Fulfilling this requirement makes it possible to use the tool for implementing simple, one-off code generators. (Note that this requirement only measures the effort needed to implement a small DSL by a developer who knows the tool. It is not tied to the time it takes to learn the tool.)

SEP_WB If the DSL tool contains an IDE builder, then it is possible to use the generated DSL IDE separately from the DSL tool. The user of the DSL IDE cannot change the language definition. Fulfilling this requirement makes it possible to separate the work of the language engineer/transformation specialist[3] and the business engineer.

BUILD It is possible to integrate the DSL compiler into the build process of the system. In particular, this means that the DSL compiler must be able to operate in non-interactive environments, e.g. when called by a build script or when used in an automated build server. Fulfilling this requirement makes it possible to implement scenario C.1.

SYSTEM It is possible to integrate the DSL compiler into the system. In particular, this means that the DSL compiler can be deployed

[3]The language engineer creates the language description and the transformation specialist creates the language implementation. See [27] for a more thorough discussion.

as a library that can directly called from the application code. This requirement is stricter than the requirement BUILD. In principle, if the tool can be invoked from the build system, then it is possible to invoke it from the system as an external program. However, the requirement SYSTEM means that the DSL compiler can be invoked without the overhead of creating another process. Fulfilling this requirement makes it possible to implement scenario C.2.

VCS The DSL tool supports version control of the DSL programs. If the DSL uses human-readable textual syntax, this is very easy because storing and merging text files is supported by all the popular version control systems. However, if the DSL program is stored in some internal format (e.g. serialized XML representation), then the DSL tool must include explicit support for version control. Fulfilling this requirement makes it possible to implement scenarios I.1 and C.1.

CUSTOM The DSL tool supports creating customizable DSL implementations that contain basic functionality and offer extension points where application-specific customizations can be applied. Examples of these customizations are set of objects that can be manipulated by the language and operations that can be applied to the objects. Customization can also be achieved if it is possible to divide the language implementation into smaller modules or fragments and then reuse these fragments in different combinations. Creating customizable DSLs and composing a DSL from several language modules may be needed if the EIS is built as a software product line.

CHAPTER 4

STATE OF THE ART

4.1 Overview

This chapter presents the state of the art in DSL tools and evaluates them with respect to the requirements discussed in the previous chapter. Because this thesis focuses on tools for industrial software development, we restrict our evaluation to tools that are reasonably mature, are supported by a community or a company and are available to the public. These criteria are elaborated below.

- Maturity. The tool should have a history of practical use for several years. It should have a stable current release that can be used for developing production-level code. There should be cases of using the tool in several production systems (this requirement excludes pure research tools that have been developed and used only once or twice). The tool should be documented reasonably well and work without crashing or malfunctioning.

- Support. The tool should currently be in active maintenance and new releases (or bug fixes) should appear periodically, in reasonable time intervals. There should be an active user community that supports the tool via mailing lists or user forums. Alternatively, there should be a company providing support for the users of the tool.

- Availability. The tool should be available to general public under a reasonable license. Open source licenses are preferable because they lower the risks caused by e.g. insufficient documentation or community support.

This evaluation only looks at tools that are directly aimed at creating language (especially DSL) implementations. Thus, it does not include tools like K [73, 76] or Maude [10, 11] that are mainly targeted at specifying and analyzing language semantics. Also, the evaluation is not comparative, meaning that we do not establish direct one-to-one parallels between pairs of tools. Instead the evaluation is based on the requirements formulated in Chapter 3.

Several detailed comparative evaluations of tools have been published in prior work. Other comparisons and overviews of DSL tools can be found in [66] (compares Microsoft DSL tools and Eclipse Modeling Framework), [54] (compares MetaEdit+, Microsoft DSL tools, and XML-Mosaic), [69] (compares openArchitectureWare, MPS, MontiCore, IMP, TCS, TEF, and CodeWorker), [91] (ANTLR+StringTemplate, Ruby internal DSLs, Stratego, and Converge), and [94] (parallel examples in Spoofax, Xtext and MPS).

In order to make the overview more compact, different tools are grouped according to their architecture and main purpose. In each group, one tool is reviewed in depth while other tools are compared to it. Note that this evaluation is based on the state of art in 2009-2010. In some cases, the updated version of the evaluation is presented in Chapter 5.

Each tool is analyzed in terms of the functions of a language development tool outlined in Section 3.2, namely parser, program checker, program transformation module, code generator, and program editor. For every tool, we only describe the components that are applicable to the tool. In addition, non-functional aspects are discussed for each tool or group of tools.

As stated in Section 1.1, this dissertation focuses on textual DSLs. Accordingly, the following review of the state of the art does not consider Domain-Specific Visual Language (DSVL) tools such as Microsoft DSL Tools, MetaEdit+ [45, 46], DEViL [74] or Pounamu [100].

Throughout this chapter we illustrate the tools with an example DSL for describing JavaBeans reusable software components, also called Java beans. The DSL specifies the name of the class and attributes together with their types. The attribute types can be either regular Java types or references to other beans. Java bean DSL programs can be used to generate Java code. Figure 4.1 shows an example DSL program that describes two beans. First, *Author*, contains only one attribute of type *String*. The second bean, *Book*, contains two attributes: *title*, typed String, and *author*, a reference to the other bean.

```
bean Author {
    name String
}

bean Book {
    title String
    author ref Author
}
```

Figure 4.1: Example program in the Java bean DSL

4.2 EMF-Based Tools

This group contains tools that are based on the Eclipse platform [15] and in particular the Eclipse Modeling Framework (EMF) [84]. In particular, they use the EMF models for expressing the AST of the DSL programs. They have some similarities with language workbenches (see Section 4.3 for discussion of language workbenches and comparison of the two categories), but are covered here as a separate group.

Common feature in this category is that the DSL implementation is deployed as an Eclipse plugin. This plugin can be distributed separately from the DSL workbench (however, the plugin will depend on the core packages of the DSL tool). Thus, the tools satisfy the SEP_WB requirement. All the tools in this group store DSL programs as plain text files, thus fulfilling the VCS requirement.

All the tools in this category use EMF models for expressing the AST. Thus, all parts of the DSL implementation will be tightly coupled to the Eclipse platform. Thus, they fail the SYSTEM and BUILD requirements. Although, it is technically possible to build Eclipse-based tools from the command line, it is quite difficult and involves extensive setup and specialized tools.

4.2.1 Xtext

Xtext [24, 98] is an EMF-based language development environment that aims to be a complete language infrastructure covering all the aspects of developing a programming or domain-specific language.

Parser Language development in Xtext starts with creating a grammar description for the language. The grammar description will then be used to

generate ANTLR [67] parser and Ecore metamodel [84] for expressing the AST of the DSL programs. Grammar annotations can be used to direct the generation of the metamodel. Figure 4.2 shows grammar description for a simple DSL for generating Java beans. Lines 1-2 declare the name of the grammar and include the standard grammar library *Terminals* that contains common grammar rules, such as strings, identifiers (in example, it is referred to as implicitly defined rule *ID*), and comments. Line 4 declares name of the Ecore package that will contain the metamodel for the bean language. The rest of the grammar consists of context-free rules starting with symbol *Model* that represents collection of beans.

```
grammar example.bean.BeanDsl                                          1
    with org.eclipse.xtext.common.Terminals                           2
                                                                      3
generate beanDsl "http://www.bean.example/BeanDsl"                    4
                                                                      5
Model: beans+=Bean*;                                                  6
                                                                      7
Bean: "bean" name=ID "{"                                              8
          attrs+=Attr*                                                9
      "}";                                                            10
                                                                      11
Attr: name=ID type=AttrType;                                          12
                                                                      13
AttrType: BeanRef | ID;                                               14
                                                                      15
BeanRef: "ref" target=[Bean];                                         16
```

Figure 4.2: Example of Xtext grammar description

Xtext creates one AST class for each grammar rule. The calls to sub-rules become attributes. Calls to terminal rules (also called lexer rules) become attributes that have primitive types (string, integer). Using annotations, it is possible to change name of the attributes and return type of the rule. If the developer wishes to have more control over the generated classes (for example, to add an attribute for use in AST processing), it is possible to write a script in model transformation language Xtend that processes the automatically generated metamodel and adds the necessary attributes or modifies the class hierarchy. However, developer must be careful because Xtext does not check that the modified model is internally consistent and type correct with respect to the parser.

Xtext also supports run-time customization of the AST building process. Similar to ANTLR, it is possible to add actions to grammar rules. These

actions operate on the AST node currently being built by the rule[1].

Xtext supports grammar inclusion. This provides support for some cases of language extension and customization. It is possible to create an extensible DSL by splitting the grammar description into several smaller files and then including the rules that are needed for a given DSL. Thus, Xtext partially satisfies the requirement CUSTOM.

Xtext automatically generates AST class model from the grammar description and stubs for the editor services. Thus, implementing simple DSLs with code generators does not require extra effort and thus Xtext satisfies requirement SMALL.

Program checking Xtext has built-in support for name resolving. In grammar description, the developer can indicate that an identifier points to an object of the given class. In the grammar example (figure 4.2), the rule *BeanRef* contains attribute target that references objects of type *Bean*. By default, the reference name must match value of the attribute *name* in the target object, but this can be changed by the developer. When parsing BeanRef rules, Xtext automatically checks that the program contains a bean with the given name. If the bean is found, the referrer is then set to directly point to the target (this is, the reference attribute is typed as the target class and the attribute points to target object instead of the identifier string). The language developer can control the link resolving process by implementing a scope provider service that can constrain the set of objects visible at given point of program. Apart from link resolving, Xtext requires the developer to write Java code that verifies the AST of the DSL program.

Program transformation Xtext does not offer direct support for program transformation. The DSL developer can either implement the transformation in Java or he can use EMF-based transformation tools (such as Eclipse M2M [58]) to transform the EMF models. Xtext satisfies the requirement LANG.

Code generation Xtext contains Xpand[2] template engine that can be used to generate code from parsed AST. Figure 4.3 shows example of Xpand

[1]However, the use of these actions seem to be undocumented and the examples imply that they are mostly used for when parsing left-associative arithmetical expressions where the used parsing algorithm does not support left-recursive grammars.

[2]See `http://www.eclipse.org/modeling/m2t/?project=xpand` for more information

template.

```
«IMPORT beanDsl»

«DEFINE Generator FOR Model»
«FOREACH beans AS bean»
«FILE bean.name + ".java"»
«EXPAND beanClass FOR bean»
«ENDFILE»
«ENDFOREACH»
«ENDDEFINE»

«DEFINE beanClass FOR Bean»
public class «name» {
    «EXPAND attribute FOREACH attrs»
}
«ENDDEFINE»

«DEFINE attribute FOR Attr»
«type.metaType == BeanRef
    ? ((BeanRef) type).target.name
    : this.toString()» «name»;
«ENDDEFINE»
```

Figure 4.3: Example of code generation using Xtend

Program editor Xtext provides full-featured Eclipse-based IDE for the DSL. The developer can create IDE that supports code coloring, outline view, code hyperlinking, code completion, and occurrence marking. Some features (such as code coloring and code completion for keywords) is available immediately. For advanced features or non-standard behavior (e.g. for coloring references differently depending on the referenced object), the developer must implement various Java-based interfaces.

4.2.2 EMFText

EMFText [31, 16] is an EMF-based language toolkit, similar to Xtext. The main difference is that when Xtext uses the grammar description to generate EMF metamodel, the EMFText starts with metamodel of the target language and uses it to generate grammar description.

Parser The language developer starts by creating a metamodel for the target language. Using this metamodel, EMFText generates initial version

of the syntax description. The developer can then modify the syntax description to achieve the desired language syntax. Like Xtext, EMFText uses ANTLR as backend parser generator.

EMFText has support for language evolution. It is possible to independently evolve the language metamodel and the syntax description. The system detects when the two become out of sync and raises error. However, the developer must separately maintain the AST description and concrete syntax description. For small language this introduces code duplication and thus EMFText does not satisfy requirement SMALL. EMFText can be used to create modular and customizable DSLs, thus it satisfies requirement CUSTOM.

Program checker EMFText implements automatic reference resolving mechanism, similar to Xtext. For other validation tasks, the developer has the option of using either EMF validation framework [84] or by registering Java-based post-processors for the processing queue.

Program transformation EMFText does not have direct support for program transformation. The developer can either use EMF transformation tools or code the transformations in Java (satisfies the requirement LANG).

Code generator EMFText does not directly include code generation functionality. However, the developer can use other EMF-based code generators, such as JET[3], Acceleo[4] or Xpand (see Section 4.2.1). Additionally, it is possible to use JaMoPP [32] that implements a meta-model for the Java programming language and pretty-printer that transforms a program from Java model to text. Thus, it is possible to generate Java code by using model-to-model transformation to transform the DSL model into a Java model and then use the pretty-printer to generate Java code.

Program editor EMFText produces an Eclipse-based IDE, similar to Xtext.

[3]See `http://www.eclipse.org/modeling/m2t/?project=jet` for more information
[4]See `http://www.eclipse.org/acceleo/` for more information

4.3 Language Workbenches

Language workbench is a term created by Martin Fowler [17, 19] (although the relevant tools existed before that time). Although the term does not have precise definition, the general consensus is that language workbenches are characterized by the following properties.

- They place significant amount of emphasis on defining an abstract representation of the DSL program, called *semantic model*. Usually the semantic model is defined before concrete syntax of the DSL.
- They use *projectional editing*. The programmer, instead of editing textual DSL program that is later parsed to AST, directly manipulates the semantic model of the DSL program [18]. The representation of the semantic model can be text (structured text, to be precise) or graphics. In some tools, it is possible to define custom editors (widgets) for particular AST nodes.
- Semantics of the DSL is usually defined by implementation. Typically there are means to build a code generator that walks over the AST and generates either structured or unstructured code.

The difference between language workbenches and EMF-based tools (see Section 4.2) is not very well defined. On the one hand, EMF-based tools also pay great attention to AST and express it using modeling terms (classes, attributes, etc.). Also, the model semantics is defined by code generation. On the other hand, the EMF-based tools listed in Section 4.2) assume that the DSL programs use textual syntax and thus implement parsers (as opposed to language workbenches that generally do not define concrete textual syntax for the DSLs and instead provide means to manipulate the semantic model directly). Thus, this section only reviews tools that do not have means to define concrete textual syntax and use projectional editing instead.

When talking about implementation, all the reviewed language workbenches have tightly integrated architecture. The semantic model is integrated with the graphical editor and often the same applies to the code generator. This means that language workbenches do not fulfill the requirements SEP_WP and SYSTEM. Also, since there is no authoritative textual syntax for the DSL, the DSL program is saved as serialized semantic model, often using XML format. Therefore the DSL programs are not saved as human-readable (and mergeable) files and the VCS requirement is not fulfilled.

4.3.1 Meta Programming System (MPS)

Meta Programming System (MPS) [14, 93, 61] is an open source language workbench developed by JetBrains Inc. MPS has been used to create several commercial products, such as the YouTrack bug-tracking system[5].

Parser and program editor MPS does not have parser in a conventional sense, it uses projectional editing instead. In MPS, language development starts with specifying semantic model that is equivalent to AST of the text-based languages. Based on the model, it is possible to create a program editor. By default, the MPS creates cell-based editor that allows the developer to lay out the program elements either in rows or columns. The cells can be nested, building up an hierarchical editor. Figure 4.4 shows semantic the model and the corresponding editor for the Bean concept. Figure 4.5 shows a screenshot of an editor created with MPS. In addition to standard structured text editor, MPS allows the developer to create customized editor widgets (programmed as Swing components) for some AST nodes.

Because of the nature of the projectional editor, it is impossible to create incorrectly structured DSL program. However, it is possible to create a program that is incomplete (all the required nodes and attributes are not filled in). One advantage of text-based editing is the wide range of editing operations available. For example, while refactoring a program, the developer can move code snippets around without having to ensure that all the intermediate steps result in syntactically correct programs. When using MPS, this freedom is restricted and it is more difficult to change, for example, *for*-statement to *while*-statement. However, MPS allows the language developer to create support for some common refactoring operations. For example, to replace identifier expression "x" with binary operator "$x + \square$" (where the $\square$ represents empty input cell) without recreating the whole branch of the AST.

Program checker MPS has explicit support for creating type systems. The language developer writes *type equations* that can assign types to AST nodes (for example, all integer literals are typed *int*), propagate the type information along the AST, and check whether types of two AST nodes are compatible. In addition to type checkers, the developer can attach constraint checkers to AST nodes. The checkers are written in a language

[5] See `http://www.jetbrains.com/youtrack/` for more information

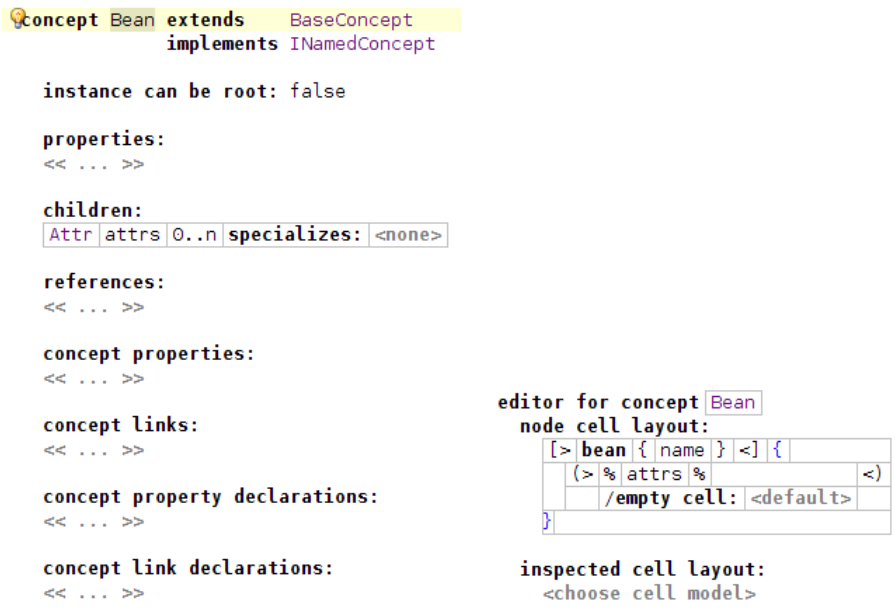


Figure 4.4: Semantic model (a) and editor (b) for the Bean concept

called "base". The base language is a Java-like language that has additional language constructs targeted at AST processing (thus, it has partial support for requirement LANG).

Program transformation MPS contains a template engine that can be used for model transformation. The projectional editing paradigm is also used for templates, thus guaranteeing that the template (and also the generated model) will be structurally correct. Figure 4.6 shows template for converting Bean DSL model to Java model.

Code generator For simple code generation tasks, MPS uses TextGen, an instance of the program transformation engine that can output text instead of models. MPS also contains model of the Java programming language. Therefore, if code generation target is Java, then it is possible to first transform the DSL AST to Java AST and then use the built-in Java code generator. This approach is preferable because it guarantees that the generated Java code will be structurally correct.

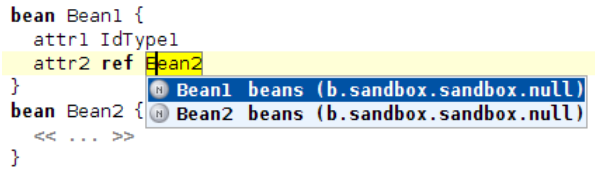


Figure 4.5: Editor created with the MPS language workbench

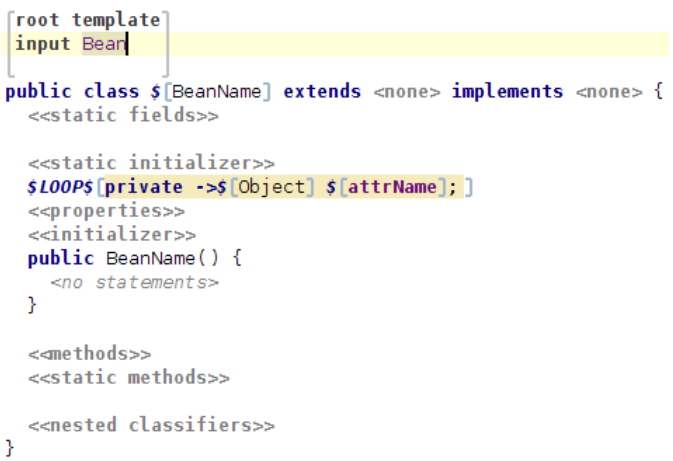


Figure 4.6: Model transformation with MPS

Non-functional requirements MPS is built on IntelliJ[6] platform and tightly integrated with it. The DSL implementation cannot be separated from the IntelliJ IDE, even building the DSL implementation requires IntelliJ and MPS. Therefore, MPS does not satisfy requirements BUILD and SYSTEM. However, the DSL implementation created with MPS can be deployed as separate IntelliJ plugin, thus satisfying requirement SEP_WB.

MPS stores the DSL programs in the model form. By default, XML files are used for storage. However, the developer is able to implement custom persistence mechanism to store the models e.g., in a database. MPS contains tool to merge two model files. Therefore, it is possible to store MPS models in traditional text-based version control system, such as Subversion or git. The VCS just needs to be configured to use MPS-specific tool to merge changes entered by different developers. The requirement VCS is satisfied.

MPS provides excellent support for creating modular and extensible DSLs. It is possible to extend a DSL with new constructs or to combine several

[6]See `http://www.jetbrains.com/idea/` for more information

DSLs. The MPS therefore satisfies requirement CUSTOM. For simple DSLs, the developer needs to define AST, create cell-based editor, and implement a code generator. Thus, MPS satisfies requirement SMALL.

4.3.2 Intentional Workbench

Intentional Workbench [78, 36] is a language workbench developed by the Intentional Corporation. Although initial versions of the tool were demonstrated in 2009 [9], there is very little publicly available information about the product. From the available material, one can conclude that the Intentional Workbench operates in a manner similar to the MPS workbench (see Section 4.3.1). The DSL program is stored in a tree form and projectional editing is used to access the tree. Intentional Workbench allows the developer to define multiple projections for a single DSL. The program can be edited via any projection. In addition to projections, the developer can define read-only views that can display e.g., call graph of the DSL program. Although there is little information about architecture of the Intentional Workbench, it seems that it is similar to the MPS as far as the non-functional requirements are concerned.

4.3.3 OOMEGA

OOMEGA [65] is mainly a model-driven engineering (MDE) platform. It puts strong focus on managing object persistence and relies on standard MDE technologies, such as ATLAS transformation language (ATL) [40].

Parser and program editor As with other language workbenches, the language development starts with specifying a metamodel (AST classes). This is done via graphical editor or by using special-purpose language called M2L. From the metamodel, it is possible to generate initial version of concrete syntax description (also expressed in M2L). The developer can then modify the concrete syntax description to create the desired grammar.

The program editor's user interface is designed to give a look and feel of conventional text editor, but in practice it is a projectional editor that directly changes the AST stored in the memory.

Program checker OOMEGA implements automatic link resolving, similar Xtext and MPS. Additionally, the developer can include constraints

in the metamodel description. The constraints are expressed in language called Edge Algebra.

Program transformation OOMEGA itself does not include program transformation components. Instead, the developer is expected to use Eclipse ATL language for program transformation tasks.

Code generator OOMEGA includes code generator that is based on JSP. In addition, the developer is free to use other Eclipse-based code generators, such as Xpand.

Non-functional requirements OOMEGA uses an object database as its storage back-end. The database can be queried with object-oriented query language. The store is accessed via an well-defined API, thus allowing for different implementations. Currently the main implementations are MemoryDB and relational database. MemoryDB is an in-memory database, the model is maintained in data files that are only loaded into memory for processing. The authors recommend using XML or other structured file format for storing models, as the OOMEGA parser is not optimized for reading large files that use the DSL syntax. The alternative storage back-end uses relational databases. This enables multiple users to simultaneously access the models. In short, OOMEGA supports the use of VCS-controlled files in principle, but is not optimized for them. Therefore, it partially satisfies the requirement VCS.

The program editor for DSL created with OOMEGA is distributed as Eclipse plugin that can be deployed separately from the main language workbench. Therefore, OOMEGA satisfies requirement SEP_WB. The language implementation itself is tightly integrated with Eclipse and cannot be deployed or built separately from Eclipse IDE (requirements SYSTEM and BUILD are not satisfied). OOMEGA can be used to develop modular DSLs that also interact with each other (for example, program in one language can refer to objects defined using some other language). Requirement CUSTOM is satisfied. OOMEGA uses existing languages for program transformation and thus satisfies the requirement LANG. In OOMEGA , the developer must separately describe the metamodel and abstract syntax and ensure that they are synchronized. This additional work makes OOMEGA unsuitable for implementing small DSLs (does not satisfy the requirement SMALL).

4.4 Spoofax/IMP

Spoofax/IMP [43, 44, 81] is a language development environment that is based on the components of the Stratego/XT toolkit.

Parser Spoofax/IMP uses the SDF [30] language for defining concrete syntax of the languages. Unlike many popular parser generators, SDF does not have a separate lexer and parser. Instead, all of the syntax is defined via production rules. On the one hand, this makes the formalism more flexible allowing it to concisely express grammars for wide range of languages. On the other hand, the generality comes at a small price. For example, for most languages the developer needs to explicitly list all the keywords to ensure that they are not parsed as identifiers.

The grammar description is also used to generate data types for the AST nodes. Spoofax/IMP uses ATerm [4] for storing the AST of the programs.

Program checking and transformation Spoofax/IMP uses Stratego [92, 6] language for language processing. In Stratego, the transformations are expressed in terms of rewriting rules that can be controlled by programmable strategies. Figure 4.7 shows how reference resolving in the bean language can be implemented in Stratego. First, we define strategy *analyze* that collects the list of all the beans in the program. It does that by matching the input as *Program* AST node and calling the rule *record-bean* on all the beans in the program. The rule *record-bean* matches the bean and creates a dynamic rule *GetBean* for resolving the bean name. Next, we define strategy *resolve-links* that walks through the AST using top-down strategy, and invokes the rule *resolve-bean-ref* for every node. The rule matches attribute definitions and replaces the "plain" type annotations *Type(typeName)* with references to beans *BeanRef(typeName)*, if the *typeName* matches name of a bean. To put all together, the combined strategies can be invoked by using sequence combinator: `analyze; resolve-links`.

Code generator The idiomatic way to generate code in Spoofax/IMP is be to transform the DSL AST into AST of the target language and then pretty-print the target. Spoofax/IMP includes support for several languages, including Java. It is possible to use concrete syntax inside transformation rules (see Figure 4.8 shows rule *generate-bean* that generates Java code for a bean. It makes use of concrete syntax feature of the Stratego

```
strategies
    analyze =
        where(?Program(<map(record-bean)>))

    resolve-links = topdown(try(resolve-bean-ref))

rules
    record-bean:
        Bean(name, attrs) -> Bean(name, attrs)
        where
            rules(
                GetBean :+ name -> name
            )

    resolve-bean-ref:
        Attr(name, Type(typeName)) -> Attr(name, BeanRef(typeName))
        where
            <GetBean> typeName
```

Figure 4.7: Stratego example: reference resolving

language: the code between "|[" and "]|" is parsed to Java AST internally. There are "unquote" sequences, beginning with tilde (~) that can be used to insert other nodes into the generated AST.

```
generate-bean:
    Bean(name, attrs) ->
        compilation-unit |[public class ~x:name {
            ~*fields
        }]|
    where
        fields := <map(generate-field)> attrs
generate-field:
    Attr(name, Type(type)) ->
        class-body-dec* |[private ~x:type ~x:name;]|
generate-field:
    Attr(name, BeanRef(type)) ->
        class-body-dec* |[private ~x:type ~x:name;]|
```

Figure 4.8: Example of code generation with Spoofax

Program editor Spoofax/IMP creates an Eclipse-based IDE that is implemented using IMP IDE toolkit (see Section 4.5.1). The IDE is configured with DSLs that can call strategies written in the Stratego language. The

IDE provides all the usual features, such as syntax highlighting, outline view and automatic completion. Unlike many other Eclipse-based tools, Spoofax/IMP allows testing the language IDE without starting another copy of Eclipse.

Non-functional requirements The program editor can be distributed as separate Eclipse plugin (satisfies SEP_WB). Spoofax/IMP uses Java-based implementations of Stratego and SDF, therefore it is possible to call the DSL implementation either from command-line (requirement BUILD) or to embed the DSL implementation into a bigger system (requirement SYSTEM).

SDF has good support for creating modular and extensible languages. It is possible to split the grammar description into several files and combine the files to produce different languages. The language processing can also be modularized (requirement CUSTOM). The DSL programs are stored as text files that can be managed using standard VCS (requirement VCS).

For small projects, Spoofax/IMP does not introduce additional overhead, therefore it satisfies the requirement SMALL. However, it uses Stratego language that has is quite dissimilar from typical programming languages, thus incurring a steep learning curve[7] (does not satisfy requirement LANG).

4.5 Special-Purpose Tools

This section covers tools that are mostly target one particular aspect (e.g., parsing, program checking, IDE creation) of creating a DSL implementation. Since the tools are not comprehensive toolkits, this section will not use the feature-by-feature overview format used in the previous sections. Instead, we will only describe the focus area of the particular tool.

4.5.1 IMP

The IDE Meta-tooling Platform (IMP) [8, 35] is an IDE building framework. It is based on the Eclipse platform and aims to hide the technical details of creating an Eclipse-based IDE. The API provided by IMP focuses on AST of the DSL program and the concept of language services. In order to provide

[7] In [91], Vasudevan and Tratt rank the Stratego implementation as having largest number of aspects to learn.

IDE features, the DSL developer must implement language services. For example, hyperlinking and tooltip features use reference resolving service that takes as input an AST node, determines whether it is a reference (such as identifier), and returns the referenced AST node.

IMP can be used with any parser generator or hand-written parser. However, it comes bundled with LPG parser generator in order to help the developer to get quickly started. IMP does not impose restrictions on non-visual part of the DSL implementation and thus satisfies all the non-functional requirements. Because of its flexibility, IMP is used as a component in DSL toolkits, such as Spoofax/IMP. IMP provides APIs that can be implemented using any JVM-based language, thus it satisfies the requirement LANG.

4.5.2 Parser Generators

4.5.2.1 ANTLR

ANTLR [67, 2] is primarily a parser generator. It takes as input a description of a DSL's syntax and produces a parser that recognizes the DSL. By adding actions to the parser, it is possible to build a translator or an interpreter. ANTLR offers limited support in the form of tree grammars, for processing the AST. ANTLR can be used together with StringTemplate [68] template engine to build a simple DSL implementation consisting of a parser and a code generator (requirement SMALL). ANTLR can generate parsers in several languages, such as Java or C/C++. Thus, using ANTLR generally does not involve adding new languages to the project (requirement LANG).

ANTLR does not have any dependencies that prevent embedding it in a larger system and therefore it satisfies most of the non-functional requirements. However, ANTLR's support for customizable languages (requirement CUSTOM) is quite limited. There is a mechanism for grammar inheritance, but this only supports some specific cases of reuse and parametrization.

4.5.3 Attribute Grammar Systems

Attribute grammars [50] are a way to define computations on the AST of the program. Attribute grammar systems generally allow the developer to

describe the calculation of the attributes in a declarative manner and encapsulate the technicalities of traversing the AST and invoking the calculations in a correct and efficient manner.

This section lists attribute grammar systems that are mostly focused on program checking and program transformation functionality. They are based on Java and generate Java-based DSL implementations. Thus, DSL implementations created using these tools are embeddable and satisfy requirements BUILD and SYSTEM.

4.5.3.1 JastAdd

JastAdd [29, 28, 37] is a meta-compilation system that supports Rewritable Circular Reference Attribute Grammars (ReCRAG). It aims to be easy to learn for Java developers, therefore using object-oriented approach and Java-like syntax (satisfies requirement LANG).

In addition to the basic inherited and synthesized attributes, JastAdd also supports attributes that reference other AST nodes. This allows performing several tasks, such as name resolution and type checking in-place, without creating a separate environment mapping names to their definitions. In addition, JastAdd supports parameterized attributes, broadcasting, nonterminal attributes, and AST rewriting. Figure 4.9 shows an example JastAdd grammar that checks the references to other beans. First, code in Figure 4.9a defines the abstract syntax of the bean language. Next, code in Figure 4.9b defines the attributes responsible for reference resolving. The main entry point is the attribute *refTarget*, defined on *Attr* class, that returns the bean object that is named in the attribute type (or *null*, if the type does not contain name of a bean). The bulk of the work is done by the inherited attribute *lookup* that is defined at the program level and propagated down to the *Attr* class. At the bean level, the attribute *lookup* uses the bean-level synthetic attribute *localLookup* that checks whether the argument matches the name of the bean.

JastAdd supports both modular languages and modular implementations (requirement CUSTOM). If JastAdd is integrated with parser generator so that parser directly produces AST that is compatible with JastAdd, then there is no repetition and thus JastAdd satisfies requirement SMALL.

```
Program ::= Bean*;
Bean ::= <Name:String> Attr*;
Attr ::= <Name:String> <Type:String>;
```

(a) Definition of abstract syntax

```
aspect RefTarget {
    syn Bean Attr.refTarget() = lookup(getType());
    inh Bean Bean.lookup(String name);
    inh Bean Attr.lookup(String name);

    eq Program.getBean(int i).lookup(String name) {
        for (Bean b : getBeanList()) {
            Bean match = b.localLookup(name);
            if (match != null)  {
                return match;
            }
        }
        return null;
    }

    syn Bean Bean.localLookup(String name) =
        name.equals(getName()) ? this : null;
}
```

(b) Implementing attributes for name lookup

Figure 4.9: JastAdd example: resolving reference in the bean language

4.5.3.2 Kiama

Kiama [79, 80, 47] is a Scala library for language processing. It is implemented as an embedded DSL. Kiama provides support for attribute grammars, tree rewriting, abstract state machines, and pretty-printing. All the tasks assume that the AST is expressed as Scala case classes. Kiama itself does not have functionality for parsing the program text and building the AST. For this purpose, the developer must use either a parser generator or a parser library.

In Kiama, attributes are expressed as Scala functions. The framework is responsible for maintaining dependencies between attributes and lazily evaluating and caching attribute values. Figure 4.10 shows reference resolving example in the bean language. The first four lines define the case classes

that represent the AST. Next, the attribute *refTarget* applies to *Attr* nodes and returns reference to a bean referred to by *attrType*. The value of the attribute can be accessed as `refTarget(attrNode)` where *attrNode* is a reference to bean attribute. In order to resolve the type name, *refTarget* calls the parameterized attribute *lookup*. The attribute *lookup* finds the bean with the given name and propagates this information down the tree.

```
abstract class Tree extends Attributable
case class Program(beans: List[Bean]) extends Tree
case class Bean(name: String, attrs: List[Attr]) extends Tree
case class Attr(name: String, attrType: String) extends Tree

val refTarget: Attr => Option[Bean] =
    attr {
        case node @ Attr(_, attrType) => lookup(attrType)(node)
    }

def lookup(name: String): Tree => Option[Bean] =
    attr {
        case Program(beans) =>
            beans.find(_.name == name)
        case node =>
            lookup(name)(node.parent.asInstanceOf[Tree])
    }
```

Figure 4.10: Example: name resolution using Kiama attributes

Kiama's tree rewriting library is influenced by Stratego language (see Section 4.4). The rewriting rules are implemented as functions that take as argument tree nodes and return modified tree nodes. The strategies are also expressed as functions. Kiama contains combinator library that implements most of the strategy combinators found in the Stratego language.

Kiama also contains library for implementing state machines (can be used for prototyping languages by converting the program to a state machine) and pretty-printing library based on Swierstra's algorithm [86].

Since Scala is a JVM language, Kiama is easily embeddable in Java-based systems (requirements BUILD and SYSTEM). Due to Scala's flexibility, it can also be used to create modular languages and modular implementations (requirement CUSTOM). Since Kiama is based on Scala, it satisfies the requirement LANG. Kiama itself does not require any code duplication (satisfies SMALL). However, when using Kiama with Scala's parser combinator library, one must specify the AST classes separately from the grammar (that also contains information about the AST).

4.5.3.3 Silver

Silver [89, 77] is an attribute grammar language that aims to support modular language specifications. Besides the usual inherited and synthesized attributes, Silver also supports other features. For example, forwarding can be used to extend the language with new constructs that can be expressed in terms of the original language. Higher-order attributes allow constructing parts of the AST as attribute. Reference attributes can refer to other AST nodes and can be used for name resolving.

Silver comes with Copper [90] parser generator. Like Silver, Copper is suited for modular specifications (requirement CUSTOM). It produces context-aware lexers that make it easier to create extensible languages (the tokenization can depend on whether the parser is currently processing an extension or not). Copper can analyze several grammars to determine whether composing them would result in conflict [75].

The program transformation logic usually operates on abstract syntax of the DSL program. However in Silver there is no automatic transformation from concrete syntax to abstract syntax. Instead, the language developer must explicitly declare the algebraic data type for abstract syntax tree and build the tree in concrete grammar specification (usually the abstract syntax tree is higher order attribute of the concrete syntax tree). On the one hand, this gives additional flexibility to the language developer, as he can use different abstract syntax trees for different purposes. On the other hand, this introduces additional work for the simpler cases (Silver fails the requirement SMALL).

4.6 Summary

This chapter provided a brief overview of DSL toolkits and some single-purpose DSL tools. For single-purpose tools, we reviewed a (hopefully representative) subset of all the existing tools because the tools in the same class (e.g., parser generators) generally have similar properties. Table 4.1 provides summary of the functionality of the reviewed DSL toolkits. Although the single-purpose tools were not evaluated for functional coverage, they are included in the table to show where the tool fits in the functionality scale.

The table shows whether the given tool has explicit support for the task. "No" in a cell does not mean that the given task cannot be accomplished

with the given tool – typically, the tools can be integrated with programs or modules written in general-purpose programming languages that can be used to provide the missing pieces. For example, Xtext scores "No" in the "program transformation" feature meaning that Xtext does not have explicit support for program transformation. However, the developer can perform program transformation tasks by writing Java code or by using EMF-based model transformation tools.

Table 4.1: DSL tools by functionality

Tool	Parsing	Checking	Transformation	Generation	Editor
Language toolkits					
Xtext	Yes	Partial[a]	No	Yes	Yes
EMFText	Yes	Partial[a]	No	No	Yes
MPS	No	Yes	Yes	Yes	Yes
OOmega	No	Yes	No	Yes	Yes
Spoofax/ IMP	Yes	Yes	Yes	Yes	Yes
Single-purpose tools					
IMP	No	No	No	No	Yes
ANTLR	Yes	No	No	Yes	No
JastAdd	No	Yes	Yes	No	No
Kiama	No	Yes	Yes	Yes	No
Silver	Yes[b]	Yes	Yes	No	No

[a] Built-in name resolution, otherwise requires writing Java code.

[b] Copper parser generator.

From the table, it can be seen that Spoofax/IMP and MPS aim to be comprehensive toolkits offering support for all the tasks required to create a full DSL implementation. The other toolkits (Xtext, EMFText, OOmega) can also be used for DSL creation. However, they have chosen to offer limited support for program checking and program transformation tasks, assuming instead that the developer will use a general-purpose programming language for implementing these tasks. As expected, the single-purpose tools are focused on one or two tasks.

Table 4.2 summarizes how well the reviewed tools satisfy the non-functional requirements.

Comparing the tables 4.1 and 4.2, it can be seen that there exists an inverse correlation between functionality and embeddability of DSL tools. While

Table 4.2: DSL tools by non-functional requirements

Tool	LANG	SMALL	SEP_WB	BUILD	SYSTEM	VCS	CUSTOM
Language toolkits							
Xtext	Yes	Yes	Yes	Yes	No	Yes	Partial
EMFText	Yes	No	Yes	No	No	Yes	Yes
MPS	Partial[a]	Yes	Yes	No	No	Yes	Yes
OOmega	Yes	No	Yes	No	No	Partial[b]	Yes
Spoofax/ IMP	No	Yes	Yes	Yes	Partial[c]	Yes	Yes
Single-purpose tools							
IMP	Yes	N/A	Yes	N/A	N/A	N/A	N/A
ANTLR	Yes	Yes	N/A	Yes	Yes	Yes	Partial
JastAdd	Yes	Yes[d]	N/A	Yes	Yes	Yes	Yes
Kiama	Yes	Yes	N/A	Yes	Yes	Yes	Yes
Silver	No	No	N/A	Yes	Yes	Yes	Yes

[a] The MPS base language is based on the Java language, but many sub-languages (e.g., for defining type equations) have only passing resemblance to Java.

[b] Managing DSL programs in VCS is possible, but it requires additional work by the DSL developer.

[c] Java implementation of Stratego language has significantly lower performance than the native implementation.

[d] This assumes that the parser produces AST in the format that can be directly used by JastAdd.

DSL toolkits offer support for most of the DSL creation tasks, it can be difficult to apply them in the enterprise software development. The toolkit parts are highly integrated, in particular the non-visual parts (parser, code generator, program checker) tend to be integrated with the IDE framework. The exception here is Spoofax/IMP that is mostly built around non-visual Stratego language, and thus not so tightly integrated[8]. On the one hand, this approach makes the whole tool simpler and more powerful because the DSL program is internally stored and processed in a format that is well-suited for implementing the IDE services. Additionally, the EMF-based tools Xtext and EMFText can be used in conjunction with other tools that share the EMF infrastructure. On the other hand, this architectural

[8]However, the Spoofax/IMP fails the requirement LANG, which means that it may be difficult to employ in enterprise setting where the developers are not interested in learning a new language in order to develop a DSL.

style makes it difficult to use the toolkit in combination with other tools or outside the original IDE environment.

The special-purpose DSL tools represent the other side of the coin. They are specifically designed so that they can be used in combination with other tools and that they produce DSL implementations with minimal amount of dependencies. However, in order to build a complete DSL implementation, the developer must either create bulk of the implementation (the part of the functionality that is not covered by the tool) by hand, or he can integrate different tools, which again may require substantial amount of manual coding.

Thus, the enterprise developer has the choice of using either a DSL toolkit that is easy to use, but difficult to combine with other tools and to embed into the system, or a single-purpose DSL tool that can be integrated into the enterprise system. This is unfortunate because technically there are no strong reasons why the two aspects cannot be combined. Ideally, the enterprise developers would benefit from conveniently packaged end-to-end solution that would cover all the aspects of DSL development and that would also be sufficiently modular and flexible to allow integrating the DSL tool and/or DSL implementations created with the tool into different systems. This toolkit should lower the barrier of entry by taking advantage of existing tools and programming languages to leverage the skills of the developers. It is the opinion of the author that the use of toolkit matching these goals would lower the cost of creating a DSL and make the DSL-oriented software development more practical in the enterprise.

CHAPTER 5

DESCRIPTION OF SIMPL

5.1 Introduction

This chapter introduces Simpl, a DSL toolkit aimed at creating embeddable DSL implementations. It is based on practical experiences gained during the Customs Engine project (see Chapter 2) and other projects. Simpl was developed at Cybernetica AS, the author is the architect of the Simpl toolkit and contributed to implementations of all of the components. Simpl aims to fulfill the requirements described in Chapter 3) while offering simple and usable tools for developers. Accordingly, in addition to the requirements listed in Chapter 3, Simpl aims at fulfilling the following design goals.

- First and most importantly, the focus of Simpl is to provide ease of use and a low learning curve. In particular, Simpl must be easy to learn for developers who are not specialists in language development.
- Contemporary software development practices (both for DSL creators and DSL users) involve intensive use of IDEs. Therefore, it must be possible to provide reasonable IDE for a simple DSL with little or no programming effort. Creating full-featured IDEs for a more complex DSLs must also be possible.
- If reasonable, the language for writing program checking and program transformation code should be an existing language. In practice, it is very difficult to convince developers to use a new tool if it requires learning a new programming language. Additionally, in order to compete with existing languages, the new language would have to have comparable level of tooling, such as IDEs, debuggers, etc.

- It must be possible to create simple DSLs with very little effort. Thus, Simpl must be usable for creating "configuration DSLs" described in the case study (Section 2.2.2). Simpl aims to minimize the amount of code that needs to be written for simple DSLs.
- The DSL processing language should be statically checked to ensure that simple programming errors are detected early.
- It must be possible to create complex languages. In particular, it must be possible to use Simpl for implementing compiler for a typical functional or imperative language.

In order to better use the available development efforts, we have decided to base Simpl on existing tools. Our contribution is a unified, integrated interface to the whole tool chain and improvements targeted at usability and productivity. In particular, Simpl is based on the ANTLR parser generator, the Scala programming language, and the IMP IDE toolkit. Additionally, Simpl can be used in conjunction with the Kiama language processing library that can be used for writing program checking or program transformation code.

Simpl builds on the aforementioned components and adds to them additional features that simplify the implementation of DSLs.

- Simpl automatically generates Scala classes for representing the abstract syntax tree (AST) of the parsed DSL program. The class model corresponds to the grammar of the DSL. The developer can influence the generation process by adding annotations to the grammar description. Additionally, the developer can use return expressions (see Section 5.3.4) to customize the AST nodes returned by the grammar rules at runtime.
- Simpl parser generator adds support for lexer states: a way to add context sensitivity to the lexical analysis process[1]. In some tools (e.g., in ANTLR) the same effect can be achieved by using free-form semantic predicates. Compared to semantic predicates, lexer states in Simpl offer a more structured approach is simpler to use and to analyze.

[1]Lexical analysis typically involves processing the source program using regular expressions. Lexer states help the developer to guide the lexical analysis by using contextual information. For example, this can be used decide whether a sequence of characters should be tokenized either as a keyword or as an identifier.

- Simpl includes pretty-printing library that can be used to reformat existing DSL programs or to implement a code generator that outputs well-formatted code.
- Simpl contains an IDE framework that provides API that focuses on the language services and abstracts away the particulars of creating an Eclipse plugin.
- Simpl contains an integration layer that ties the used components into a seamless whole. It aims to minimize the amount of code that needs to be written by offering high-level APIs, framework that handles more technical aspects of implementing a code generator, and wizards that simplify creating a new DSL project.

5.2 Overall Architecture

In the introduction we mentioned that Simpl is based on the existing tools. Next, we describe these tools in more detail and evaluate their suitability as building blocks of the Simpl tool.

An important design decision was selecting the language for writing program transformations and IDE services. We chose the **Scala** programming language based on the following main reasons.

- Scala has functional programming features and support for pattern matching, thus making it very suitable for language processing tasks. Implementing AST processing functionality in a popular enterprise language such as Java is possible, but the result is cumbersome and difficult to understand. The low abstraction level of Java and its lack of support for first-class functions makes it difficult to encapsulate repeating patterns of code into a reusable language library.
- Scala compiles to Java bytecode and interfaces well with Java-based infrastructure and tools. Thus we get best of both worlds – broad applicability of the JVM platform and high-level features of the Scala programming language.
- Scala supports several programming styles (functional or object oriented) and is applicable for wide variety of problem domains.

- Scala has similarities to popular enterprise programming languages, such as Java and C# and thus is easy to learn for enterprise developers. Additionally, Scala is reasonably popular[2] and there is sizable body of information available for it.

The main reasons for choosing **ANTLR** as the back-end to the Simpl parser generator were its maturity and popularity meaning that it is relatively bug-free and well-documented. Additionally, there is substantial body of information on implementing parsers with ANTLR. Feature-wise, ANTLR has good error recovery that is useful for implementing IDEs where user's editing actions often create syntactically incorrect programs. Additionally, parsers implemented with ANTLR produce user-friendly error messages when encountering syntax errors.

For the IDE platform, we chose **Eclipse** that is the most popular Java IDE (see e.g., [99] for a report on popularity of Java IDEs) based on the assumption that it has the best chances of seamlessly integrating into the developers' workspace. On top of Eclipse we use **IMP** that abstracts away some of the complexity associated with building a language editor on top of Eclipse. IMP offers an abstraction layer that does not assume the use of any particular type of parser or AST representation.

Figure 5.1 shows the components of the Simpl toolkit and their connections with the base tools (shown with the gray background). The arrows indicate dependencies between the components. Simpl consists of a parser generator (see Section 5.3); parsing and AST library containing the base classes and common functionality for invoking the parser and manipulating the AST; a generator library that simplifies writing of simple code generators (see Section 5.5); a pretty-printing library (see Section 5.5); and an IDE framework (see Section 5.6).

In order to produce embeddable DSL implementations, Simpl keeps clear separation between the "visual" and "non-visual" parts if the DSL implementation. The "non-visual" part contains the core of the DSL implementation: parser, program checker, code generator, etc. that can be embedded into a larger system. The other, "visual" part contains an IDE for editing and managing DSL programs. The non-visual part does not make any assumptions on how the system is implemented. In particular, the non-visual part does not have dependencies on the visual part and does not assume that

[2]For example, one study [64] found Scala to be 12th popular language among users of Stack Overflow and GitHub web sites. Another study [99] found Scala to be third popular language (after Java and Groovy) on the JVM platform.

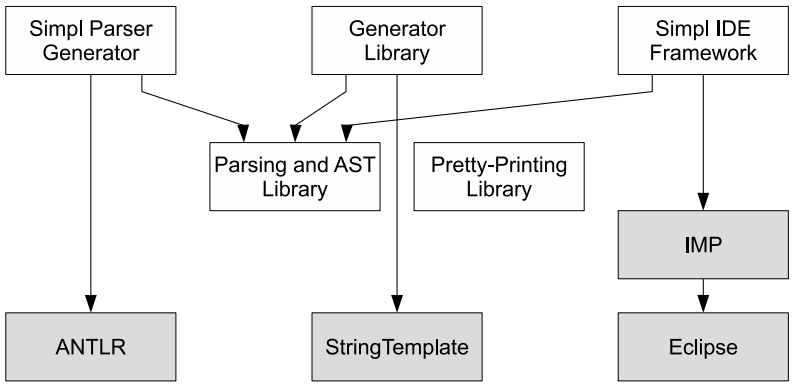


Figure 5.1: Overview of Simpl components

the DSL implementation is a top-level program or function in the system. In addition, it does not use any global data or global caches, making sure that the DSL implementation is thread safe.

Figure 5.2 shows the architecture of a typical DSL implementation created with Simpl. The first part is the non-visual language implementation that can be embedded into a bigger system. Development of a new DSL starts with grammar description that specifies both the context-free grammar of the DSL and the classes for representing the abstract syntax tree (AST) of a DSL program. The Simpl parser generator takes the grammar description as input and produces a parser and the AST classes. The (optional) program transformation component takes as input an AST and checks or transforms it. The code generator converts the preprocessed AST to text. The second part of the DSL implementation is the language IDE. It builds on the Simpl IDE framework and the non-visual part of the language implementation.

The visual part of the DSL implementation consists of an Eclipse plugin. In principle, the IDE part is not freestanding and embeddable. Because creating a full-featured IDE from scratch is not practical (and probably not well-received by developers wishing to spend most of their time in single, all-powerful IDE), we deploy the DSL IDE as Eclipse plugins that can be added to an existing Eclipse installation.

The intended workflow for implementing a DSL with Simpl consists of the following steps.

1. Creating a grammar description. This is used to create a parser that

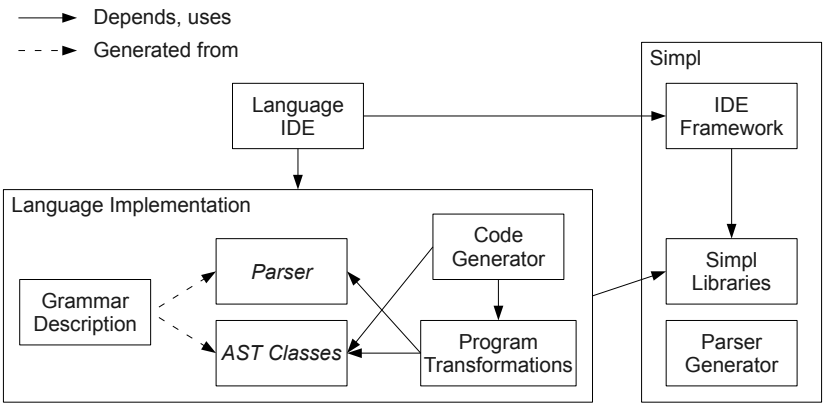


Figure 5.2: Architecture of a DSL implemented with Simpl. Components with captions in italic are automatically generated.

can be run and tested against the sample DSL programs. Based on the unannotated grammar description, Simpl also provides a functional IDE that supports syntax highlighting and error marking.

2. Annotating the grammar description to achieve the desired structure of the AST. The annotations can be used to name the attributes of the AST classes and to modify the class hierarchy.

3. Creating the core of the language implementation. The core is usually a code generator or interpreter, possibly containing a program checker or program transformer.

4. Creating the IDE for the language. Typically, the IDE part is quite thin as it uses the non-visual language implementation for tasks such as program checking, name resolution and code generation.

The following sections describe the Simpl DSL toolkit in more detail. Throughout the sections we will use an example language "SpamDetector" – a DSL expressing spam detection rules, similar to DSL used by SpamAssassin tool[3]. The SpamDetector DSL allows the user to specify conditions that are matched against incoming e-mail messages. If the conditions match, the "spamness" score of the message is incremented or decremented accordingly. The SpamDetector DSL can be considered a typical representative of a moderately complex DSL, containing arithmetic expressions and rudi-

[3]See `http://spamassassin.apache.org/`

mentary user-defined functions (in this case, named conditions that can be used in rules).

Figure 5.3 shows an example SpamDetector program containing two rules. The first rule checks if *Subject* header of incoming message contains case-insensitive string "*viagra*" and if it does, increases the message's "spamness" score by 2.0. The second rule uses named condition *from_mydomain*. It matches all the messages that contain (case-insensitive) string "*business proposal*" in their subject line and is not sent from a "good" domain.

```
rule "Subject contains Viagra" 2.0:
    Subject = /viagra/i

condition from_mydomain: From = /@mydomain.com/
rule "Business proposal" 1.5:
    Subject = /business proposal/i
    and not from_mydomain
```

Figure 5.3: Example rules in SpamDetector DSL.

5.3 Parsing

5.3.1 Basic Grammar Rules

Simpl's grammar language is similar to EBNF notation that is used on most parser generators. Figure 5.4 shows full grammar description for the SpamDetector language. The first line names the package that will host the classes generated for this language and the generated class that will act as the front-end of the parser. The first rule (*Program* in this case) will become start symbol of the grammar. Simpl grammar descriptions can contain four kinds of rules. All the rules start with optional keyword, followed by name of the rule, colon and a pattern that will be matched against input. The names of the rules must be unique – each terminal or non-terminal is completely specified in a single grammar rule.

Terminal (lexer) rules start with keyword *terminal*. The pattern of the rule is a regular expression. Terminal rules produce tokens that are analyzed by context-free rules. Terminal rules can be **hidden** (indicated by keyword *hidden*) meaning that they are ignored by the context-free rules. In the example grammar, hidden terminal *WS* matches white space and comments so that they do not interfere with the other grammar rules.

```
grammar ee.cyber.simplicitas.spamdetector.SD;

Program: Item+;

option Item: Rule | Condition;

Rule: "rule" Name Score ":" Expression;

Condition: "condition" Id ":" Expression;

option Primitive:
      Contains
    | NotContains
    | Count
    | ConditionCall
    | ParenExpression
    | NotExpression;

Expression: AndExpression ("or" AndExpression)*;
AndExpression: Primitive ("and" Primitive)*;

ParenExpression: "(" Expression ")";
NotExpression: "not" Primitive;

Contains: Id "=" Regexp;
NotContains: Id "!=" Regexp;
Count: "count" "(" ExprList ")" "=" Num;
ExprList: Expression ("," ExprList)?;
ConditionCall: Id;

terminal Regexp: "/" (~"/")* "/" "i"?;
terminal Score: "-"? Digit+ "." Digit*;
terminal Num: Digit+;
terminal Name: '"' (~'"')* '"';
fragment Digit: '0'..'9';

terminal Id: IdStart IdNext*;
fragment IdStart: 'a'..'z'|'A'..'Z'|'_';
fragment IdNext: IdStart|'0'..'9'|'-';

fragment MlComment: '/*' (~'*' | '*' ~'/')* '*/';
fragment SlComment: '//' ~('\n'|'\r')*;
hidden terminal WS: (' '|'\t'|'\r'|'\n'|SlComment|MlComment)+;
```

Figure 5.4: Grammar description for the example language

Terminal rules can be structured using **fragment rules** (keyword *fragment*). Fragment rules use the same regular expression syntax as terminal rules. The main difference is that the fragment rules do not create tokens when applied (and therefore cannot be called by the context-free rules). Also, AST classes will not be generated for fragment rules and thus the fragments are not explicitly reflected in the AST. Thus, the main purpose of fragment rules is to split a complicated terminal rule into several named parts.

Non-terminal rules are written without any preceding keyword and describe the context-free syntax of the language. Non-terminal rules can contain literals (in single or double quotes) and calls to other rules. Patterns can be grouped with parentheses. Alternatives are separated by vertical bar. Patterns can be suffixed by symbols to indicate arity: *"?"* means optional, "*" means zero or more, and "+" means one or more.

Option rules start with keyword *option* and consist of alternatives where each alternative is call to another rule. The difference between option and regular non-terminal rules lies in AST generation. See Section 5.3.3 for details.

5.3.2 Advanced Grammar Definition Features

5.3.2.1 Grammar Modularity

Simpl offers basic support for modular grammars. The *import* directive can be used to import all the rules from other grammar into the current one. The rules in the current grammar override the rules from the imported grammar with the same name. Similar logic applies when importing multiple grammars. For example, if grammar *Foo* imports grammars *Bar* and *Baz*, then rules from *Baz* override rules from *Bar* and rules from *Foo* override both *Bar* and *Baz*.

Figure 5.5 shows an example of grammar extension. The SDExtended grammar extends the SpamDetector language and adds a new type of expression. Expression "*random n*" (where $0 \leq n \leq 100$) evaluates to true if newly generated random number from 0 to 100 is less or equal than n (in other words, marks message as a spam $n\%$ of the time). This is accomplished via the following steps. First, we use the *import* keyword to import all the rules from the basic SpamDetector grammar. Next we override the rule *Primitive* to add new option *Random* to list of alternatives. Since the rules are replaced as a whole, it is necessary to repeat all the options

```
grammar ee.cyber.simplicitas.spamdetector.SDExtended;

import "SD.spl";

Program: Item+;

option Primitive:
      Contains
    | NotContains
    | Count
    | ConditionCall
    | ParenExpression
    | NotExpression
    | Random;

Random: "random" Num;
```

Figure 5.5: Example grammar that extends the SpamDetector grammar to add new primitive expression

from previous version of the rule. Finally, since Simpl uses the first rule of the grammar as the start symbol, we have to repeat the rule *Program* as the first rule. Otherwise *Primitive* would become the start symbol since the rules from imported grammars are assumed to come after the rules of importing grammar. Naturally, import statements in grammar definitions should not create cyclic dependencies.

5.3.2.2 Lexer States

Simpl has support for lexer states. They can be used to make the lexer more context-sensitive. In general, the lexer analyzes the input based on regular expressions. Lexer states makes the lexer (terminal and fragment) rules stateful and allows parsing of more complex languages.

Figure 5.6 shows an example grammar that makes use of lexer states. This example extends the basic SpamDetector grammar (cf. Figure 5.4) with *include* directive that imports the rules from another SpamDetector file. The need for lexer states comes from the additional terminal *Filename* used for representing file names. The terminal *Filename* shadows the terminal *Id* and adds additional characters used in file names, such as "/" and ".". Our goal is to restrict use of the *Filename* terminal to the *include* statement so that it does not shadow the terminal *Id* in the body of the program.

The lexer state machine is organized in a stack. Each *enter-state* directive

```
grammar ee.cyber.simplicitas.spamdetector.LexerStates;
                                                                    
// Import the base grammar
import "SD.spl";
                                                                    
// Declare the lexer state
lexer-states (filenameAllowed)
                                                                    
// Add call to Include rule
Program: Include* Item+;
                                                                    
Include: IncludeStart Filename;
                                                                    
// Keyword 'include' triggers state filenameAllowed
terminal IncludeStart
    enter-state(filenameAllowed)
    : 'include';
                                                                    
// Filename is only applicable in state fileNameAllowed and
// also it exits this state
terminal Filename
    check-any(filenameAllowed)
    exit-state(filenameAllowed)
    : ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'-'|'/'|'.')+;
                                                                    
// Identifiers must have lower priority than "include" keyword
terminal Id: IdStart IdNext*;
```

Figure 5.6: Example of using lexer states to resolve conflict between terminals *Filename* and *Id*

takes as argument any number of states and pushes them all to the top of the stack. The corresponding *exit-state* directive pops items from the top of the stack until it encounters the state given as an argument. Thus, if the stack contains states S_1, S_2, S_3, S_4, then after processing directive *exit-state*(S_3), the stack will contain S_1, S_2. Two directives serve as guards that control whether a rule is enabled or not. Directive *check-any*(S_1, ..., S_n) enables the rule if the stack contains any of the states S_1, ..., S_n. Directive *check-all*(S_1, ..., S_n) enables the rule if the stack contains all of the states S_1, ..., S_n in any order. Directive *check-none*(S_1, ..., S_n) works as an opposite to *check-all* – it enables the rule if stack contains none of the states S_1, ..., S_n.

In the example program, we use state *filenameAllowed* to indicate that the terminal *Filename* should be enabled. First, we need to declare the used states with *lexer-states* keyword (line 7). When the lexer encounters the terminal *IncludeStart*, the *enter-state* directive (line 16) pushes the state *filenameAllowed* to the stack. This enables the *Filename* terminal rule. In the *Filename* rule (lines 21-24) we use *check-any* to restrict the rule for a particular state. After processing the *Filename*, the *exit-state* directive pops the *filenameAllowed* state from the stack and disables the *Filename* rule (until next occurrence of the *include* keyword or until the end of the input file). As a technical detail, the *Id* rule must be repeated in the extending grammar so that it comes after the *IncludeStart* rule and will not shadow it.

5.3.3 AST Generation

The grammar generator will automatically generate a Scala case class for every rule in the grammar. The exceptions to this are fragment rules since they do not generate tokens and thus are not reflected in the AST. The generated class will have the same name as the corresponding rule. Each AST class has attributes corresponding to rule references from this rule. If the references are not explicitly named (see later in this section), the name of the attribute will be derived from the type. Figure 5.7a shows example grammar rules and Figure 5.7b classes that are generated for these rules.

The class attributes are generated as modifiable (using the *var* keyword) to make it easier to modify the tree in the later processing steps. If the grammar rule contains several instances of the same sub-rule, the attribute will be typed as list. In the example rules *Rule2* and *Rule3*, the attribute *id* is typed as *List[Id]*. It is possible to override the default attribute names

```
Rule1: Id Str;
Rule2: Id Str Id;
Rule3: Id+ Str;
Rule4: myId=Id myStr=Str;
```

(a)

```
case class Rule1(
    var id: Id,
    var str: Str)
case class Rule2(
    var id: List[Id],
    var str: Str)
case class Rule3(
    var id: List[Id],
    var str: Str)
case class Rule4(
    var myId: Id,
    var myStr: Str)
```

(b)

Figure 5.7: Example grammar rules (a) and the generated AST classes (b)

by specifying them explicitly in grammar. This is demonstrated by the example *Rule4*.

Code generation for option rules is different than for regular non-terminal rules. The body of the option rule can only contain list of alternatives where each alternative is call to another rule. For each option rule Simpl generates Scala trait[4] named after the rule and makes all the rules called by the option rule inherit from this trait. Figure 5.8 shows example rule from the SpamDetector grammar and the corresponding Scala classes. It is also possible to use artificial option rules to create common base classes for several AST classes. For this purpose, one can create option rules that are not called by any other grammar rules. However, these rules still take part of AST generation.

5.3.4 Shaping the Generated AST

Figure 5.9 shows the full SpamDetector grammar with annotations related to the AST generation (highlighted in bold). Some annotations (naming class attributes) were covered in the previous section. This section walks through the rest of the AST shaping features.

All the rules can include code blocks that will be placed inside generated classes. Typically, these code blocks are used to add attributes and methods

[4]In Scala, trait is an abstract class that can be combined with other classes using mixins [63].

```
option Item: Rule | Condition;
```

(a)

```
trait Item
case class Rule(...)
    extends Item
case class Condition(...)
    extends Item
```

(b)

Figure 5.8: Example option rule (a) and the corresponding classes (b)

that are later called from language processing code. Line 42 of the example grammar adds a new attribute to the *Id* class.

The developer can specify the **return type** of the rule using *returns* keyword, followed by identifier. For example, in line 11 of the grammar, return type of the rule *Primitive* is set to be *Expression*. This has the following effects.

1. The generated class *Primitive* is generated as before and will extend class *Expression*.
2. If the grammar does not contain rule named *Expression*, then a new trait is created with that name.
3. All the grammar rules that call the rule Primitive will be modified so that the corresponding attribute will be typed *Expression*. For example, the rule *NotExpression* will generate the following class:

   ```
   case class NotExpression(expr: Expression) extends Expression
   ```

The return type can be used for making the rule return a more general type. An attempt to change the return type of a rule to an incompatible type will result in compilation error.

In addition to changing the return type of the rule, it is also possible to modify the AST node returned by the rule using return code blocks (**return expressions**). Return expression is a Scala expression that returns an object of the class corresponding to the AST node. The expression must be surrounded with curly brackets ("{}"). The type of the returned object must match the return type of the grammar rule. If the return type is not explicitly specified for the given rule, then the return expression must return the case class generated for this rule.

```
grammar ee.cyber.simplicitas.spamdetector.SD;

Program: Item+;

option Item: Rule | Condition;

Rule: "rule" Name score=Score ":" expr=OrExpression;

Condition: "condition" name=Id ":" expr=OrExpression;

option Primitive returns Expression:
      Contains
    | NotContains
    | Count
    | ConditionCall
    | ParenExpression
    | NotExpression;

OrExpression returns Expression
    {if (items.length == 1) items.head else _self}
    : items=AndExpression ("or" items=AndExpression)*;
AndExpression returns Expression{if (items.length == 1) items.head else _self}
    : items=Primitive ("and" items=Primitive)*;

ParenExpression returns Expression {expr}
    : "(" expr=OrExpression ")";
NotExpression: "not" expr=Primitive;

Contains: field=Id "=" Regexp;
NotContains: field=Id "!=" Regexp;
Count: "count" "(" items=ExprList ")" "=" count=Num;
ExprList: items=OrExpression ("," items=OrExpression)*;
ConditionCall: cond=Id;

terminal Regexp: "/" (~"/")* "/" "i"?;
terminal Score: "-"? Digit+ "." Digit*;
terminal Num: Digit+;
terminal Name: '"' (~'"')* '"';
fragment Digit: '0'..'9';

terminal Id {var ref:  Id = null}: IdStart IdNext*;
fragment IdStart: 'a'..'z'|'A'..'Z'|'_';
fragment IdNext: IdStart|'0'..'9'|'-';

fragment MlComment: '/*' (~'*' | '*' ~'/')* '*/';
fragment SlComment: '//' ~('\n'|'\r')*;
hidden terminal WS: (' '|'\t'|'\r'|'\n'|SlComment|MlComment)+;
```

Figure 5.9: SpamDetector grammar (cf. Figure 5.4) extended with annotations

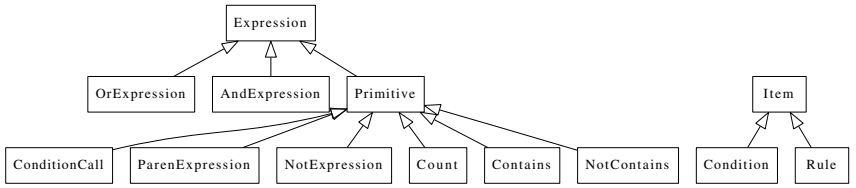


Figure 5.10: Inheritance hierarchy of SpamDetector AST classes

The return expression is executed in a scope that contains all the parameters of the rule. The AST node corresponding to the rule itself is also accessible via the identifier *_self*. For example, the rule *ParenExpression* on line 26 returns the expression inside parentheses. This results in a cleaner AST. Without the return expression, string "`(foo)`" would generate AST node `ParenExpression(Id("foo"))`. With the return expression, the result is simply `Id("foo")`. Because the rule *ParenExpression* now returns objects of type *Expression* (the type of attribute *expr*), we must also change the return type of the *ParenExpression*.

A more complicated example can be found in rules *OrExpression* and *AndExpression* (lines 19 and 22). The return expression checks whether the *OrExpression* represents a degenerate case (call to *AndExpression* without using the *or* keyword) and, if so, returns the single node. In this case, the list of child nodes can be accessed via name *items*, whereas the whole *OrExpression* node is named *_self*. This use of return expressions avoids layers of wrapping that would be generated with the default code generation. For example, the string "`foo or bar`" will be parsed as follows:

```
OrExpression(List(Id("foo"), Id("bar")))
```

instead of

```
OrExpression(List(
    AndExpression(List(Id("foo"))),
    AndExpression(List(Id("bar")))))
```

Long and complicated return expressions can make the grammar description long and obscure the structure of the language. In these cases it is recommended to put the longer code in a separate Scala file and only use function calls as return expressions.

Figure 5.10 shows the inheritance hierarchy of the SpamDetector grammar. For clarity, we have omitted classes that do not have inheritance relationship with the other classes.

Objects returned by the return expressions are not limited to AST classes generated from the grammar. It is possible to manually define new AST classes and use them from the return expressions. When doing this, the new classes must extend the return type of the grammar rule returning the AST node. If the types do not match, compilation fails with a type error. Besides the requirement to extend the return type of the rule, the synthetic AST class does not need to correlate with the rule (for example, it makes no difference whether the rule is an option rule or a non-terminal rule).

To illustrate the use of synthetic classes, we show how to use an synthetic class *BinOp* to offer a uniform representation of the binary arithmetic operators. Using this class, the expression `a and b or c` is represented by the following AST node:

```
BinOp("or", BinOp("and", List(Id("a"), Id("b"))), "c")
```

Using a single class for representing all the different arithmetic operators can simplify processing code because only one kind of AST node needs to be matched when processing different types of expressions[5].

Figure 5.11a shows the definition of the *BinOp* class. The *AndExpression* and *OrExpression* nodes can be converted to *BinOp* nodes using the *makeBinary* method defined in Figure 5.11b. The first parameter to the method is the operator ("*and*" or "*or*") and the second parameter is the list of arguments given to the operator. In the grammar description, the conversion function can be called by return expression, such as `{makeBinary("or", items)}`. When constructing the *BinOp* nodes, *setStart* and *setEnd* methods are called to set source location of the node. The start location is copied from the left operand and the end location is copied from the right operand. Setting the source locations is important if the AST nodes are used to report errors (e.g., type errors). Similar to complicated return expressions, the synthetic AST classes should be defined in a separate file.

5.3.5 Implementation

Currently the Simpl parsing subsystem is implemented as a front-end to the ANTLR parser generator. The Simpl parser generator parses the grammar description and generates two artifacts. The first artifact is a Scala file that contains definitions for the case classes that are used to represent

[5]However, for SpamDetector the effect is not very strong because the language only contains two arithmetic operators.

```
case class BinOp(
   var op: String, // The operation, such as "and" or "or"
   var left: Expression, // Left operand
   var right: Expression) // Right operand
   extends Expression
```

(a)

```
def makeBinary(op: String, args: List[Expression]): Expression =
    args match {
        case single :: Nil => single
        case left :: right :: rest =>
            makeBinary(
                BinOp(left, right).setStart(left).setEnd(right) ::
                rest)
    }
```

(b)

Figure 5.11: Definition of the *BinOp* synthetic AST class (a) and a method to convert the original AST node to use the *BinOp* class

the AST. It also contains a class that acts as a front-end to the parsing functionality (parsing the file, retrieving token stream, retrieving AST) and that allows easy integration of the parser with the other Simpl APIs. The second artifact is an ANTLR grammar that parses the input and builds the AST via grammar actions. The generated ANTLR grammar is then processed with ANTLR to produce the parser that will be linked into the main program.

Because Simpl uses the ANTLR parser generator, it also inherits limitations of ANTLR. In particular, ANTLR uses the LL(k) parsing algorithm that limits the set of grammars that can be expressed. The most important restriction is that Simpl does not currently support left-recursive grammars. This can make expressing arithmetic operators cumbersome and can produce inefficient ASTs. In Simpl, this limitation can be worked around by using return expressions (see Section 5.3.4).

5.4 Language Processing

Simpl relies on the Scala programming language for expressing program transformations. Scala is a high-level programming language that supports both functional and object-oriented programming styles. For example, the AST classes can be processed either via pattern matching or by using object-oriented attribute access. The high abstraction level of Scala language makes it possible to express transformations in a declarative manner and to combine elementary transformations into compound transformations.

Figure 5.12 shows implementation of reference resolving for the SpamDetector language. The resolving process makes use of two instance variables. Variable *conditions* stores mapping from condition names to *Id* nodes. Variable *errors* stores list of error messages created during the analysis. The process consists of two steps. First, the method *collectConditions* iterates over all the named conditions and adds them to mapping. If a condition has duplicate name, this is logged as an error. Second, the method *doResolve* walks over all the AST nodes using the *walkTree* method that calls given function for each AST node. We use block syntax to create a function that matches *ConditionCall* nodes and checks whether they reference existing conditions. If not, we log an error.

Simpl contains a base class that simplifies implementation of main program for simple code generators or program checkers. Figure 5.13 shows how the *MainBase* is used to implement the SpamDetector main function. First, the call to *parseOptions* parses the command-line options that specify the destination directory and source files (the base options are determined by Simpl, the concrete tool can add additional options). The list of source files is used in line 5. In lines 7 and 11, we use the error reporting functionality that checks presence of errors. If any errors were found, they are displayed to the user and the process exits. Section 5.5 shows how the code generator is instantiated with the destination directory determined by the Simpl framework.

In addition to writing Scala code, Simpl ASTs can be processed with the Kiama language processing library. It is possible to use both Kiama's rewriting library and attribute grammars. Figure 5.14 shows an example that calculates call graph for conditions in the SpamDetector language. The call graph is represented as mapping from condition name to set of condition names that are directly called by the given condition. The example defines two attributes. The attribute *callSet* computes call set for

```
class ResolverScala {
    val errors = ArrayBuffer[SourceMessage]()
    val conditions = Map[String, Id]()

    def resolveReferences(program: Program) {
        errors.clear()
        conditions.clear()

        collectConditions(program)
        doResolve(program)
    }

    private def collectConditions(program: Program) {
        program.items foreach {
            case Condition(id @ Id(name), _) =>
                if (conditions contains name) {
                    errors += error("Duplicate condition name: " +
                            name, id)
                } else {
                    conditions += name -> id
                }
            case _ => () // Do nothing
        }
    }

    private def doResolve(program: Program) {
        program walkTree {
            case ConditionCall(id @ Id(name)) =>
                if (conditions contains name) {
                    id.ref = conditions(name)
                } else {
                    errors += error("Condition not found: " + name, id)
                }
            case _ => ()
        }
    }
}
```

Figure 5.12: Resolving name references in SpamDetector language using Scala

```
object SDMain extends MainBase {                                    1
    def main(argv: Array[String]) {                                  2
        parseOptions(argv)                                           3
        val grammar = new SDGrammar()                                4
        for (arg <- sources) {                                       5
            grammar.parseFile(arg)                                   6
            checkErrors(grammar.errors)                              7
                                                                     8
            val resolver = new ResolverScala                         9
            resolver.resolveReferences(grammar.tree)                 10
            checkErrors(resolver.errors)                             11
                                                                     12
            // Invoke code generator and save                        13
            // results to destDir                                    14
        }                                                            15
    }                                                                16
}                                                                    17
```

Figure 5.13: Main program that calls the parser and program checker

a given expression. The attribute is defined on type *CommonNode* that is the base class for all the generated AST classes in Simpl (the attribute is also evaluated on nodes that are not expressions). For nodes that represent condition calls (*ConditionCall* grammar rule), the call set consists of the name of the condition. For other nodes, we calculate call sets for all of the child nodes and take the union of the results. The attribute *callGraph* is applicable for program items (conditions or rules) and returns a call graph for the given program item. For conditions, the call graph consists of one item and for rules it is an empty map. The method *callGraph(Program)* creates a single graph that unifies the call graphs of all the items.

Figure 5.15 shows method *checkCycles* that uses the call graph attributes to detect loops in condition calls. For each entry in the call graph, it recursively follows the calls while building a list representing the call stack ("blacklist"). If the checked condition calls any conditions in the blacklist, then the checker outputs an error.

5.5 Code Generation

Simpl supports two means for code generation and pretty-printing. For simple code generation tasks, the developer can use the bundled StringTemplate [68] template engine. For cases that require better control over the

```
def callGraph(program: Program): Map[String, Set[String]] =
    program.items.map(callGraph).flatten.toMap

val callGraph: Item => Map[String, Set[String]] =
    attr {
        case Condition(Id(name), expr) =>
            Map(name -> callSet(expr))
        case Rule(_, _, _) =>
            Map.empty
    }

val callSet: CommonNode => Set[String] =
    attr {
        case ConditionCall(Id(name)) => Set(name)
        case node =>
            node.children.map(callSet).flatten.toSet
    }
```

Figure 5.14: Calculating the call graph of condition calls using Kiama attribute library

```
private def checkCycles(program: Program) {
    val callGraph = callGraph(program)

    def check(condToCheck: String, blacklist: Set[String]) {
        val called = callGraph(condToCheck)
        val intersect = called & blacklist
        if (intersect.isEmpty) {
            called foreach(check(_, blacklist + condToCheck))
        } else {
            errors += error(
                "Condition " + condToCheck + " creates endless loop",
                conditions(condToCheck))
        }
    }
    callGraph.keys foreach (cond => check(cond, Set(cond)))
}
```

Figure 5.15: Cycle detection that uses call graph example

generated output and high-quality formatting, Simpl includes a pretty-printing combinator library, based on Philip Wadler's Haskell library [95].

For StringTemplate template engine, Simpl provides a glue that allows accessing AST nodes from the templates. Additionally, Simpl contains helper classes that simplify loading and invoking templates that are packaged with the DSL implementation.

Figure 5.16 shows a StringTemplate template that generates Java code from SpamDetector DSL. In order to run the template, Simpl converts the program AST to a Java map so that it can be accessed from the template with the attribute access syntax. The top-level template *program* (line 3) receives as an implicit argument the AST node corresponding to the *Program* AST class. The main work is done in the *detectorClass* template (line 5) that generates a Java class for a given SpamDetector program.

In order to implement polymorphism (generation of different code for different types of the AST nodes), the template takes advantage of the implicit field *nType* that contains the name of the given AST class. This field is automatically generated by Simpl and the name of the field can be changed so that it does not collide with any existing attribute in the AST. The *nType* field is used in two places to invoke different template depending on the type of the AST node.

The first use is in template *detectorClass* to differentiate between rules and conditions. The code `<items:{itm | <(defName())(itm)>}>` on line 7 iterates over all the items in the program and, for each item executes template whose name is derived from the item type (the derivation is done by *defName()*, defined on line 15). Thus for *Condition* nodes it will execute the *ConditionDefinition* template and for *Rule* nodes the *RuleDefinition* template. Since there are no definitions to be generated for rules, the *RuleDefinition* template will generate an empty string.

The second use in template *expression* (line 34) uses similar mechanism. The *nType* attribute is used to determine the name of template that will be used to generate code for a given type of expression.

Figure 5.17 shows the generator output for the example program from Figure 5.3.

Figure 5.18 shows code for invoking the code generation template. This makes use of the *GeneratorBase* class from the Simpl library. The *GeneratorBase* class simplifies common tasks: loading the StringTemplate templates that are bundled into a jar file together with the code; invoking the

```
group SDGenerator;

program() ::= "<detectorClass(...)>"

detectorClass() ::= <<
class Detector extends DetectorBase {
    <items:{itm | <(defName())(itm)>}>

    public void run() {
        <items:{itm | <(runName())(itm)>}>
    }
}
>>

defName() ::= "<itm.nType>Definition"

ConditionDefinition(cond) ::= <<
private boolean <cond.name.text>() {
    return <expression(cond.expr)>;
}
>>
RuleDefinition(rule) ::= ""

runName() ::= "<itm.nType>Run"

RuleRun(rule) ::=
<<if (<expression(rule.expr)>) {
    addMatch(<rule.name.text>, <rule.score.text>);
}

>>
ConditionRun(def) ::= ""

expression(expr) ::= "<(expr.nType)(expr)>"

OrExpression(expr) ::=
    "<expr.items:expression(); separator=\" || \">"
Contains(expr) ::=
    <<fieldContains("<expr.field.text>", "<expr.regexp.text>")>>

...
```

Figure 5.16: Code generation with StringTemplate

```
class Detector extends DetectorBase {
    private boolean from_mydomain() {
        return fieldContains(From, /@mydomain.com/);
    }

    public void run() {
        if (fieldContains(Subject, /viagra/i)) {
            addMatch("Subject␣contains␣Viagra", 2.0);
        }
        if (fieldContains(Subject, /business proposal/i) &&
                !(from_mydomain())) {
            addMatch("Business␣proposal", 1.5);
        }
    }
}
```

Figure 5.17: Code generated from the example program

```
class SDGenerator(destDir: String)                                    1
        extends GeneratorBase(destDir) {                              2
    val templates = getTemplates("SD.stg")                            3
                                                                      4
    def generate(program: Program) {                                  5
        val args = program.toJavaMap("nType")                         6
        writeFile("GeneratedProgram.java",                            7
                templates.getInstanceOf("program", args))             8
    }                                                                 9
}                                                                     10
```

Figure 5.18: Invoking the StringTemplate engine

template and passing the AST as parameters; and saving the generated file to a given destination directory.

On line 3, the call to *getTemplates* method loads the bundled template file. Line 6 converts the program AST to Java map that is accessible to the StringTemplate library. The parameter to *toJavaMap* method names the attribute that will contain the name of the AST class ("*nType*" in this case). If there is no need for type-based dispatch, this parameter can be omitted. On line 7, the call to *writeFile* method invokes the template and writes the result to a file. The file is placed into directory specified in the *destDir* constructor parameter.

StringTemplate is suitable for creating code generators for simple DSLs where formatting of the output is not very important. For cases that re-

quire better control over the generated output and high-quality formatting, Simpl includes a pretty-printing combinator library, based on Philip Wadler's Haskell library [95]. The pretty-printing library is useful for generating quality code from complex, highly structured ASTs, and for implementing code formatting tools.

Figure 5.19 implements the example SpamDetector code generator with the pretty-printing library (code generation of expressions is omitted for clarity). The entry point function first transforms the AST to a structured document (datatype *Doc*) and then formats the document for required line length (75 in this example). This example demonstrates the use of basic concatenation operators: "*::*", "*:+:*", "*:#:*", and "*:|:*". The first three simply concatenate two documents separating them, respectively, with nothing, space, and newline. The last operator concatenates two documents with optional line break between them. If there is enough room, the documents will be separated by space, otherwise, there will be a line break between the two documents. Additionally, the example uses modifiers *indent* and *hang* to control how the broken lines are formatted. The expression *indent(n, x)* indents document x by additional n spaces. The expression *hang(n, x)* specifies that if document x is wrapped on formatting, then the continuing line will start at indent n.

The pretty-printing library provides the developer a fine control over indenting and line wrapping. To illustrate this, Figure 5.20 shows excerpt of code generated from the example rule, formatted using a very short line length.

5.6 IDE

Simpl contains an IDE framework that is based on Eclipse IMP. IMP is an eclipse-based framework for creating language IDEs that contain the program editor and other typical features such as code navigation and outline view. IMP requires the language developer to implement set of *language services* that are called by the IMP framework to direct the behavior of the IDE features. Examples of the language services are the outlining service that builds the contents of the outlining view and the content-proposer service that returns the list of completion options that are displayed when the user activates the auto-complete feature in the code editor. IMP language service API is quite abstract and does not require the use of any particular parsing technology or AST representation.

```
object CodegenPP {
    def prettyPrint(program: Program, writer: Writer) {
        val doc = prettyPrint(program)
        show(doc, 0.8, 75, writer)
    }

    private def prettyPrint(program: Program): Doc =
        "class␣Detector␣extends␣DetectorBase␣{" :#:
            indent(
                hcat(program.items map itemDef) :#:
                "public␣void␣run()␣{" :#:
                    indent(hcat(program.items map itemCall)) :#:
                text("}")) :#:
        text("}")

    private def itemDef(item: Item) = item match {
        case Condition(Id(name), expr) =>
            "private␣boolean" :+: name :: "()␣{" :#:
                indent(hang(4, "return" :+: prettyPrint(expr) ::
                    ";")) :#:
            "}" :: line
        case _ => empty
    }

    private def itemCall(item: Item) = item match {
        case Rule(Name(name), Score(score), expr) =>
            hang(4, "if" :+: parens(prettyPrint(expr)) :+: "{") :#:
                indent(hang(4, "addMatch(" :: name ::"," :|:
                    score :: ");")) :#:
            "}" :: line
        case _ => empty
    }

    private def prettyPrint(expr: Expression): Doc = expr match {
        ...
    }

    private val indent: Doc => Doc = Doc.indent(4, _)
}
```

Figure 5.19: Code generation using pretty-printing library

```
if (fieldContains(
        Subject,
        /viagra/i)) {
    addMatch("Subject␣contains␣Viagra",
        2.0);
}
if (fieldContains(
        Subject,
        /business proposal/i) &&
    !(from␣mydomain())) {
    addMatch("Business␣proposal",
        1.5);
}
```

Figure 5.20: Pretty-printed output with short line length

Simpl builds on top of IMP APIs to provide a IDE framework for languages implemented with the Simpl DSL toolkit. It integrates the IMP with Simpl parser generator and AST representation. The Simpl IDE framework aims to abstract away some of the technical details in present in the IMP APIs and offer intuitive and high-level interfaces for the DSL developer. In addition, the IDE services can be programmed without writing any XML configuration files and without using any Eclipse APIs.

The Simpl IDE framework supports the following IDE features:

- syntax highlighting in the code editor;
- marking the errors in the code. The parsing errors are highlighted automatically, the developer can also add errors encountered during program checking;
- outline view that displays the structure of the DSL program;
- code folding that allows the user to hide and show code blocks (such as classes, functions);
- navigating to the place of definition of an identifier (hyper-linking);
- highlighting all the occurrences of an identifier;
- automatic code completion that offers completion choices based on the text under cursor;
- documentation tooltips that are displayed when the user hovers mouse over an identifier; and

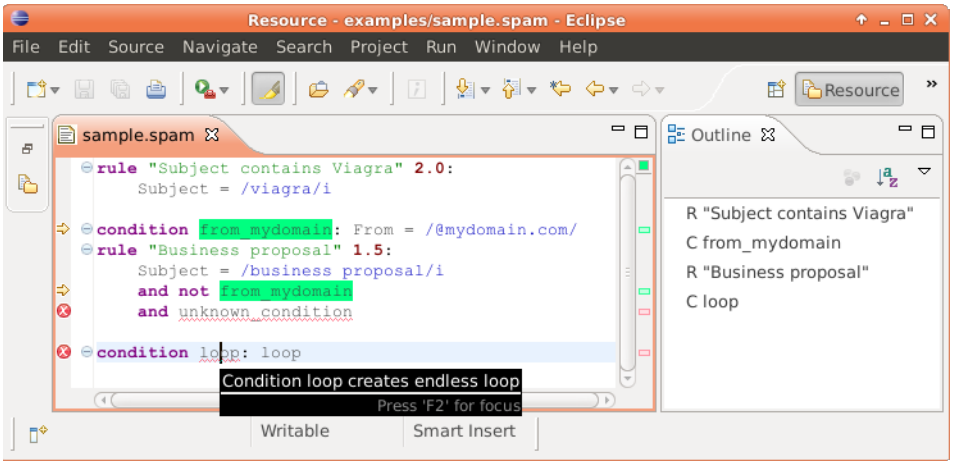


Figure 5.21: Screenshot of SpamDetector IDE

- running code generator for the current file.

Figure 5.21 shows a screenshot of IDE for the SpamDetector language, created with the Simpl IDE framework.

In order to make it easy to develop IDEs for simple languages with minimal amount of effort, Simpl defines default behaviors for some of the IDE services. The syntax coloring service automatically colors keywords and comments using information in the grammar description. The code completion service automatically offers keywords for the completion. The code folding service offers folding for all the code blocks that are present in the outline view. The occurrence marking service uses information provided by the hyper-linking service to find all the instances of a given identifier. Using the Simpl IDE framework, the DSL developer can create a working language IDE without writing any code at all. For the simple cases, the developer can implement DSL parser and code generator and get the DSL IDE for free.

However, the IDE services not mentioned in the previous paragraph (such as the outline view) require the developer to write code to provide information necessary for functioning of the service. In addition, the developer can customize the services provided by Simpl to gain better control over the IDE behavior.

Figure 5.22 shows an implementation of a basic IDE for the SpamDetector language. The parsing service is called when a file is loaded into the editor or changed. The simplest implementation of the *parse* method (lines 3-9)

```
class SDConfig extends APluginConfig {
    /** Parses the input using the correct grammar. */
    def parse(ctx: ParseCtx) {
        val grammar = new SDGrammar
        ctx parse grammar
        val resolver = new ResolverScala
        resolver resolveReferences grammar.tree
        ctx reportErrors resolver.errors
    }

    /** There is nothing to show in the outline view. */
    def treeLabel(node: CommonNode) = node match {
        case Condition(Id(name), _) => "C␣" + name
        case Rule(Name(name), _, _) => "R␣" + name
        case _ => null
    }

    override def referenceTarget(node: CommonNode) = node match {
        case id: Id => id.ref
        case _ => null
    }

    override def runGenerator(dir: String, file: String) {
        SDMain.main(Array("--dest", dir, dir + file))
    }

    override def getTokenColor(token: GenericToken): Symbol = {
        val myToken = token.asInstanceOf[CommonToken[SDKind.Kind]]

        myToken.kind match {
            case SDKind.Name => 'ruleName
            case SDKind.Score => 'score
            case SDKind.Regexp => 'regexp
            case _ => null
        }
    }
}
```

Figure 5.22: IDE services for the SpamDetector example

is "`ctx parse new SDGrammar`" that simply parses the program text using the SpamDetector grammar. In the example, we also run the reference resolver and report the errors found (lines 6-8). The outlining service consists of two methods: *treeLabel* (lines 12-16) and *treeIcon* (not implemented in the example). These methods should return, respectively, a label and an icon that is displayed for a given AST node in the outline view, or *null*, if the node is not part of the outline. The reference resolving service is implemented via the *referenceTarget* method (lines 18-21). It uses the links established by the reference resolver to return the AST node that is pointed by the given identifier (the pointers were filled in by the call to reference resolver on line 7). The *referenceTarget* method is also used by the Simpl framework for marking the occurrences of a given identifier. The generation service is called when the DSL user selects "Generate" from the DSL file context menu. In the simplest case, the *runGenerator* method (lines 23-25) calls the main function of the non-visual code generator.

Finally, the token coloring service returns highlighting information for a given token. Instead of assigning specific colors to different tokens (keywords, identifiers, etc.), the coloring service operates on abstract token codes (*ruleName*, *score*, and *regexp* in the example). Simpl manages the concrete colors and font attributes (such as bold and italic) associated with each token code. The default values are provided by the DSL developer, but they can be configured by the DSL user. Figure 5.23 shows the syntax highlighting configuration screen provided by Simpl to configure the syntax highlighting for the SpamDetector language.

Lines 27-36 of the IDE code contain the *getTokenColor* method that takes as an argument a program token and returns a color code that is used to display this token. The example code does not alter the behavior for Simpl's built-in token types: keywords, comments and operators. If the developer wishes to e.g., color some keywords differently than the others, he can override the default behavior by defining an additional color code and adding a case to the match statement in the *getTokenColor* method that checks for the particular keyword and returns the newly defined color code.

5.7 Comparison with Parallel Research

Simpl was developed in 2009-2010. During that time period, there was also parallel research performed by other research groups that resulted in creation of new DSL toolkits or improvement of existing tools. This section

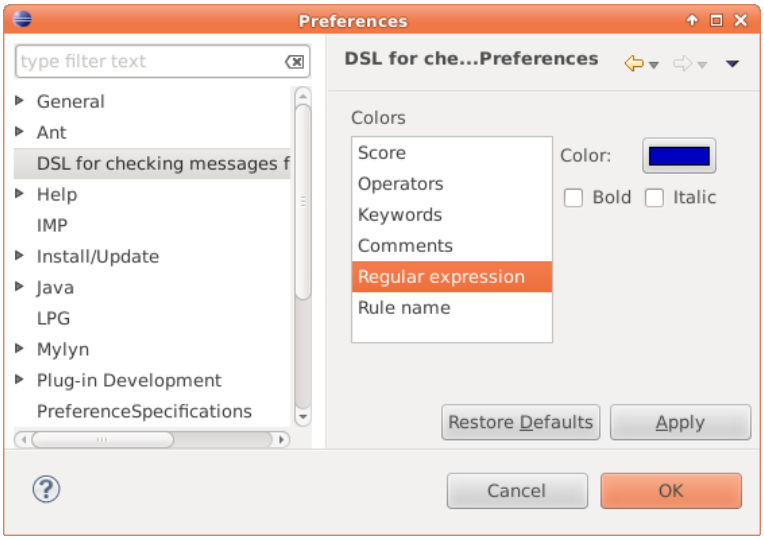


Figure 5.23: Screenshot of SpamDetector IDE configuration

reviews two toolkits developed in parallel with Simpl (namely Spoofax and Rascal) and compares them with the Simpl DSL toolkit.

Note that these two parallel tools received more development effort (larger development teams) and thus at the current time they can be somewhat more mature and have features not present in Simpl.

5.7.1 Spoofax

At the time of writing this thesis, the Spoofax/IMP workbench (see Section 4.4) has incorporated the previously separately distributed Stratego language processing tool into a unified language toolkit named Spoofax. The Stratego compiler and runtime have been converted to Java. With this modification, Spoofax produces DSL implementations that can be embedded in Java-based enterprise systems (the requirement SYSTEM is now fulfilled by Spoofax). Therefore, Spoofax can be used in the same way as Simpl. However, the two systems have different architecture and offer different user experiences. The remainder of this section compares the two tools.

Spoofax uses SDF for describing grammar of the target language. SDF produces scannerless parsers that do not have some of Simpl's limitations. Current implementation of Simpl uses LL(k) parsing algorithm and thus cannot parse left-recursive grammars. Also, the separation between lexer

and parser makes it difficult to parse languages such as PL/I where keywords can also be used as identifiers. On the other hand, in this generality comes at a price. For example, when using SDF, the DSL developer must create a separate list of all the keywords in a grammar description to ensure that they are not parsed as identifiers. In Simpl this is not necessary because all the keywords in grammar description have higher priority than other lexer rules (such as rules for recognizing identifiers).

SDF offers better support for creating modular and extensible parsers. It is possible to split the grammar description into several files and combine the files to produce different languages. It is easy to extend a language by adding additional production rule. Simpl does have support for modular and extensible grammars (see Section 5.3.2.1), but it is more constrained and handles fewer different modularity use cases.

For language processing (program checking and program transformation) Simpl and Spoofax take different approaches. Spoofax is based on the Stratego program transformation language. Stratego offers facilities for expressing rewrite rules and controlling the application of the rewrite rules via programmable strategies. While these features can allow expressing the program transformations very succinctly, they can also lead to code that is difficult to understand. Unlike typical programming languages, the strategies usually do not explicitly refer to data and instead focus on combining the transformation rules and strategies into higher-level strategies. This and the use of backtracking can make it more difficult to reason about the program and the programming paradigm is likely unfamiliar to a typical "mainstream" software developer.

On the other hand, Simpl targets professional software developers who are not specialists in language tools research. Simpl is based on general-purpose Scala programming language that offers good language-processing facilities while being reasonably popular and easy to learn for developers who come from Java or C++ background. One of the goals of Simpl is to lower the entry barrier and make the tool more accessible for mainstream developers.

Both Simpl and Spoofax use the IMP toolkit for creating Eclipse-based IDEs. Thus, they implement similar features with similar look and feel. The Spoofax has an advantage in that it is possible to preview and test the DSL IDE without restarting the Eclipse. Simpl, like most other Eclipse-based language toolkits, requires the developer to launch another Eclipse instance to test the language IDE.

5.7.2 Rascal MPL

Rascal MPL (Meta Programming Language) [49, 71] is a domain specific language and a toolset for source code analysis and manipulation. Rascal was developed in parallel with Simpl and, at the time of writing this thesis, is at alpha stage. However, it is actively developed and provides functionality that is comparable to Simpl. Note that because the Rascal documentation is occasionally incomplete, this review may contain errors in description of some details.

Parser Similarly to Spoofax, Rascal uses the SDF language for describing concrete syntax of the DSLs (see Section 5.7.1 for comparison of SDF and Simpl). Like Spoofax, Rascal makes available for the developer both the concrete syntax tree and the abstract syntax tree of the DSL program. The former corresponds directly to the source program and the production rules (the concrete syntax tree contains complete information about the source, including the white space). The latter is more abstract and is generally used for program manipulation. For a single DSL, the developer can define several abstract syntaxes for different purposes (analysis, transformation, etc.). The conversion from the concrete to the abstract syntax tree can be either manual or automatic. When using automatic conversion, the developer annotates the grammar rules with names of abstract syntax tree nodes and the built-in function *implode* automatically converts the parse tree. If the developer requires more flexibility he can write a conversion function from the concrete to the abstract syntax tree using Rascal's pattern-matching capabilities. In both the automatic and the manual case, the developer must explicitly create the definitions for the AST data types. This may make it inconvenient to quickly develop small DSLs because the developer needs to create and keep synchronized multiple copies of the language structure (Rascal fails requirement SMALL). Like Spoofax, Rascal can be used to develop modular and extensible languages (satisfies CUSTOM).

Program checking and transformation For language processing tasks, Rascal takes a similar approach to Spoofax – it is based on a domain-specific language. The Rascal language has support for comprehensions, pattern matching and rewrite rules. In addition, the Rascal language has support for common language processing tasks, such as tree walking (visiting) and constraint solving.

```
data Program = program(list[Bean] beans);
data Bean = bean(str name, list[Attr] attrs);
data Attr = attr(str name, AttrType attrType);
data AttrType = idRef(str id) | beanRef(Bean ref);

public Program resolve(Program prog) {
    map[str, Bean] beans = beansMap(prog);
    return visit(prog) {
        case idRef(refName) => replaceRef(refName, beans)
    }
}

private map[str, Bean] beansMap(Program prog) =
    (b.name: b | b <- prog.beans);
private AttrType replaceRef(str refName,
        map[str, Bean] beans) {
    return if (refName in beans) beanRef(beans[refName]);
           else idRef(refName);
}
```

Figure 5.24: Example Rascal code that implements name resolving for the bean language

Figure 5.24 shows the name resolution example introduced in Chapter 4, in Rascal. The first four lines describe the abstract syntax for the bean language. The entry point is the function *resolve* that takes as input the program AST and returns the transformed AST. In the transformed AST the occurrences of *idRef* (reference to an arbitrary identifier) are replaced with *beanRef* (direct reference to a bean) where appropriate. First, the function constructs a map of beans, indexed by the bean name. This is done by the helper function *beansMap* that uses list comprehension to build a map. In the second step, the function *resolve* walks the whole AST using the *visit* keyword. When visiting encounters a node of type *idRef*, the node is replaced with the return value of the function *replaceRef*. The *replaceRef* simply checks whether a bean with the referred name exists, and if so, returns *beanRef* node. If the identifier does not refer to a bean, it returns the unchanged input node.

While the Rascal language aims to have more conventional program structure and control flow than Spoofax, it is still quite different from typical "business" languages and therefore the same considerations as with Spoofax apply (Rascal fails requirement LANG, see Section 5.7.1 for more discussion).

Code generation For simple code generation tasks, it is convenient to use Rascal string building facilities. In Rascal, strings can contain templates that function in a manner similar to the PHP language. The templates can be nested. Rascal also supports Box language [38, 39] for pretty-printing trees. In addition to support for generating code, Rascal has support for program visualization (generating graphs from the program). For example, the visualization feature can be used to generate a call graph of the DSL program.

Program editor Rascal comes with an Eclipse-based IDE library that is based on the IMP framework. This allows creating language IDEs with all the typical features (syntax highlighting, error marking, outlining, code folding, hyperlinking, code completion, etc.).

Non-functional requirements DSL implementations created with Rascal can be built from the command line (satisfies BUILD) and embedded into a bigger system (satisfies SYSTEM). The program editor can be distributed as a separate Eclipse plugin (satisfies SEP_WB).

5.8 Evaluation Against Requirements

This section evaluates the Simpl DSL toolkit with respect to the requirements described in Chapter 3 and used in Chapter 4 to evaluate state of the art. First, we overview the functional requirements.

Parser Simpl uses a grammar description to generate both the parser and the Scala case classes that will be used to represent the AST of the parsed program. The developer can annotate the grammar description to direct the generation of an AST. In addition to simple transformations, such as renaming attributes or changing return types of the grammar rules, the developer can also use return expressions that calculate the AST node that is returned by the grammar rule. Simpl supports lexer states – a structured way to make lexical syntax context-sensitive.

Simpl parser generator currently uses ANTLR as the backend and is thus constrained with the LL(k) parsing algorithm. This makes it inconvenient for expressing some grammars because the grammar description must be refactored to eliminate left recursion.

Program Checking and Program Transformation For language processing tasks, Simpl relies on the Scala programming language. Scala's high level of abstraction and support for pattern matching make it very suitable for this purpose. In addition, Simpl supports the use of Kiama language processing library.

Code Generation Simpl supports two methods for code generation. For simple tasks, Simpl has bindings for the StringTemplate template engine. For more complex tasks, Simpl contains a pretty-printing library that can be used to produce high-quality output.

Program Editor Simpl contains IDE framework that is based on the Eclipse IMP. The editor API is high-level, abstracting away the specifics of Eclipse. Also, Simpl aims to reduce the boilerplate code. Simpl provides default implementations to most basic IDE services and therefore a basic language IDE can be obtained without writing any lines of code.

Non-Functional Requirements Simpl is based on the Scala programming language that is popular and easy to learn for a developer with Java or C++ background. Therefore Simpl satisfies the requirement LANG. Additionally, Simpl parser and APIs can, in principle, be used from Java code (the program transformation code can be written in Java).

Simpl aims to aid in creating language implementations that are small and simple. In particular, we try to minimize the amount of boilerplate code that needs to be written. Automatic AST generation means that once the language developer has written the grammar description, he can already start to implement program checker or code generation. Additionally, the basic version of the IDE requires no coding effort and thus, for simple DSLs, it comes for free. Thus Simpl satisfies the requirement SMALL.

The language IDE created with Simpl is packaged as an Eclipse plugin that can be installed separately from the language developer's workbench. Thus, Simpl satisfies the requirement SEP_WB.

Language implementations based on Simpl run on the Java Virtual Machine and do not make assumptions on their environment. Therefore, they can be embedded both into a build system or into an enterprise system itself (satisfies requirements BUILD and SYSTEM).

Simpl stores the DSL programs in plain text files, thus satisfying requirement VCS.

Simpl offers partial support for implementing customizable and modular languages (requirement CUSTOM). The grammar inclusion feature can be used to implement simple cases where the extension language adds additional language constructs to the base language or where one core set of grammar rules is reused in several languages. See Section 6.2 for an example of modular language implementation. From the language processing side, the Scala language has excellent support for writing modular code.

CHAPTER 6

EVALUATION

This chapter presents results of usability evaluations of the Simpl DSL tool. First, we describe our experiences of implementing the DSLs in the Customs Engine (presented in Chapter 2) using Simpl. Second, we present results of a tool challenge where different language tools were used to complete the same task – implementing an Oberon0 compiler. Next, we describe another experiment that compared implementation of a benchmark DSL with implementations done using different DSL tools. Finally, we report on an experiment that compared time spent on implementing a DSL using Simpl and using a comparison tool.

6.1 Languages from the Customs Engine

In Chapter 2 we presented Customs Engine (CuE) as an example of an enterprise system that relies extensively on DSLs. In particular, it used two different kinds of DSLs. The first was Burula, a fairly complicated DSL for expressing document verification rules. The second kind was a set of configuration DSLs that were mainly used by programmers to generate repetitive parts of the Java code and to rise the level of abstraction by creating high-level descriptions of component structure or behaviour.

As the CuE was one of the main forces that inspired creation of the Simpl DSL tool, we reimplemented representatives of both types of languages present in CuE using Simpl in order to verify that Simpl is indeed suitable for this domain. The next subsections describe our experiences during these re-implementations.

6.1.1 Burula

We used Simpl to re-create the parser and the IDE for the Burula language. A screenshot of the resulting IDE is shown on Figure 6.1. Because the Burula code generator is quite complicated and there are no problems with the current implementation, we decided not to replace the current code generator but instead use Simpl to create convenient IDE. The Burula IDE is currently being used by system analysts to create business rules for different Customs Engine modules and the feedback has been positive. The parser and the IDE was implemented by the author of this thesis and the total time spent was about 10 hours.

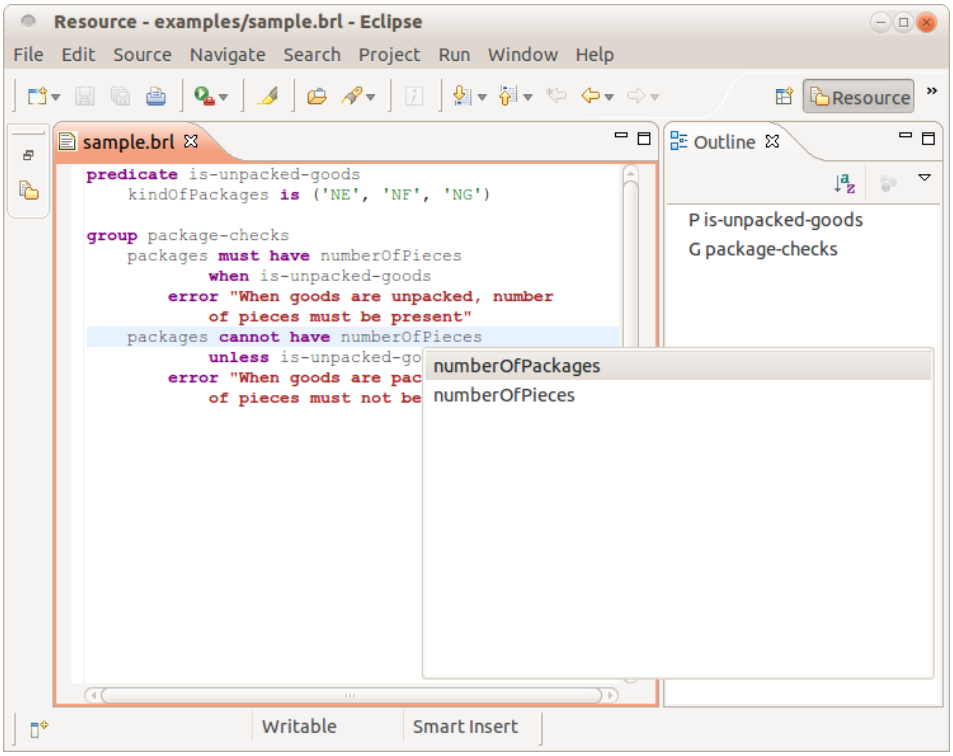


Figure 6.1: Screenshot of Burula IDE created with Simpl

Burula makes extensive use of syntactic white space. In particular, it uses indenting to determine when one language construct (such as rule, predicate or group) ends and another begins. This makes the syntax more English-like (it uses no semicolons, blocks or other interfering symbols), but more difficult to parse. Because the original parser was hand-crafted, this did not pose significant problems. However, Simpl, like most parser generators,

does not have support for syntactic white space. Instead, in a typical parser, white space is detected by the lexer and ignored by the context-free rules.

When creating the Burula implementation in Simpl, a large part of the work (6 hours) was spent on implementing support for syntactic white space. Before parsing, the Burula program was fed through a filter that analyzed the indentation at the beginning of each line and inserted additional tokens to signify end of statements and statement groups. We used characters @ and # as terminators because they do not otherwise occur in the Burula language. The resulting program had explicit statement terminators and was parsed with the parser created with Simpl. In addition to the parser, the IDE also needed some reworking because the program text fed to the parser was now different from the original program text and that caused inconsistencies in the IDE. In order to fix that, the parsed program was again processed and the previously inserted terminator tokens were removed. Thus, the modified version of the language only existed during the parsing phase.

In retrospect, the decision to use syntactic white space in Burula may not have been the best course of action. From the technical point of view, this feature also caused some problems in the hand-crafted parser. From a user-experience point of view, syntactic white space provides for more "natural" and less cluttered syntax, but careful application of line terminators and blocks (e.g., {}) can visually separate or group program elements and give clear visual indication what parts of the program belong together.

The implementation of the Burula IDE took about 1 hour and was spent improving on the basic editor provided by Simpl. The necessary tasks were the creation of outline view, syntax highlighting for additional tokens (by default, Simpl only colors keywords and comments), and automatic completion of identifiers. The auto-complete works by reading in a word list that contains database fields and functions that are available in a given CuE module. The word list can be generated from an XML schema that describes the structure of the custom documents being processed by the Burula programs.

In CuE, the compiler to Burula language is embedded in the system and invoked at run time when the user uploads new set of rules. Simpl fulfills this requirement elegantly: the language implementation created with Simpl can be packaged as a Java library that makes no assumptions about its runtime environment and does not use globally shared data (thus being thread safe). It can be called directly from the CuE code without no performance penalty.

We decided not to reimplement the existing Burula compiler using Simpl because the existing compiler is very stable and mature and there is a substantial body of DSL programs that may depend on its implementation details. However, the Burula IDE implemented with Simpl is currently being used by systems analysts at Cybernetica AS to write document processing rules.

6.1.2 Configuration DSLs

In the original implementation of CuE, an in-house templating tool Templater was used to create simple DSLs with code generators. These DSLs were mostly used by programmers to generate boilerplate code. Thus, the focus was not on comprehensive tooling but instead the ability to quickly create a DSL with code generator. In order to test the suitability of Simpl for this purpose, we re-implemented a representative configuration DSL with Simpl.

We chose the DSL for describing the layout of the document editing user interface that is also given as an example in Section 2.2.2. It is a representative of a typical CuE configuration DSL in that it mainly describes a data structure and generates code that constructs this data structure. However, the data structure is non-trivial and the generated code varies depending on the input data.

DSLs created using Templater use S-expressions as basic syntax. In the Simpl implementation we decided to omit the parentheses and instead use keywords to structure the DSL program. Figure 6.2 shows example screen description in both formats. The code generation was implemented in StringTemplate and invoked using Simpl's glue to the StringTemplate library. The total amount of time spent on the implementation was 2 hours with 0,5 hours spent on parser and 1.5h spent on code generator. We did not spend any time on the IDE, but with very little effort it is possible to improve on the basic IDE offered by Simpl to implement outline view and code folding. Thus, Simpl is suitable for creating "quick and dirty" DSLs for code generation. Simpl-based code generator integrates well with the overall build system of the project.

When comparing the two tools, Simpl has several benefits over Templater. First, Simpl provides cleaner and easier to read syntax for DSLs. For technical users, this is not so big issue, but if the DSLs are to be used by

```
screendescription

(package ee.cyber.emcs.web)

(import
  java.util.List)

(class EaadScreen)

(box-classificator EaadBox)

(tabs
  (general "General")
  (subject "Consignor/Consignee"))

(default-tab general)

(table xmlEaad.guarantor
  12A
  12B
  (order-hint 10)
  (size-limit 20))

(descriptions
  (1D general)
  (1A general ProfileClassificatorField
    (GetValues DestinationTypeCode
      (check-classif-date)))
  ((1B 1BA) general)
  ((6A 6B) subject TableEntryWidget
    "6AB - Complement consignee"))
```

(a)

```
package ee.cyber.emcs.web

import
  java.util.List

class EaadScreen

box-classificator EaadBox

tabs
  general "General"
  subject "Consignor/Consignee"

default-tab general

table xmlEaad.guarantor
  12A
  12B
  order-hint 10
  size-limit 20

descriptions
  1D general
  1A general
    editor ProfileClassificatorField
      get-values DestinationTypeCode
        check-classif-date
  (1B 1BA) general
  (6A 6B) subject
    editor TableEntryWidget
      ("6AB - Complement consignee")
```

(b)

Figure 6.2: Example screen description using Templater syntax (a) and Simpl syntax (b)

analysts, better syntax may help (or at least make the DSL more attractive). Second, using Simpl provides the DSL user better overview of the DSL grammar. With Templater, there is no grammar definition as such and the DSL user simply has to read the template and see which data structures the input file is assumed to contain (this assumes that a typical quick DSL would lack comprehensive documentation). When using Simpl, the DSL grammar is clearly specified and it is possible to generate grammar documentation that can be used as a reference when writing DSL programs[1]. Third, Simpl provides basic IDE for the language that can also be improved

[1]Simpl contains a tool that generates documentation from the grammar specifications. It also supports Javadoc-style documentation comments.

to offer additional features. The main benefit of Templater over Simpl is better performance. Templater contains a manually optimized parser for S-expressions that is used for both the input files and templates, and the template engine also offers good performance.

6.2 Oberon-0

In this section, we discuss the validation of Simpl in the context of a DSL tools challenge.

6.2.1 LDTA Tool Challenge

Simpl was used to create an entry in the 2011 LDTA tool challenge [55]. The goal of the challenge was to compare different language tools by implementing the same set of tasks that was based on Oberon-0 – a simple imperative language designed by Niklaus Wirth for his compiler construction book [97]. The challenge was structured around language levels (see Table 6.1) and tasks (see Table 6.2). The language implementations were created incrementally, starting from a simpler subset of the language and subset of the tasks, and, through several tasks, moving up to full implementation of the Oberon-0 language. The challenge defined five implementation artifacts (listed in Table 6.3) to guide the incremental implementation of the language. Each artifact represented a combination of language levels and tasks that were implemented for this language level. In the implementations, each artifact is implemented by a software module that is able to run separately but can depend on the modules for lower-level artifacts.

Table 6.1: Language levels used in task description

Level	Description
L1	Oberon0 with primitive types, simple expressions, and assignment statements
L2	L1 with Pascal-style *for* loop and *case* statement
L3	L2 with support for procedures
L4	L3 with support for arrays and records

The Simpl implementation of the challenge is described in technical report [22]. The source code can be found at `https://github.com/margusf/ldta-challenge`. Simpl is not directly targeted at implementing full-featured programming languages (instead, the focus is on DSLs

Table 6.2: Functional tasks

Task	Description
T1	Parsing and pretty-printing the Oberon0 program
T2	Name analysis – binding the name uses to their declarations and reporting the errors
T3	Type analysis – checking type correctness of the program and reporting the errors
T4	Source-to-source transformation – lifting the nested procedures to top level and performing other transformations, such as expressing complex language constructs in terms of simpler ones
T5	C code generation – translating the Oberon0 program to ANSI C

Table 6.3: The challenge artifacts

Artifact	Language	Tasks	Comments
A1	L2	T1-2	Core language with pretty-printing and name analysis
A2a	L3	T1-2	A1 with added support for procedures
A2b	L2	T1-3	A1 with type checking
A3	L3	T1-3	Composition of A2a and A2b
A4	L4	T1-5	A3 with support for arrays and records, source-to-source transformation, and code generation

and simple code generators). Therefore, Simpl only offers direct support for a subset of all the tasks in the tool challenge – parsing, pretty-printing and code generation. Rest of the challenge tasks were simply programmed in Scala with some help from the Simpl libraries. Nevertheless, the tool challenge was a useful experiment to verify that Simpl is also suitable for implementing complex languages. The next subsections describe the Simpl entry to the tool challenge and compares it with the other entries.

6.2.2 Implementation in Simpl

The Oberon0 parser was implemented using Simpl's parser generator. Since the challenge involved significant amount of AST manipulation, we used re-

turn expressions to give the generated AST a better shape. In particular, we created unified AST nodes for all the unary and binary operators so that they could be treated in an uniform manner. We also added additional attributes (referred node, type of the expression, etc.) to AST classes for use in later processing phases. The challenge prescribed that the Oberon0 language is implemented in several steps, starting with the base language and incrementally adding additional language constructs. The Simpl implementation used the grammar import feature: the grammar for the more complex language imported the grammar for the simpler language and added production rules for the new language constructs. Thus, the modularity features of Simpl were adequate for the task.

Language processing tasks were implemented in the Scala programming language. During name analysis, the identifier nodes in AST were annotated with the locations of their declaration (pointer to AST node containing identifier declaration). Type analysis annotated the AST with type information and checked whether the required types matched the actual types. Source to source transformation walked through the program and lifted (in-place) nested procedures to the top level. Overall, Scala proved to be a very suitable language for these tasks.

Code generation and pretty-printing was implemented using Simpl's pretty-printing library. The implementation used Scala's pattern matching features combined with the pretty-printing combinators. This resulted in very readable code that produced high-quality output.

In addition to the official challenge task, we implemented an IDE for the Oberon0 language. This involved little effort because the IDE depended on the functionality implemented in the Oberon0 compiler. Thus, most of the IDE services were implemented in a few lines of code that called the relevant functionality of the Oberon0 compiler.

Overall, we did not encounter any major difficulties in implementing the Oberon0 compiler. The modularity features of Simpl proved adequate for iterative development of language features. The code size of the Simpl implementation was similar to that of the other implementations.

6.2.3 Comparison with Silver

Silver (see also Section 4.5.3.3) is an attribute grammar system. The parser was implemented with the Copper parser generator that is based on the LALR(1) parsing algorithm. Compared to Simpl, Copper uses more intuitive representation for expressing rules for arithmetic operations (that is,

it supports left-recursive rules). However, Silver and Copper require that the developer describes structure of the AST of the language separately from the grammar and uses parser actions to manually build the AST.

The language processing tasks were implemented using attributes. Unlike Simpl, name lookup did not create explicit symbol table but instead used parameterized attributes to calculate list of identifiers visible by any given AST node. Name lookup was implemented by defining a reference attribute that pointed to the node referred to by the identifier. Silver implementation used forwarding to implement language features that could be expressed in terms of simpler features. For example, the *for* and *case* statements can be expressed in terms of the *while* and *if* statements, respectively. Forwarding allows to delegate the language processing to a simplified AST while e.g., error messages are still constructed based on the original program. Thus, Silver has good support for implementing syntactic sugar in a convenient and modular manner.

6.2.4 Comparison with JastAdd

JastAdd (see also Section 4.5.3.1) is an attribute grammar system that builds on object-oriented concepts. As JastAdd has no direct support for parsing, the challenge entry used the JFlex scanner generator and the Beaver parser generator. Like Silver, the JastAdd implementation does not construct explicit symbol table for name analysis and uses parameterized attribute `lookup(String)` instead.

Procedure lifting is implemented by traversing the AST of the program and generating a new AST where procedures, types and constants are moved to the top level. In each node, the lifting procedure consults attributes to determine the generated AST node. When generating code for the *for* and *case* statements, the generating method first uses attribute to generate the equivalent node using simpler constructs and then delegates the code generation to the generated node.

6.2.5 Comparison with Rascal

Rascal (see also Section 5.7.2) is a DSL and a toolset for language processing. The parser was implemented using Rascal's parser generator. The scannerless parser allowed for easy definition of modular grammar. Additionally, there was no need to refactor the grammar definition to satisfy needs of the parser and, instead, the grammar rules were defined to produce

optimal AST. Grammar descriptions were annotated with AST constructor names. This information was used later by the *implode* function to construct the AST from the concrete syntax tree.

Name and type analysis used the ability to add annotations to AST nodes. During name analysis, the uses of a name are annotated with references to declarations of the name. The type analysis does not add annotations, instead it uses the name info to determine the expected and actual types of expressions and checking that they are compatible. Source to source transformations were implemented using visiting statements that offer a convenient way to express AST transformations (see Section 5.7.2 for an example). In particular, the visiting creates new AST that is based on the old one but that that contains specified changes. The developer does not have to manually specify copying of the unchanged nodes.

Pretty-printing was implemented using the Box formatting language [38, 39]. The Box language mixes formatting directives (e.g., indenting or alignment) with Rascal expressions that output data to be formatted. C code generation was implemented in a simpler way using string templates. In Rascal, string templates can contain Rascal code and nested templates. Using explicit margin markers, they can produce reasonably well formatted output.

Like Simpl, Rascal implementation also included an Eclipse-based IDE for the Oberon0 language. The Rascal implementations was one of the smallest in the challenge in terms of lines of code (about 1400 lines of code, compared to 1800-2000 lines for the other implementations).

6.2.6 Comparison with OCaml

OCaml (Objective Caml) is a high-level functional language. The tool challenge entry made extensive use of type-driven transformers. Scanning and pretty-printing was implemented with combinator library Ostap. The pretty-printer was implemented in a generic way, abstracting the concrete syntax of the output into printing scheme. Thus, the same pretty-printer was used for generating both Oberon0 and C code.

Both the name and type analysis convert the AST into a format that contains, respectively, name and type annotations. The annotated trees have different types than the unannotated trees. Source to source transformations operated on the name-annotated AST.

The OCaml implementation relies heavily on the OCaml type system. In contrast, the Simpl implementation is simpler and relies less on advanced

concepts. In the OCaml implementation, analysis tasks transform the AST into an annotated form that has different type than the original AST. In the Simpl implementation, the AST is mutable and the analysis tasks simply modify the reference and type fields present in the AST. The combinator library Ostap used by the Ocaml implementation allows implementing more generic code generators than the Simpl library. For example, Ostap can be parameterized with concrete syntax so that the same code can be used both for pretty-printing Oberon0 code and for C code generation.

6.2.7 Comparison with Kiama

Kiama (see also Section 4.5.3.2) is a Scala library for language processing. Like in Simpl implementation, the AST was represented as Scala case classes. However, in the Kiama implementation the case classes were explicitly defined separately from the grammar description. The parser was implemented using the Scala parser library.

The name and type analysis tasks are implemented using attributes. The implementation details are similar to the Silver and JastAdd implementations. The source to source transformation is implemented with the term rewriting library. The C code generation is implemented in the same manner as with Simpl – first the Oberon0 AST is transformed to C AST, next the C AST is pretty-printed using Kiama's pretty-printing library. Despite using different methods for parsing, program analysis and program transformation the Simpl and Kiama implementations are very close in code size (slightly less than 2000 lines of code).

6.3 Measuring Code Metrics

In this section we discuss an experiment in which an implementation of a DSL with Simpl is compared with implementations of the same language using other DSL tools, as reported in previous work by Klint et al. [48].

6.3.1 Description of Experiment

In order to measure the effectiveness of the Simpl DSL tool we performed an experiment that compared code metrics of DSL implementations created using different DSL tools. Because creating implementations of a non-trivial DSL requires substantial amount of effort we decided to reuse

the results of an existing study. Klint et al. have compared six different implementations of a DSL named Waebric [48]. Three implementations – so called "vanilla" – were developed from scratch in Java, JavaScript, and C#. The other three implementations were written in the same languages but with the help of respective DSL tools: ANTLR, OMeta, and "M". By using both quantitative and qualitative analysis, it was shown that the DSL tools indeed do increase maintainability of DSL implementations. This was especially noticeable in terms of code size reduction of the DSL implementations compared to the vanilla versions.

Waebric [85] is a DSL for generating XHTML web sites. Besides the constructs to generate markup, it also has control flow statements, iteration statements, and user-defined functions with rudimentary ability to pass code blocks to functions. Figure 6.3 shows a Waebric program that generates a simple "Hello World" web site. From an implementation point of view, Waebric is a rather comprehensive language. It supports user-defined functions and scoped variables. Functions can also be defined as local. From a parsing point of view, Waebric represents a typical programming language. One nuance in Waebric's syntax is that the decision whether an identifier such as *title* should be parsed as a markup or a variable depends on the context and the surrounding tokens. Thus, a Waebric program cannot be tokenized using only regular expressions. This somewhat inconveniences parser generators that use separate lexing and parsing phase.

```
module homepage

site
  index.html: home("Hello World!")
end

def home(msg)
  html {
    head title msg;
    body echo msg;
  }
end
```

Figure 6.3: Example Waebric program

It must be noted that the goal of the Klint et al. paper [48] is to measure effectiveness of the DSL tools versus implementing DSLs without tool support. They do not draw any conclusions about the effectiveness of different tools in the comparison. However, we believe that comparing the code metrics achieved with different tools can also offer some insight on the

relative merits of the tools. Thus, we concentrated on the metrics of the tool-assisted implementations and ignored the metrics from "vanilla" implementations (i.e. the implementation not using a DSL tool or framework)[2]. One drawback of the study is that it only compares parser generators, not full-featured DSL toolkits with program transformation features (e.g., Xtext, MPS, Spoofax/IMP, Rascal, JastAdd). Thus, the comparison did not involve state of the art in language processing tools.

To compare Simpl with the DSL tools presented in [48], we implemented the Waebric DSL using the Simpl DSL tool. In order to verify that the Simpl implementation is correct, we applied the same 100-program test suite that was also used to verify implementations in the original paper. Therefore, the Simpl implementation was functionally equivalent to the other implementations. In order to make the results more comparable, we used Simpl to generate the parser and the AST classes and did not use the Kiama library for program transformation tasks. The grammar was implemented in the Simpl grammar description language (see Section 5.3) and the code generator was implemented using Scala's built-in XML library.

The program transformation code used Scala pattern matching facilities to process the AST. In the following discussion, the grammar and generator implementations will be referred to as *parse* and *eval* components, respectively. Quantitative analysis was based on a metrics suite proposed by Kuipers and Visser [53]. The suite evaluates the maintainability of software by measuring the volume, the structural complexity and the duplication[3] of the code. The same set of metrics was used in the original study by Klint et al.

The volume was measured in terms of non-comment lines of code (NCLOC) and number of units in the *parse* and *eval* components. For grammar, a non-terminal is considered to be one unit. For Scala code, unit is equated to a method. Unlike the other implementations in the comparison, the Simpl implementation does not separate code for checking the validity of AST from the *eval* component (the *eval* component also contains program checking functionality). Also, miscellaneous supporting code such as main objects and constant value definitions are counted as part of the *eval*[4].

[2]In any case, the "vanilla" implementations were bigger and more complex than their tool-assisted counterparts and thus would not have added interesting information to the comparison.

[3]Since the original study did not provide concrete figures for code duplication, we also left out this metric.

[4]It is unclear from the original paper whether the main function and other supporting code was included in the line count. Including the main program in the *eval* component

The structural complexity of the code was measured by determining the percentage of NCLOC in the units that have the cyclomatic complexity (CC) higher than 6. CC of the *parse* component was calculated by counting decision points of the non-terminal nodes as described in [70, 48]. Alternatives ("|"), optionals ("?"), and iteration closure operators ("*", "+") count as a decision point. Additionally, Scala code in return expressions adds to the CC count, if it contains decision points. For calculating CC of the *eval* component, we used the Cyvis[5] tool on the compiled Scala code. Since this tool operates on compiled bytecode, there is possibility of overestimating the CC with respect to the original Scala source code (in cases where Scala implicitly adds checks that are not present in the source file).

6.3.2 Results and Discussion

Table 6.4 presents the volume measurements of the implementations. Note that the Simpl implementation does not have a separate *check* component. Instead, the checking functionality is included in the *eval* component. It can be seen that numbers for the Simpl implementation are smaller than or similar to the best numbers of the other DSL implementations. Quantitatively, the Simpl implementation consists of 4 files, 142 units, and 563 NCLOC. The latter is especially remarkable as it is over 40% smaller than that of ANTLR – the next implementation in terms of the smallest NCLOC.

When comparing the Simpl and ANTLR implementations, we can make two observations. First, the Simpl implementation has significantly more units. This can be explained by the fact that the Simpl implementation makes extensive use of fragment rules (see Section 5.3), e.g., to give names to various separators and symbols (this also increases the line count of the grammar). Also, some non-terminal rules are split into two to produce more convenient AST classes. Secondly, the ANTLR implementation has a prelude that lists used tokens, Java imports and defines Java method that acts as an entry point to parsing. Additionally, the grammar rules contain code for building the AST. This boilerplate code is missing from the Simpl implementation.

Table 6.5 shows the percentage of NCLOC in units with CC > 6. The *parse* and *eval* components are shown separately. The *parse* component of the Simpl implementation has similar complexity to "M", lower than ANTLR

ensured that the Simpl implementation will not have an unfair advantage.

[5] `http://cyvis.sourceforge.net/`

Table 6.4: Volume metrics: number of files (#F), number of units (#U), and NCLOC (#N). The data for ANTLR, OMeta, and "M" is taken from [48]. The smallest totals for each category are underlined.

	Simpl			ANTLR			OMeta			"M"		
	#F	#U	#N	#F	#U	#N	#F	#U	#N	#F	#U	#N
Parse	1	94	137	1	52	151	4	72	195	3	84	382
AST	-	-	-	-	-	-	22	130	612	-	-	-
Check	3	48	426	2	73	294	8	37	333	9	26	430
Eval				1	29	377	-	-	-	7	91	1574
Misc				2	3	186	7	19	222	2	2	74
Total	4	142	563	6	157	1008	41	258	1362	21	203	2460

and OMeta. Although Simpl and ANTLR implementations had similar line count in the *parse* components, the complexity of the Simpl component is significantly lower. The main reason seems to be that the Simpl grammar description is refactored into large number of small and simple rules (especially fragment rules). The rules with the high CC count tend to be option rules that simply list the alternatives and thus take up small amount of the code size. On the other hand, in the ANTLR grammar the rules with high CC count also contain code for building up the AST, thus increasing line count and overall complexity. The complexity of the *eval* component is similar across the tools, although the Simpl implementation seems to have somewhat lower complexity than the ANTLR and "M" implementations.

Table 6.5: Percentage of NCLOC in units with cyclomatic complexity greater than 6.

	Simpl	ANTLR	OMeta	"M"
parse	12	20	26	10
eval	25	29	-	33

Overall, compared to the other implementations, the Simpl implementation of Waebric is comparable and in some aspects simpler and shorter. Part of it can be attributed to the simple and compact grammar description language of Simpl. The low volume and rather low complexity of the *eval* component can be attributed to the automatically generated class model for representing the AST and compactness of the Scala programming language.

6.4 Usability Evaluation

In this section we present a controlled usability experiment of Simpl.

6.4.1 Description of Experiment

Overview

Empirical evaluation of the Simpl toolkit was based on a controlled usability study that measured time that was required to implement a typical DSL by test subjects. The subjects were split into the "control group" and the "experimental group". The control group used a baseline DSL tool, and the experimental group used Simpl – the aim being to compare the added value of Simpl with respect to the baseline. We chose ANTLR as the tool for the control group because it most closely matches the non-functional requirements described in Chapter 3 and had a strong positive impact on code size in the Waebric experiment (see Section 6.3). ANTLR has functionality similar to Simpl – it includes a parser generator, a basic program transformation engine (tree rewriting) and a code generator (StringTemplate). The main difference is that ANTLR does not include an IDE generator.

The subjects were given a task of implementing a realistic, non-trivial DSL. We measured the time spent and compared the results of the experimental group with the control group. In addition, after completing the task, the subjects were asked to complete a questionnaire for evaluating usability of the tools. In order to control for the experience of the subjects, we divided the subjects into two categories based on the level of expertise: junior developers and senior developers.

The Subjects

The subjects for this experiment were five professional programmers working at Cybernetica AS. Three subjects were junior programmers with Bachelor's degree and some programming experience. Two subjects were senior programmers with Master's degree and experience in using language tools and implementing programming languages. The level of programming skill was roughly consistent between subjects in the same group[6]. The junior programmers had no previous experience in language processing tools. None of the subjects had any previous experience with Simpl or Scala. The

[6]This is subjective evaluation by project manager of the subjects

subjects did not receive training for use of the tools. Instead, they used user manuals and other online materials[7]. There were three developers in the experimental group and two developers in the control group.

Because the task was relatively labor-intensive, it could not be completed in a controlled lab environment. Instead, we opted for a setting where the test subjects completed the task unsupervised on their normal workplaces (i.e., using workstations they used for doing everyday work). The subjects were responsible for logging the time spent on various subtasks. Although this setting provides less accurate measurements than the controlled lab setting, it is more realistic because real DSLs are not implemented over a couple of hours in a specifically set up computer. The task included installing and learning the appropriate tools (Simpl or ANTLR).

The Task

All the subjects performed the same task, namely, implementing the SpamDetector example language described in Section 5.2. The subjects were given SpamDetector grammar in BNF format and simple example programs illustrating the language.

Since the goal was to compare DSL tools, the subjects needed to implement only the DSL-related parts of the SpamDetector system. In particular, the subjects were required to implement the following components:

- parser for the SpamDetector language;
- program checker that detects calls to undefined conditions;
- code generator that converts rule files to Java;
- basic program editor/IDE that supports syntax coloring, syntax checking, outline view, and hyper-linking (navigating from reference to definition of condition).

Regarding the code generator, the subjects were not required to implement the runtime code responsible for parsing E-mail messages, extracting message header fields, normalizing space in strings, dealing with various encodings etc. Instead, the subjects could make an assumption that there exists some kind of message-processing API and generate Java code that

[7]In this matter, ANTLR has advantage over Simpl because of large body of online tutorials and examples.

calls this API. The subjects using ANTLR were free to choose means for building the editor. In practice, they adapted an example program demonstrating the creation of basic IDE for the Java programming language.

The Questionnaire

After completing the implementation task, the subjects were presented a questionnaire that asked their opinions of the DSL tool they used. The questionnaire for evaluating DSL tools was based on Technology Acceptance Model (TAM). TAM was proposed by Davies et al. [12] to evaluate "how users come to accept and use a given technology." In particular, the focus is on factors such as perceived usefulness and perceived ease of use. Since we did not have enough subjects to develop and verify our own scales, we used the questions and scales constructed by Recker [72] for assessing business process modeling languages (... (the scales are based on TAM). The questions were reused almost verbatim, the phrase "business process modeling" was replaced with "ANTLR/Simpl DSL tool", depending on whether the subject used ANTLR or Simpl for completing the assignment. Users rated the questions on a scale of 1 to 5 where 1 designated "disagree completely", 2 "somewhat disagree", 3 "neutral", 4 "somewhat agree", and 5 "agree completely". The questionnaire, shown in table 6.6, measured the following aspects of usability:

- Perceived Usefulness (PU) – the degree to which a person believes that using a particular tool would enhance his or her job performance;
- Perceived Ease of Use (PEOU) – the degree to which a person believes that using a particular tool would be free of effort;
- Confirmation (Con) – the extent to which a person's initial expectations were confirmed;
- Satisfaction (Sat) – the extent to which a person is satisfied after adopting the particular tool; and
- Intention to Use (ItU) – the extent to which a person intends to use or continue to use a particular tool.

6.4.2 Experiment Results

Table 6.7 shows the amount of time the subjects spent on implementing various subtasks. The subtask "bug fixes" contains general activities not

Table 6.6: The usability questionnaire.

Topic	Question
PU	Overall, I find Simpl/ANTLR useful for implementing DSLs.
	I find Simpl/ANTLR useful for achieving the purpose of my DSL implementation.
	I find Simpl/ANTLR useful for meeting my DSL implementation objectives.
PEOU	I find it easy to implement DSLs in the way I intended using Simpl/ANTLR.
	I find learning Simpl/ANTLR for DSL implementation is easy.
	I find implementing DSLs using Simpl/ANTLR is easy.
Con	Compared to my initial expectations, the ability of Simpl/ANTLR to help me implement DSLs was much better than expected.
	Compared to my initial expectations, the ability of Simpl/ANTLR to help me achieve the purpose of my DSL implementation was much better than expected.
	Compared to my initial expectations, the ability of Simpl/ANTLR to help me meet my DSL implementation objectives was much better than expected.
Sat	I feel extremely contented about my overall experience of using Simpl/ANTLR for DSL implementation.
	I feel extremely satisfied about my overall experience of using Simpl/ANTLR for DSL implementation.
	I feel extremely delighted about my overall experience of using Simpl/ANTLR for DSL implementation.
ItU	If I have access to Simpl/ANTLR I expect I will continue to use it for implementing DSLs.
	My intention is to continue to use Simpl/ANTLR for implementing DSLs.
	In the future, I would prefer to continue to use Simpl/ANTLR instead of using another DSL tool for implementing DSLs.

related to any specific subtask (such as final testing and fixing of the language implementation).

It must be noted that the "set-up and learning" task does not accurately

Table 6.7: Implementation times (in hours).

	Junior developers			Senior developers	
	J1	J2	J3	S1	S2
	Simpl	ANTLR	Simpl	Simpl	ANTLR
Set-up and learning	4		7	3	8.5
Parser	16	21	6	4	10
Code generator	11	15	10	5	10.5
Editor	#	45	7	4.5	56*
Bug fixes	5			2	
Total	**36**	**81**	**30**	**18.5**	**85**

#Subject J1 did not record the time spent on the editor and parser separately.
** Subject S2 did not actually complete the editor because of insufficient time available. After spending 16 hours on editor, he estimated the remaining amount of work to be 40 hours.*

reflect the time spent on learning the tools. It rather indicates the time spent on the installation and studying the existing examples before starting the implementation. According to the feedback, most of the learning time is included in the actual tasks – the subjects referred to user manual only when they encountered some particular problem. Additionally, Simpl users also spent some time learning the Scala programming language (none of the subjects had previous experience with Scala).

We recognize that there were too few subjects in the study to draw any definite conclusions, however, based on the numbers in Table 6.7, some results should be highlighted. Firstly, the Simpl users were consistently faster than the ANTLR users. For instance, the Simpl junior subjects were able to implement the problem significantly faster than the senior ANTLR user. Much of the success of Simpl can be attributed to the editor – building an editor in Simpl is relatively easy task for a person with even beginner level experience in Java or Scala. Secondly, Simpl also offers good usability for creating the non-visual parts of a language implementation. Table 6.7 shows that the time spent on writing the parser and the code generator are somewhat smaller for subjects who used Simpl.

Table 6.8 shows answers to questionnaire completed after finishing the study. Notably, the average Simpl user rates all the aspects of using Simpl equal or above 4.0 in the scale of 1 to 5 whereas the average ANTLR user rates most aspects close to 3. Given the presented results, it is evident that Simpl can be efficiently used to develop a full DSL, including a parser, a

code generator, and an editor and it might have certain advantages over ANTLR.

Table 6.8: Answers to the questionnaire.

Area	Simpl	ANTLR
Perceived Usefulness	4.7	4.0
Perceived Ease of Use	4.1	3.2
Confirmation	4.4	3.3
Satisfaction	4.0	3.3
Intention to Use	4.3	3.0

6.4.3 Experiment Validity

Next, we discuss threats to validity. The metrics experiment (Section 6.3) is essentially a continuation of a previous study, we refer the reader to the validity section of the respective paper [48]. This subsection focuses on validity of the usability study (Section 6.4).

Construct validity determines whether the measured values really correspond to usability and usefulness. For the SpamDetector DSL we measured time spent on learning Simpl and implementing a realistic DSL. This correlates well with tool's usefulness in practical world as the point of using a DSL tool is to save time (and hence money) when implementing DSLs. As for questionnaire, the proposed questions are based on TAM model that has been validated by numerous researchers (see [72] for literature review).

Internal validity determines whether our conclusion is valid with respect to the measured data. Although the test subjects were chosen carefully, their skill level was determined mainly based on number of years of experience. Thus, it is possible that some of the programmers in the group were weaker in some aspects compared to the others. However, as noted earlier, rather than trying to draw profound conclusions from the results we are more interested in overall usability of Simpl compared to the other DSL tool.

External validity determines whether it is possible to generalize the results to a larger population. The current study was conducted in a setting very similar to commercial software development. The task was fairly long (in the range of a man-week), the subjects implemented the task at their normal work environment during the normal working hours, and the subjects had freedom to make some of the design decisions (e.g. design the

API for the message processing). Also, all the subjects were professional programmers. The fact that they all worked for the same company, might have somewhat influenced the results. On the other hand, the target group of Simpl is mainly skilled programmers, thus the subjects fit within the scope of the study.

In addition to the discussed validity threats, the small number of participants must be taken into account. This was mainly caused by the fact that the requirements for the subjects' skills were quite high. As the task was time-consuming, it would have been too costly and difficult to find senior (professional) developers who would be willing to learn Simpl and spend non-negligible amount of time in this controlled study. At the same time, research by Hwang and Salvendy [34] indicates that 5-10 participants is enough to identify major usability issues.

We acknowledge that given the small number of subjects, the conclusions of the study do not have statistical significance. However, the study has provided us some valuable information about the relative strengths and weaknesses of the tools. Furthermore, it gave an indication about the time it takes for a programmer to learn the Simpl tool and develop a real DSL.

6.5 Discussion

This chapter presented several evaluations of the Simpl DSL tool. First, we reimplemented two languages from the Customs Engine system using Simpl. The first language was Burula – a fairly complex DSL with difficult grammar. We created a parser and an IDE for the Burula language. The main difficulty was implementing support for syntactic whitespace[8]. The resulting implementation conforms to the original Burula language and the IDE is currently being used by the developers of the Customs Engine system. The second language was a technical DSL for describing document editing screens. In this case we changed the syntax of the language by removing the parentheses surrounding every expression, thus reducing visual clutter. The time spent on implementing the parser and the code generator the DSL showed that Simpl is suitable for creating simple code generators to reduce the amount of boilerplate code in an enterprise system. From the non-functional point of view, the implementations created with Simpl can be embedded into build process and the system itself, similar to the original hand-crafted implementations.

[8]However, this is difficult with majority of parser generators.

Second, Simpl took part in a tool challenge where the Oberon0 language was implemented using different language tools. At the time of the challenge Simpl did not have support for Kiama language processing library and thus lacked direct support for program transformation tasks. However, Simpl was able to fulfill the challenge task that required implementation of the Oberon0 language in an incremental manner. The code size of the Simpl implementation is similar to the other implementations (except Rascal implementation that was somewhat smaller).

Third, we implemented a fairly complex DSL, Waebric, using Simpl and compared the code metrics to previously published implementations that used other DSL tools: ANTLR, OMeta and "M". In this comparison, Simpl yielded the lowest NCLOC measure and the code complexity (measured as percentage of NCLOC in units with cyclomatic complexity greater than 6) was slightly lower than the other tools.

The fourth evaluation was based on a controlled usability study that compared Simpl to popular DSL tool ANTLR. The subjects were junior and senior professional programmers who implemented a realistic, non-trivial DSL. After completing the task, the subjects were presented with a questionnaire that evaluated the tool's ease of use and the subject's satisfaction with the tool. The results showed that the Simpl users implemented the task faster and rated the usability and satisfaction with the used tool higher than the ANTLR users. From these tests we conclude that Simpl does offer usability advantages over ANTLR.

Each of the above evaluations arguably has its own threats to validity. The first evaluation involves re-implementation of parts of a real-life system. It is a "post-mortem" case study and has the usual pros and cons of such studies. Also, the comparison here is against a "no DSL tool" baseline, and merely shows the benefits of Simpl over little or no DSL tool support. The second evaluation pitches Simpl against state-of-the-art tools. However, this evaluation is not based on concrete measures, but is rather qualitative given the nature of the tools challenge. The third evaluation is quantitative (based on metrics) and involves state of the art tools, as of the beginning of this study. However it suffers from known limitations of software size metrics, which are not fully reliable in terms of capturing usefulness of a development tool. The final evaluation is a controlled experiment with a limited set of subjects. Like the first evaluation, it demonstrates the benefits of Simpl over limited DSL tool support. Despite the limitations of each study, we believe that overall the evaluations show that Simpl provides clear benefits over limited DSL development support and comparable benefits to state-of-the-art tools, when putting aside the number of person-months put into the various existing tools.

CHAPTER 7

CONCLUSION

7.1 Contributions of this Thesis

The main contributions of this work are the following.

- We provided an analysis of the requirements that DSL tools should satisfy so that they fit well into the overall development process. The result was a concise list of requirements that can be applied to evaluate DSL tools for suitability for enterprise software development.
- We analyzed state of the art in DSL tools with respect to the requirements. We reached the conclusion that although single-purpose tools usually satisfy the non-functional requirements, full-featured DSL toolkits tend to assume a particular architecture and are usually not suited for creating DSL implementations that can be embedded into a larger system.
- We designed and implemented Simpl DSL toolkit that is aimed at enterprise software development and, in particular, creating DSL implementations that have low footprint and can be embedded into a larger system. Simpl is based on existing tools, such as ANTLR parser generator, Eclipse IDE framework and Scala programming language. On top of these tools we add a parser generator that automatically generates classes for representing AST of the program, pretty-printing library, lightweight IDE framework, and bindings for StringTemplate template engine and Kiama language processing library. Simpl was developed at Cybernetica AS. The author was architect of the system and contributed to design and implementation of all the components.

- We evaluated the usability of Simpl in a multi-pronged way. First, we reimplemented some representative DSLs from an existing EIS. Simpl was capable of implementing both a complex and a simple DSL. The Simpl implementation is already used by systems analysts to write document processing rules. Second, we took part of a tool challenge that compared different implementations of Oberon0 programming language. This challenge highlighted different approaches to the same problem. Also, the Simpl implementation was comparable in code size to other implementations. Third, we extended an existing research that compared code metrics of different implementations of a fairly complex DSL. The Simpl implementation had the smallest code size and complexity measures. Fourth, we performed a controlled usability study where subjects implemented the same DSL using either Simpl or a reference DSL tool (ANTLR). The subjects using Simpl spent less time on implementation and were more satisfied with the tool.

7.2 Future Work

The current implementation of Simpl is just a first step. The next step is to use Simpl in implementing enterprise projects and act on the feedback. Because Simpl is targeted at enterprise developers, the main validation must come from enterprise developers and actual projects. We are looking forward to using Simpl in the upcoming projects at Cybernetica AS and hopefully in other companies.

An interesting future topic for Simpl is language modularity and language embedding. Language modularity allows to reuse separately developed language modules, such as support for arithmetic expressions or data queries. Language embedding combines two languages in such a way that statements or expressions of the embedded language can be used as statements or sub-expressions in the embedding language. For example, a HTML document can contain embedded CSS style sheets. Language embedding and language modularity can be useful in a product line setting where the developer can create a "base" DSL and customize it in each single product. The customization can be either adding new constructs or filling in holes in the original language description. Many practical DSLs are extensible by using low-level code snippets in the DSL programs. This allows the developer to keep the DSL simple and to implement the seldom-occurring but inevitable corner cases in the low-level programming language (such as

Java) that has support for the necessary operations. Ideally, the low-level code would also be checked (both syntactically and semantically) and the errors reported based on the original DSL program. When editing such DSL programs, the low-level code snippets should make use of full power of modern IDEs, such as Eclipse. There has been considerable work in language embedding (see [83, 101, 5, 33, 41] for some examples), but much work still remains in order to consolidate results in this field and to validate them in industrial settings. The other side of the embedding coin is embedding of DSLs into mainstream languages. A good example of this technique is Microsoft's LINQ query language [57] that is embedded in the C# language. However, extending the Java or C# compiler with custom syntax is only available to the compiler developers because the compilers do not support plugins for adding custom syntax. This means that regular developers must use work-arounds such as encoding small DSL programs in annotations or trying to process and/or type-check string literals [25, 1]. Extending and standardizing the compiler plugin mechanisms would open up new areas of language innovation.

An interesting topic is the relationship between innovative language tools and their adoption in industrial settings. When implementing the Customs Engine system I experienced firsthand that there tends to be a resistance to introducing new ideas to the workplace. For a more global example, functional programming languages have been, for a long time, used by researches and enthusiasts. However, only in the recent decade there have been a rising interest towards functional languages from the industry. Interestingly, this interest is resolved not by adopting existing functional languages but by incorporating functional features into existing mainstream programming languages such as Java, C#, C++. Similar process seems to be happening in the field of DSLs. Although DSLs, especially internal DSLs, are gaining popularity, the most-used tools are the simple ones that do not take advantage of the current research results. In the same vein, Simpl aims for simplicity and uses tools familiar (or at least not too foreign) to a typical developer. An interesting practical research area would be to investigate what are the main barriers to adoption of new language processing paradigms in industry and how to overcome these barriers by making DSL development technologies more accessible to mainstream software developers.

Bibliography

[1] Annamaa, A., Breslav, A., Kabanov, J., Vene, V.: An Interactive Tool for Analyzing Embedded SQL Queries. In: Ueda, K. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 6461, pp. 131–138. Springer (2010)

[2] ANTLR Home Page. Available online at `http://www.antlr.org/`

[3] Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving Extensibility Through Product-Lines and Domain-Specific Languages: a Case Study. ACM Transactions on Software Engineering and Methodology 11(2), 191–214 (2002)

[4] Brand, M.G., de Jong, H., Klint, P., Olivier, P.A.: Efficient Annotated Terms. Software – Practice & Experience 30, 259–291 (2000)

[5] Bravenboer, M., Groot, R., Visser, E.: MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) Generative and Transformational Techniques in Software Engineering (GTSSE), Lecture Notes in Computer Science, vol. 4143, pp. 297–311. Springer (2006)

[6] Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. Science of Computer Programming 72(1-2), 52–70 (2008)

[7] Chandra, S., Richards, B., Larus, J.R.: Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols. IEEE Transactions on Software Engineering 25(3), 317–333 (1999)

[8] Charles, P., Fuhrer, R.M., Sutton, J.S.M.: IMP: a Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse. In: ASE '07:

Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. pp. 485–488. ACM, New York, NY, USA (2007)

[9] Christerson, M., Clifford, S.: Intentional Software. Available online at http://channel9.msdn.com/Series/DSL-DevCon-2009 (2009), presentation at DSL DevCon

[10] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and Programming in Rewriting Logic. Theoretical Computer Science 285(2), 187–243 (2002)

[11] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 System. In: Nieuwenhuis, R. (ed.) Rewriting Techniques and Applications (RTA 2003). Lecture Notes in Computer Science, vol. 2706, pp. 76–87. Springer (June 2003)

[12] Davis, F.D., Bagozzi, R.P., Warshaw, P.R.: User Acceptance of Computer Technology: a Comparison of Two Theoretical Models. Management Science 35(8), 982–1003 (August 1989)

[13] van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices 35(6), 26–36 (2000)

[14] Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. Available online at http://www.onboard.jetbrains.com/is1/articles/04/10/lop/ (Nov 2004)

[15] Eclipse Project Home Page. Available online at http://eclipse.org/

[16] EMFText Home Page. Available online at http://www.emftext.org/

[17] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? Available online at http://www.martinfowler.com/articles/languageWorkbench.html (May 2005)

[18] Fowler, M.: Projectional Editing. Available online at http://www.martinfowler.com/bliki/ProjectionalEditing.html (Jan 2008)

[19] Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley, Upper Saddle River, N.J. (2011)

[20] Freudenthal, M.: Domain-Specific Languages in a Customs Information System. IEEE Software 27(2), 65–71 (Mar 2010)

[21] Freudenthal, M.: Using DSLs for developing enterprise systems. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications. pp. 11:1–11:7. LDTA '10, ACM, New York, NY, USA (2010)

[22] Freudenthal, M.: Implementing Oberon0 Language with Simpl DSL Tool. Tech. Rep. T-4-18, Cybernetica AS (2013), available online at `http://research.cyber.ee/`

[23] Freudenthal, M., Pugal, D.: Simpl: a Toolkit for Rapid DSL Implementation. In: Proceedings of the 12th Symposium on Programming Languages and Software Tools - SPLST'11. Institute of Cybernetics (October 2011)

[24] Friese, P., Efftinge, S., Köhnlein, J.: Build Your Own Textual DSL with Tools from the Eclipse Modeling Project. Available online at `http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html` (Apr 2008)

[25] Garcia, M.: Compiler Plugins Can Handle Nested Languages: AST-Level Expansion of LINQ Queries for Java. In: Norrie, M.C., Grossniklaus, M. (eds.) Object Databases, Lecture Notes in Computer Science, vol. 5936, pp. 41–58. Springer (2010)

[26] Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., Tolvanen, J.P.: DSLs: the Good, the Bad, and the Ugly. In: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications. pp. 791–794. OOPSLA Companion '08, ACM, New York, NY, USA (2008)

[27] den Haan, J.: Roles in Model Driven Engineering. `http://www.theenterprisearchitect.eu/archive/2009/02/04/roles-in-model-driven-engineering` (Feb 2009)

[28] Hedin, G.: An Introductory Tutorial on JastAdd Attribute Grammars. In: Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering (GTSSE). Lecture Notes in Computer Science, vol. 6491, pp. 166–200. Springer (2011)

[29] Hedin, G., Magnusson, E.: JastAdd — an Aspect-Oriented Compiler Construction System 47(1), 37–58 (2003)

[30] Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF – Reference Manual. ACM SIGPLAN Notices 24(11), 43–75 (Nov 1989)

[31] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications. pp. 114–129. ECMDA-FA '09, Springer (2009)

[32] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap Between Modelling and Java. In: Proceedings of the Second International Conference on Software Language Engineering. pp. 374–383. SLE'09, Springer (2010)

[33] Hudak, P.: Modular Domain Specific Languages and Tools. In: Proceedings of the 5th International Conference on Software Reuse. pp. 134–142. ICSR '98, IEEE Computer Society, Washington, DC, USA (1998)

[34] Hwang, W., Salvendy, G.: Number of People Required for Usability Evaluation: the 10±2 rule. Communications of the ACM 53(5), 130–133 (May 2010)

[35] IMP Home Page. Available online at `http://www.eclipse.org/imp/`

[36] Intentional Software Home Page. Available online at `http://intentsoft.com/`

[37] JastAdd Home Page. Available online at `http://jastadd.org/`

[38] de Jonge, M.: A Pretty-Printer for Every Occasion. In: Ferguson, I., Gray, J., Scott, L. (eds.) Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000). pp. 68–77. University of Wollongong, Australia (Jun 2000)

[39] de Jonge, M.: Pretty-Printing for Software Reengineering. In: Proceedings of the International Conference on Software Maintenance (ICSM 2002). pp. 550–559. IEEE Computer Society Press (Oct 2002)

[40] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-Like Transformation Language. In: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 719–720. ACM Press, New York (2006)

[41] Kamin, S.N.: Research on Domain-Specific Embedded Languages and Program Generators. Electronic Notes in Theoretical Computer Science 14, 149–168 (1998)

[42] Kärnä, J., Tolvanen, J.P., Kelly, S.: Evaluating the Use of Domain-Specific Modeling in Practice. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09). pp. 14–20 (Oct 2009)

[43] Kats, L.C.L., Kalleberg, K.T., Visser, E.: Domain-Specific Languages for Composable Editor Plugins. In: Ekman, T., Vinju, J. (eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009). Electronic Notes in Theoretical Computer Science, vol. 253, pp. 149–163. Elsevier Science Publishers (April 2009)

[44] Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010. pp. 444–463. ACM, Reno/Tahoe, Nevada (2010)

[45] Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment. In: Proceedings of 8th Intl. Conference on Advanced Information Systems Engineering (CAiSE). Lecture Notes in Computer Science, vol. 1080, pp. 1–21 (1996)

[46] Kelly, S., Lyytinen, K., Rossi, M., Tolvanen, J.P.: MetaEdit+ at the Age of 20. In: Bubenko, J., Krogstie, J., Pastor, O., Pernici, B., Rolland, C., Sølvberg, A. (eds.) Seminal Contributions to Information Systems Engineering, pp. 131–137. Springer (2013)

[47] Kiama Home Page. Available online at `http://code.google.com/p/kiama/`

[48] Klint, P., van der Storm, T., Vinju, J.: On the impact of DSL tools on the maintainability of language implementations. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications. pp. 10:1–10:9. LDTA '10, ACM, New York, NY, USA (2010)

[49] Klint, P., van der Storm, T., Vinju, J.: EASY Meta-programming with Rascal. In: Fernandes, J., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III International Summer School, Lecture Notes in Computer Science, vol. 6491, pp. 222–289. Springer (2011)

[50] Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems 2(2), 127–145 (Jun 1968)

[51] Kosar, T., Mernik, M., Carver, J.C.: Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. Empirical Software Engineering 17, 276–304 (2012)

[52] Kosar, T., Mernik, M., Črepinšek, M., Henriques, P.R., da Cruz, D.C., Pereira, M.J.V., Oliveira, N.: Influence of Domain-Specific Notation to Program Understanding. In: Proceedings of International Multiconference on Computer Science and Information Technology. pp. 675–682. IMCSIT '09 (2009)

[53] Kuipers, T., Visser, J.: Maintainability Index Revisited. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07). Special session on System Quality and Maintainability (SQM) (2007)

[54] Langlois, B., Jitia, C.E., Jouenne, E.: DSL Classification. In: The 7th OOPSLA Workshop on Domain-Specific Modeling (Oct 2008)

[55] LDTA 2011 Tool Challenge Home Page. Available online at `http://ldta.info/2011/tool.html` (2011)

[56] Leijen, D., Meijer, E.: Parsec: Direct Style Monadic Parser Combinators for the Real World. Tech. Rep. UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001)

[57] LINQ (Language-Integrated Query). Available online at `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`

[58] Eclipse M2M Project Home Page. Available online at `http://m2m.eclipse.org/`

[59] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM 3(4), 184–195 (Apr 1960)

[60] Mernik, M., Heering, J., Sloane, T.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys 37(4), 316–344 (dec 2005)

[61] Meta Programming System Home Page. Available online at http://www.jetbrains.com/mps/

[62] Northrop, L.M., Clements, P.C.: A Framework for Software Product Line Practice, Version 5.0. Tech. rep., Software Engineering Institute (Jul 2007), available online at http://www.sei.cmu.edu/productlines/tools/framework/

[63] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An Overview of the Scala Programming Language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2004)

[64] O'Grady, S.: The RedMonk Programming Language Rankings: January 2013. Available online at http://redmonk.com/sogrady/2013/02/28/language-rankings-1-13/ (Feb 2013)

[65] OOMEGA Home Page. Available online at http://www.oomega.net/

[66] Özgür, T.: Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling in the context of Model-Driven Development. Master's thesis, Blekinge Institute of Technology (Jan 2007)

[67] Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software Practice and Experience 25(7), 789–810 (1994)

[68] Parr, T.J.: Enforcing Strict Model-View Separation in Template Engines. In: Proceedings of the 13th International Conference on World Wide Web. pp. 224–233 (2004)

[69] Pfeiffer, M., Pichler, J.: A Comparison of Tool Support for Textual Domain-Specific Languages. In: Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling (Oct 2008)

[70] Power, J.F., Malloy, B.A.: A Metrics Suite for Grammar-Based Software. Journal of Software Maintenance and Evolution: Research and Practice 16(6), 405–426 (2004)

[71] Rascal Home Page. Available online at `http://www.rascal-mpl.org/`

[72] Recker, J., Rosemann, M.: Understanding the Process of Constructing Scales Inventories in the Process Modelling Domain. In: Proceedings of the 15th European Conference on Information Systems. pp. 2014–2025. St. Gallen, Switzerland (Jul 2007)

[73] Rosu, G., Serbanuta, T.F.: An Overview of the K Semantic Framework. The Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)

[74] Schmidt, C., Kastens, U., Cramer, B.: Using DEViL for Implementation of Domain-Specific Visual Languages. In: Proceedings of the 1st Workshop on Domain-Specific Program Development (DSPD) (July 2006)

[75] Schwerdfeger, A., Van Wyk, E.: Verifiable Composition of Deterministic Grammars. ACM SIGPLAN Notices - PLDI '09 44(6), 199–210 (Jun 2009)

[76] Serbanuta, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Rosu, G.: The K Primer (Version 2.5). Tech. Rep. K2012 (2012)

[77] Silver Home Page. Available online at `http://melt.cs.umn.edu/silver/`

[78] Simonyi, C., Christerson, M., Clifford, S.: Intentional software. ACM SIGPLAN Notices - Proceedings of the 2006 OOPSLA Conference 41(10), 451–464 (Oct 2006)

[79] Sloane, A.M.: Lightweight Language Processing in Kiama. In: Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering. pp. 408–425. GTTSE'09, Berlin, Heidelberg (2011)

[80] Sloane, A.M., Kats, L.C., Visser, E.: A Pure Object-Oriented Embedding of Attribute Grammars. Electronic Notes in Theoretical Computer Science 253(7), 205–219 (2010), proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)

[81] Spoofax Language Workbench Home Page. Available online at `http://www.spoofax.org/`

[82] Sprinkle, J., Mernik, M., Tolvanen, J.P., Spinellis, D.: Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software 26(4), 15–18 (2009)

[83] Standish, T.A.: Extensibility in Programming Language Design. ACM SIGPLAN Notices - Special Issue on Programming Language Design 10(7), 18–21 (Jul 1975)

[84] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley, Boston, MA, second edn. (2009)

[85] van der Storm, T.: WAEBRIC: a Little Language for Markup Generation. Tech. rep. (July 2009), available online at `http://code.google.com/p/waebric`

[86] Swierstra, S.D., Chitil, O.: Linear, Bounded, Functional Pretty-Printing. Journal of Functional Programming 19(1), 1–16 (Jan 2009)

[87] Thibault, S.A., Marlet, R., Consel, C.: Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation. IEEE Transactions on Software Engineering 25(3), 363–377 (may 1999)

[88] van Deursen, A., Klint, P.: Little Languages: Little Maintenance? Journal of Software Maintenance: Research and Practice 10(2), 75–92 (1998)

[89] Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Science of Computer Programming 75(1–2), 39–54 (January 2010)

[90] Van Wyk, E., Schwerdfeger, A.: Context-Aware Scanning for Parsing Extensible Languages. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE '07. pp. 63–72. ACM Press (Oct 2007)

[91] Vasudevan, N., Tratt, L.: Comparative Study of DSL Tools. Electronic Notes in Theoretical Computer Science 264(5), 103–121 (Jul 2011)

[92] Visser, E.: Program Transformation with Stratego/XT. Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. Tech. Rep. UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University (2004)

[93] Voelter, M.: Language and IDE Modularization, Extension and Composition with MPS. In: Proceedings of the Generative and Transformational Techniques in Software Engineering Summer School (GTTSE'2011). Lecture Notes in Computer Science, vol. 7680, pp. 383–430 (Jul 2011)

[94] Voelter, M.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace Independent Publishing Platform (2013)

[95] Wadler, P.: A Prettier Printer. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming. pp. 223–244. Palgrave Macmillan (1998)

[96] Wile, D.S.: Supporting the DSL Spectrum. Journal of Computing and Information Technology 9(4), 263–287 (2001)

[97] Wirth, N.: Compiler Construction. Addison-Wesley (1996)

[98] Xtext Home Page. Available online at `http://www.eclipse.org/Xtext/`

[99] Developer Productivity Report 2012. Available online at `http://zeroturnaround.com/rebellabs/devs/developer-productivity-report-2012/`

[100] Zhu, N., Grundy, J., Hosking, J., Liu, N., Cao, S., Mehra, A.: Pounamu: A Meta-Tool for Exploratory Domain-Specific Visual Language Tool Development. Journal of Systems and Software 80(8), 1390–1407 (2007)

[101] Zingaro, D.: Modern Extensible Languages. Tech. Rep. 47, McMaster University SQRL (2007)

ACKNOWLEDGMENTS

This research is partly funded by the EU Regional Development Funds through the Estonian Competence Centre Program (Software Technology and Applications Competence Centre).

The author wishes to thank Arne Ansper, Madis Janson and the rest of the Customs Engine team for the collaboration that served as original motivation to this work. Marlon Dumas provided invaluable help in guiding the research and helping to express it in research papers and this thesis.

KOKKUVÕTE (SUMMARY IN ESTONIAN)

SIMPL: VALDKONNASPETSIIFILISTE KEELTE LOOMISE TÖÖRIIST ETTEVÕTTETARKVARA ARENDAMISEKS

Valdkonnaspetsiifilised programmeerimiskeeled (*domain specific language*, DSL) on keeled, mis on välja töötatud kasutamiseks mingis konkreetses rakendusvaldkonnas. Spetsialiseerumine võimaldab DSLis kasutada konstruktsioone, mis sobivad hästi antud valdkonna mõistete esitamiseks. DSLide näited on Unixi *make* ehitusskriptid (*Makefile*), regulaaravaldised, HTML ja GraphViz (graafide kirjeldamine). DSLide kasutamine annab võrreldes üldotstarbeliste keeltega mitmeid eeliseid nagu näiteks kõrgem tarkvaraarenduse efektiivsus ning paindlikum ja hästi hooldatav lõpptulemus. Samuti saavad DSLide abil tarkvaraarenduses osaleda ka isikud, kelle tehnilised oskused ei ole piisavad üldotstarbelistes keeltes programmeerimiseks, näiteks süsteemianalüütikud, lõppkasutajad jne. Teisest küljest kaasnevad DSLide kasutamisega ka kulutused DSLide välja töötamiseks ning haldamiseks. DSL-põhist tarkvaraarendust saab muuta efektiivsemaks, kasutades DSLide realiseerimiseks spetsiaalseid tööriistu.

Käesoleva väitekirja fookuses on kuluefektiivne DSLide kasutamisel põhinev ettevõttetarkvara arendus. Ettevõtteinfosüsteemid (EIS) realiseeritakse tüüpiliselt raamistike ja valmiskomponentide abil. Seega peab olema või-

malik pakendada DSLi realisatsioon moodulina, mida on võimalik välja kutsuda kas ehitussüsteemist või EISist endast. DSLi realiseerimise tööriist peab sobima kasutamiseks ka tarkvaraarendajatele, kellel ei ole kogemusi programmeerimiskeelte ja neid toetavate vahendite arendamiseks.

Töö olulisemad väited on järgmised. Esiteks, ettevõttetarkvara arendamisel on oma spetsiifika, mis seab nõudeid DSLidele ning nende realiseerimiseks kasutatavatele tööriistadele. Teiseks, enamik populaarseid tööriistu, eriti integreeritud tööriistu, mis katavad ära kogu DSLi realiseerimiseks vajaliku tegevuste spektri, ei rahulda vähemalt osaliselt neid nõudeid. Kolmandaks, me demonstreerime, et on võimalik töötada välja DSL tööriist, mis on sobiv EIS arendamiseks ning mis pakub olemasolevate tööriistadega võrreldavat kasutusmugavust.

Väitekiri koosneb seitsmest peatükist. Esimene peatükk juhatab töö sisse, andes ülevaate DSLidest ning tutvustades töö ülesehitust.

Teine peatükk kirjeldab juhtumianalüüsi, kus väitekirja autori osalusel kasutati tollideklaratsioonide töötlemise infosüsteemi loomisel DSL-põhist tarkvaraarendust. Peatükk annab ülevaate DSLide rollist süsteemi arendamisel ning kirjeldab DSLide kasutamisel saadud kasust nii süsteemi välja töötamisel kui ka hilisemal hooldamisel. Peatükk kirjeldab ka edasise töö aluseks olevat arendusstsenaariumit.

Kolmas peatükk analüüsib arendusstsenaariumit ning töötab välja komplekti nõudeid, millele peaksid vastama EIS arendamiseks kasutatavad DSL tööriistad.

Neljas peatükk annab ülevaate olemasolevatest DSL tööriistadest ning evalveerib nende vastavust eelmises peatükis toodud nõuetele.

Viies peatükk kirjeldab DSL tööriista Simpl, mis sobib hästi EIS arenduseks. Simpl võimaldab luua DSL realisatsioone, mida on hea integreerida suurema EIS koosseisu ning mis on sobiv kasutamiseks tarkvaraarendajatele, kes ei ole programmeerimiskeelte spetsialistid. Simpl töötati välja ettevõttes Cybernetica AS. Väitekirja autor oli arhitekti rollis ning osales kõigi komponentide projekteerimises ja programmeerimises.

Kuues peatükk kirjeldab nelja eksperimenti, millega evalveeriti tööriista Simpl kasutusmugavust ning sobivust EIS arenduseks. Esimeses eksperimendis realiseeriti Simpli abil teises peatükis kirjeldatud tolliinfosüsteemis kasutatud DSLe. Teises eksperimendis realiseeriti Simpl abil programmeerimiskeele Oberon0 kompilaator ning võrreldakse tulemust sama keele kompilaatoritega, mis on loodud teiste tööriistade abil. Kolmandas eksperimendis võrreldi erinevate tööriistadega loodud sama DSLi realisatsioonide

lähtekoodi meetrikaid. Neljanda eksperimendina viidi läbi kasutusmugavuse test, kus katsealused realiseerisid sama DSLi, kasutades kas tööriista Simpl või võrdlusalust tööriista. Mõlemal juhul mõõdeti kulunud aega ning kasutajate rahulolu kasutatud tööriistaga.

Seitsmendas peatükis esitatakse töö kokkvõte ning kirjeldatakse edasisi uurimissuundi.

CURRICULUM VITAE

Personal data

Name	Margus Freudenthal
Birth	July 31st, 1976
Citizenship	Estonian
Languages	Estonian, English, Russian, Finnish
Contact	+372 50 79 273 margus@cyber.ee

Education

2008–	University of Tartu, Ph.D. candidate in Computer Science
1999–2001	Tallinn Technical University, MSc in Computer Science
1994–1999	Tallinn Technical University, diploma in Computer Science
1991–1994	Tallinna Technical Gymnasium, secondary education
1983–1991	Tallinna 55th Secondary School, primary education

Employment

2001–	Cybernetica AS, researcher
1999–2001	Cybernetica AS, software engineer
2005–2007	Estonian Information Technology College, lecturer
1996–1999	Lexbahn Software Corp., lead programmer
1995–1999	AUMA EXPO AS, engineer

ELULOOKIRJELDUS

Isikuandmed

Nimi	Margus Freudenthal
Sünniaeg ja -koht	31. juuli 1976
Kodakondsus	eesti
Keelteoskus	eesti, inglise, vene, soome
Kontaktandmed	+372 50 79 273 margus@cyber.ee

Haridustee

2008–	Tartu Ülikool, informaatika doktorant
1999–2001	Tallinna Tehnikaülikool, MSc informaatikas
1994–1999	Tallinna Tehnikaülikool, arvuti-ja süsteemitehnika diplomiõpe
1991–1994	Tallinna Tehnikagümnaasium, keskharidus
1983–1991	Tallinna 55. Keskkool, põhiharidus

Teenistuskäik

2001–	Cybernetica AS, teadur
1999–2001	Cybernetica AS, tarkvarainsener
2005–2007	Eesti Infotehnoloogia Kolledž, lektor
1996–1999	Lexbahn Software Corp., peaprogrammeerija
1995–1999	AUMA EXPO AS, insener

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

1. **Mati Heinloo.** The design of nonhomogeneous spherical vessels, cylindrical tubes and circular discs. Tartu, 1991, 23 p.

2. **Boris Komrakov.** Primitive actions and the Sophus Lie problem. Tartu, 1991, 14 p.

3. **Jaak Heinloo.** Phenomenological (continuum) theory of turbulence. Tartu, 1992, 47 p.

4. **Ants Tauts.** Infinite formulae in intuitionistic logic of higher order. Tartu, 1992, 15 p.

5. **Tarmo Soomere.** Kinetic theory of Rossby waves. Tartu, 1992, 32 p.

6. **Jüri Majak.** Optimization of plastic axisymmetric plates and shells in the case of Von Mises yield condition. Tartu, 1992, 32 p.

7. **Ants Aasma.** Matrix transformations of summability and absolute summability fields of matrix methods. Tartu, 1993, 32 p.

8. **Helle Hein.** Optimization of plastic axisymmetric plates and shells with piece-wise constant thickness. Tartu, 1993, 28 p.

9. **Toomas Kiho.** Study of optimality of iterated Lavrentiev method and its generalizations. Tartu, 1994, 23 p.

10. **Arne Kokk.** Joint spectral theory and extension of non-trivial multiplicative linear functionals. Tartu, 1995, 165 p.

11. **Toomas Lepikult.** Automated calculation of dynamically loaded rigid plastic structures. Tartu, 1995, 93 p, (in Russian).

12. **Sander Hannus.** Parametrical optimization of the plastic cylindrical shells by taking into account geometrical and physical nonlinearities. Tartu, 1995,74 p, (in Russian).

13. **Sergei Tupailo.** Hilbert's epsilon-symbol in predicative subsystems of analysis. Tartu, 1996, 134 p.

14. **Enno Saks.** Analysis and optimization of elastic-plastic shafts in torsion. Tartu, 1996, 96 p.

15. **Valdis Laan.** Pullbacks and flatness properties of acts. Tartu, 1999, 90 p.

16. **Märt Põldvere.** Subspaces of Banach spaces having Phelps' uniqueness property. Tartu, 1999, 74 p.

17. **Jelena Ausekle.** Compactness of operators in Lorentz and Orlicz sequence spaces. Tartu, 1999, 72 p.

18. **Krista Fischer.** Structural mean models for analyzing the effect of compliance in clinical trials. Tartu, 1999, 124 p.

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.

20. **Jüri Lember.** Consistency of empirical k-centres. Tartu, 1999, 148 p.

21. **Ella Puman.** Optimization of plastic conical shells. Tartu, 2000, 102 p.

22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.

23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.

24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.

25. **Maria Zeltser.** Investigation of double sequence spaces by soft and hard analitical methods. Tartu, 2001, 154 p.

26. **Ernst Tungel.** Optimization of plastic spherical shells. Tartu, 2001, 90 p.

27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 p.

28. **Rainis Haller.** M(r,s)-inequalities. Tartu, 2002, 78 p.

29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.

30. **Eno Tõnisson.** Solving of expession manipulation exercises in computer algebra systems. Tartu, 2002, 92 p.

31. **Mart Abel.** Structure of Gelfand-Mazur algebras. Tartu, 2003. 94 p.

32. **Vladimir Kuchmei.** Affine completeness of some ockham algebras. Tartu, 2003. 100 p.

33. **Olga Dunajeva.** Asymptotic matrix methods in statistical inference problems. Tartu 2003. 78 p.

34. **Mare Tarang.** Stability of the spline collocation method for volterra integro-differential equations. Tartu 2004. 90 p.

35. **Tatjana Nahtman.** Permutation invariance and reparameterizations in linear models. Tartu 2004. 91 p.

36. **Märt Möls.** Linear mixed models with equivalent predictors. Tartu 2004. 70 p.

37. **Kristiina Hakk.** Approximation methods for weakly singular integral equations with discontinuous coefficients. Tartu 2004, 137 p.

38. **Meelis Käärik.** Fitting sets to probability distributions. Tartu 2005, 90 p.

39. **Inga Parts.** Piecewise polynomial collocation methods for solving weakly singular integro-differential equations. Tartu 2005, 140 p.

40. **Natalia Saealle.** Convergence and summability with speed of functional series. Tartu 2005, 91 p.

41. **Tanel Kaart.** The reliability of linear mixed models in genetic studies. Tartu 2006, 124 p.

42. **Kadre Torn.** Shear and bending response of inelastic structures to dynamic load. Tartu 2006, 142 p.

43. **Kristel Mikkor.** Uniform factorisation for compact subsets of Banach spaces of operators. Tartu 2006, 72 p.

44. **Darja Saveljeva.** Quadratic and cubic spline collocation for Volterra integral equations. Tartu 2006, 117 p.

45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.

46. **Annely Mürk.** Optimization of inelastic plates with cracks. Tartu 2006. 137 p.

47. **Annemai Raidjõe.** Sequence spaces defined by modulus functions and superposition operators. Tartu 2006, 97 p.

48. **Olga Panova.** Real Gelfand-Mazur algebras. Tartu 2006, 82 p.

49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.

50. **Margus Pihlak.** Approximation of multivariate distribution functions. Tartu 2007, 82 p.

51. **Ene Käärik.** Handling dropouts in repeated measurements using copulas. Tartu 2007, 99 p.

52. **Artur Sepp.** Affine models in mathematical finance: an analytical approach. Tartu 2007, 147 p.

53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007,

54. **Kaja Sõstra.** Restriction estimator for domains. Tartu 2007, 104 p.

55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.

56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.

57. **Evely Leetma.** Solution of smoothing problems with obstacles. Tartu 2009, 81 p.

58. **Ants Kaasik.** Estimating ruin probabilities in the Cramér-Lundberg model with heavy-tailed claims. Tartu 2009, 139 p.

59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.

60. **Indrek Zolk.** The commuting bounded approximation property of Banach spaces. Tartu 2010, 107 p.

61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.

62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.

63. **Marek Kolk.** Piecewise Polynomial Collocation for Volterra Integral Equations with Singularities. Tartu 2010, 134 p.

64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.

65. **Larissa Roots.** Free vibrations of stepped cylindrical shells containing cracks. Tartu 2010, 94 p.

66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.

67. **Margus Niitsoo.** Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.

68. **Olga Liivapuu.** Graded q-differential algebras and algebraic models in noncommutative geometry. Tartu 2011, 112 p.

69. **Aleksei Lissitsin.** Convex approximation properties of Banach spaces. Tartu 2011, 107 p.

70. **Lauri Tart.** Morita equivalence of partially ordered semigroups. Tartu 2011, 101 p.

71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.

72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.

73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.

75. **Nadežda Bazunova.** Differential calculus d3 = 0 on binary and ternary associative algebras. Tartu 2011, 99 p.

76. **Natalja Lepik.** Estimation of domains under restrictions built upon generalized regression and synthetic estimators. Tartu 2011, 133 p.

77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.

78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.

79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.

80. **Marje Johanson.** M(r, s)-ideals of compact operators. Tartu 2012, 103 p.

81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.

82. **Vitali Retšnoi.** Vector fields and Lie group representations. Tartu 2012, 108 p.

83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.

84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.

85. **Erge Ideon.** Rational spline collocation for boundary value problems. Tartu, 2013, 111 p.

86. **Esta Kägo.** Natural Vibrations of elastic stepped plates with cracks. Tartu, 2013, 114 p.