Methods and Techniques for the Design and Implementation of Domain-Specific Languages

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op woensdag 11 januari 2012 om 12:30 uur door

Zef HEMEL

doctorandus informatica
geboren te Groningen

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. A. van Deursen

Copromotor: Dr. E. Visser

The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was financially supported by the Netherlands Organisation for Scientific Research (NWO)/Jacquard project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

Printed and bound in The Netherlands by CPI Wöhrmann Print Service.

ISBN 978-90-8570-794-3

Preface

I vividly remember my application interview for this Ph.D. position, early 2007. A time when buses still stopped almost in front of "the tallest building on campus" in Delft where the Software Engineering Research Group is situated. My Ph.D. supervisor to be, Eelco Visser, decided to try out a new style of interviewing he had picked up during his postdoc in Portland, Oregon. This style entailed about four hours of interviews with him, Arie and other Ph.D. students in the group. The talks with Eelco were toughest, he asked me challenging questions, including "so, are you any good?" Arie spent his interviewing time laying out his vision for the group and its history. One Ph.D. student I talked to, Ali Mesbah, was happy that there was somebody applying with a web background; another, Martin Bravenboer, said I would really enjoy working with Eelco Visser. After hours of interviews they offered me the job with only one concern: "will you be able to focus on a single topic for four years?"

Well, dear reader, you have the answer in your hands. It takes you on a journey of four years of research, Zef-style. I worked on topics ranging from workflow, intermediate languages, compiler implementation techniques to design concerns related to languages for web applications and mobile applications. Yet, the questions I asked myself over the past four years have remained the same: how do we make software development more productive and more fun? How do we stop spending our programming time spelling out the nitty gritty detail that we don't care about? Domain-specific languages are one way to accomplish that, and this little book contains my scribblings related to both designing and implementing such languages.

Even though the work is done, at least for me, I look forward to collaborating with the people in Delft in the future. I'm happy to work at Cloud9 IDE, Inc. where I will apply much of what I have learned the past four years and I look forward to do so in collaboration with the people in Delft. Industry has a lot to learn from research — but research also has a lot to learn from industry.

ACKNOWLEDGEMENTS

As support during this process I would like to thank a couple of people. First and foremost, my supervisor and co-promotor Eelco Visser. He provided guidance and taught me everything I needed to know about how to conduct research and has been an inspiration throughout the process. Second, I would like to thank my promotor, Arie van Deursen, for his help, feedback and guidance along the way — whenever I felt insecure, he was reassuring. I would also like to thank my reading committee for their valuable feedback and kind words: William Cook, Rinus Plasmeijer, Mark van den Brand, Catholijn Jonker and Geert-Jan Houben.

Then, I would like to thank all my other colleagues at SERG, and especially our coffee/tea/lunch group: Danny Groenewegen, Sander Vermolen, Sander van der Burg, Maartje de Jonge, Rob Vermaas, Eelco Dolstra and Lennart Kats (my bro', who people always confused me with, for some unknown reason). I would also like to thank Esther van Seters, who helped me with the administrative side of finishing up this Ph.D.

I would like to thank Martin Bravenboer and Eelco Dolstra for the source files that formed the basis for this thesis' lay-out, and Alberto González Sánchez for the inspiration for its cover. The photograph used for the cover was taken by Trey Ratcliff and is entitled "Morning. Coffee. Yellowstone. Fog."[1] The interpretation of how it relates to the thesis is left as an exercise to the reader.

I would like to thank my family, for their support throughout the years — my parents in particular. And last, but surely not least, I would like to thank my wife Justyna — without her I would likely not have done this Ph.D., she supported me always.

Zef Hemel
December 8, 2011
Poznań, Poland

[1]http://www.flickr.com/photos/stuckincustoms/4885953697/

Contents

1 Introduction

The promise of *model-driven engineering* is to reduce the development and maintenance effort of software by developing at a higher-level of abstraction through the use of *domain-specific languages* (DSLs). Domain-specific languages, as opposed to *general-purpose languages*, are software languages that focus on a specific problem domain, e.g. insurance, database querying, grammars or workflow.

The research in this thesis is conducted as part of the *MoDSE* (Model-Driven Software Evolution) project. The goal of the MoDSE project is to develop a systematic approach to model-driven software development using domain-specific languages. This approach includes methods, techniques, and underlying tool support. The group in which the research is conducted (the Software Engineering Research Group at Delft University of Technology) is building and evolving tools to simplify the development of domain-specific languages, including SDF [Heering et al., 1989] and SGLR [Visser, 1997a] for parsing, Stratego/XT [Visser, 2004, Bravenboer et al., 2008] for program transformation and Spoofax [Kats and Visser, 2010a] for building IDE (Integrated Development Environment) plug-ins for the developed languages.

The goal of the research is to explore the DSL design space and to develop techniques to simplify the implementation of DSLs. The research is conducted through case studies in DSL design, using tools developed as part of the MoDSE project.

1.1 DOMAIN-SPECIFIC LANGUAGES

Domain-specific languages have been in use for decades. In Unix there is a long tradition of such "little languages" [Bentley, 1986], including `sh` and `bash` (for shell scripting), `lex` and `yacc` (for lexical analysis and parsing), and `make` (for automated software builds). Other examples of commonly used DSLs in software engineering include XML to express structured data, XSLT to transform XML documents, SQL (Structured Query Language) to query relational databases, HTML to mark-up web documents, and CSS to style HTML documents.

Advantages and properties of domain-specific languages have been studied in literature [Mernik et al., 2005, van Deursen et al., 2000a, van Deursen and Klint, 1998, Spinellis and Guruprasad, 1997, Spinellis, 2001] and offer many opportunities:

- *Concise, domain-specific notations.* A DSL offers natural notations to more succinctly express the programmer's intention.

- *Analysis and verification.* The limited scope of a DSL enables a domain-specific verification tool to analyze a DSL program and report domain-specific warnings and error messages.
- *Abstraction.* A DSL can abstract from boilerplate code (low-level, repetitive code code, e.g. the initialization sequence of a library) that has to be written by hand otherwise.
- *Platform independence.* DSLs can abstract from a particular platform or implementation. Therefore, multiple implementations of a DSL can exist, e.g. an interpreter, a compiler that compiles the DSL to Java and a compiler that compiles to C.
- *Domain-specific tooling support.* The limited application domain and higher level of abstraction of a DSL enable better program understanding. Consequently, there is an opportunity to offer better tooling as part of an Integrated Development Environment (IDE).

In the current state of practice, especially in software development for the web, *mixing* of general-purpose languages (such as C, Java, Python and Ruby) and domain-specific languages is ubiquitous. For instance:

1. A Java program (GPL) that has a SQL (DSL) query embedded (in a string) to retrieve data from a database;
2. A shell script (DSL) that invokes a number of GPL programs in sequence; and
3. A HTML document (DSL) that embeds JavaScript (GPL), which in turn manipulates the web page (the HTML Document Object Model) when the user clicks a button.

The examples of DSLs mentioned thus far are examples of *external DSLs*: languages that define their own custom syntax and semantics. Many modern software frameworks use *internal DSLs*, a style of API design that uses techniques such as fluent interfaces [Fowler, 2005] and meta-programming to give the GPL the look and feel of a domain-specific language. For instance, the EasyMock[1] Java library uses fluent interfaces to implement an internal Java DSL that looks like an English sentence:

```
EasyMock.expect(mockCollection.remove(null))
        .andThrow(new NullPointerException())
        .atLeastOnce();
```

Other well-known examples of internal DSLs include Ruby on Rails[2] for building web applications, Rake[3] for building Ruby projects and JQuery[4] for

[1]http://easymock.org
[2]http://www.rubyonrails.org
[3]http://rake.rubyforge.org
[4]http://www.jquery.com

manipulating the HTML Document Object Model. Languages such as Haskell and the various Lisp dialects rely heavily on internal DSLs as well.

Internal DSLs are cheap to develop, because they do not require any knowledge of parsing and compiler construction. However, their syntax and semantics are limited by their host language. In addition, the amount of checking that is performed on their programs at compile time is limited by the power of the type system of the host language, and its error messages are typically generic and difficult to interpret. Therefore, in this thesis, we focus primarily on external DSLs.

1.2 DSLS AND THE WEB

Over the past years, many types of applications have moved to the web: encyclopedias, e-mail, word processing, book keeping — they all have web-based versions today. The main advantage of developing for the web, compared to building desktop applications, is deployment. A web application can be accessed from anywhere in the world, from any platform, be it Windows on a desktop, a Macintosh computer or a mobile device. There is no need to install any software beyond a web browser, and new versions of software can be deployed once to a web server and are immediately accessible for all users.

There is an abundance of libraries and frameworks available enabling developers to rapidly develop web applications. Most modern web web frameworks have integrated many DSLs, both internal and external. For instance, consider the Java-based Seam framework[5]. Applications developed using the Seam framework use Java for application logic, the JavaServer Faces DSL (external DSL – an extension of HTML) to construct user interfaces, CSS for styling the user interface (external DSL), Hibernate annotations on Java classes to define the application's data model (internal DSL), a policy language for defining access control rules (external DSL) and various XML-based configuration files (external DSLs).

While the Seam framework simplifies web development using Java, it introduces some new problems:

- *Code encoding protocols.* Repetitive code that the used language cannot abstract over, e.g. code required to setup application components (encoded using different languages) to work together.

- *Verification.* The composition of application components written using different languages only happens at run-time. No reliable tools are available that verify the composed program, as e.g. a Java compiler does for pure-Java programs. As a result, long compile, deploy and test cycles are required to detect errors.

[5]`http://seamframework.org`

1.3 WEBDSL

Due to the aforementioned problems, the web is an interesting domain to invent a new domain-specific language for. Therefore, Visser [2008] developed a prototype of *WebDSL*, an external domain-specific language to rapidly develop data-driven web applications. Rather than a collection of loosely coupled DSLs, which are difficult to verify statically, WebDSL is a single *syntactically integrated* language. While syntactically integrated, WebDSL still supports *separation of concerns* through a number of domain-specific sub-languages for defining the various aspects of a web application, including user interfaces, data models (to be persisted to a database) and actions to define application logic.

WebDSL is a first case study in the design and implementation of domain-specific languages as part of the MoDSE project. The syntax of WebDSL is defined using SDF (Syntax Definition Formalism) [Heering et al., 1989] and the compiler is implemented using Stratego/XT [Visser, 2004].

As discussed, DSLs are typically only used to develop *aspects* of an application, for instance to query a database (SQL queries), to glue parts of an application together (shell scripts), or to generate scaffolding code that needs to be extended to build a complete application (Mod4J [Lussenburg et al., 2010]). In contrast, WebDSL is designed to construct *complete* applications. The WebDSL compiler translates programs into ready-to-deploy web applications and thus performs 100% code generation, i.e. the generated application need not (and should not) be modified after generation.

WebDSL significantly reduces the amount of code required to build data-driven web applications, by introducing more concise notations and automatically generating required boilerplate code. Developer productivity is further improved by the static verification of WebDSL applications, which detects common inconsistencies in web applications, including non-existing data model properties referenced in the user interface and internal links to pages that do not exist.

1.4 PROBLEM STATEMENT

The original WebDSL as described by Visser [2008], was a proof of concept for a systematic approach to the design and implementation of DSLs. The prototype implementation of WebDSL sparked a range of new research challenges and opportunities:

- *Application verification.* WebDSL had a type checker that could verify basic properties of its web applications, but many errors remained undetected by the checker — there were many opportunities to improve *verification* of WebDSL applications and to support it in a scalable and extensible manner in the compiler.

- *Coverage.* While WebDSL could be used to build basic web applications, its *coverage* was limited, a large class of applications could not yet be built.

- *Abstraction.* Many aspects of a web application still required too much boilerplate code to be written by hand — there were opportunities to develop better *abstractions*.

- *Code generation.* The code generation component of the WebDSL compiler was a large, monolithic piece of code that was difficult to maintain and extend — there was an opportunity to refactor the compiler and make it more extensible and to better separate compiler concerns.

- *Portability.* The WebDSL compiler generated code using the JBoss Seam framework. The performance of generated applications was poor. Therefore, the possibility to *port* the compiler to run on a different platform, possibly multiple, needed to be investigated.

Clearly, the WebDSL project offered a fertile playground to explore both *design* and *implementation* aspects of syntactically integrated DSLs. In addition, the design approach taken to the web domain in WebDSL could lead to a series of similarly designed languages for different domains. Thus, we formulate the core research question of this thesis as follows:

CORE RESEARCH QUESTION

> *How to design and implement statically verifiable and syntactically integrated domain-specific languages?*

The specific research challenges mentioned are addressed in the following sections.

1.5 VERIFICATION

The original WebDSL paper [Visser, 2008] describes the problem that faults in JBoss Seam applications only manifest themselves at run time. However, what is unclear is whether the problem of late failure is a problem of the Seam framework specifically, or if it is a more widespread problem.

For frameworks that rely on dynamically typed languages, such as Ruby on Rails (based on Ruby) and Django (based on Python) the lack of static verification is to be expected. These languages do not have compilers or type checkers, so any error can only be detected through repeated systematic testing. However, one may expect that web frameworks based on *statically typed* languages, such as Lift (based on Scala) and the Seam framework (based on Java) would benefit from static verification by the Scala and Java compilers. However, at least in the case of the Seam framework, this is only partly true.

The use of frameworks involves adhering to numerous additional rules, conventions and protocols that are specific to the framework's domain — properties not verifiable by the Java compiler.

In addition, Seam web applications are not built using a single language; numerous DSLs are used to define components of the application. While the components interact frequently, these interactions are not statically verified. Therefore, in practice, inconsistencies manifest at run-time as software failures. To investigate if the problems exposed by the Seam framework are more widespead, to get a better understanding of the origin of these issues and to come up with a solution, we formulate the following research question:

RESEARCH QUESTION 1

> *Is the lack of static verification a common problem in today's web frameworks? If so, how can that problem be remedied?*

To answer this question, Chapter 2 surveys a number of state-of-practice web frameworks (Ruby on Rails, Lift and Seam) by introducing errors and observing how and when these errors manifest as faults when the application is run. Subsequently, we discuss an approach to language design that supports the static verification of entire applications and demonstrate how this approach is implemented in WebDSL using the Stratego transformation language.

1.6 COVERAGE AND ABSTRACTION

A risk of domain-specific languages is that their constructs are *too* high-level, preventing good *coverage* of the domain. While such DSLs typically support very concise solutions to a set of problems, they are rendered useless as soon as variations of the problem have to be solved that the language does not support.

A manifestation of this problem can be found in the context of *workflow* description languages. Workflow is concerned with the coordination of activities performed by participants involving artifacts. For instance, a user registration workflow involves filling in a registration form, e-mail verification and administrator approval; the process of academic paper reviewing involves bidding, review submission, discussion and notification of authors. There are many DSLs that support high-level definition of workflows, including UML activity diagrams [Dumas and ter Hofstede, 2001], BPEL [Curbera et al., 2003], and YAWL [van der Aalst and ter Hofstede, 2005]. While workflow languages raise the level of abstraction from manually encoding to declarative descriptions of workflows, they have no mechanism to deal with *coverage* issues, i.e. what if a workflow follows a pattern that is not supported by the language?

This raises the issue of maintaining *coverage* while introducing *abstractions* in a DSL:

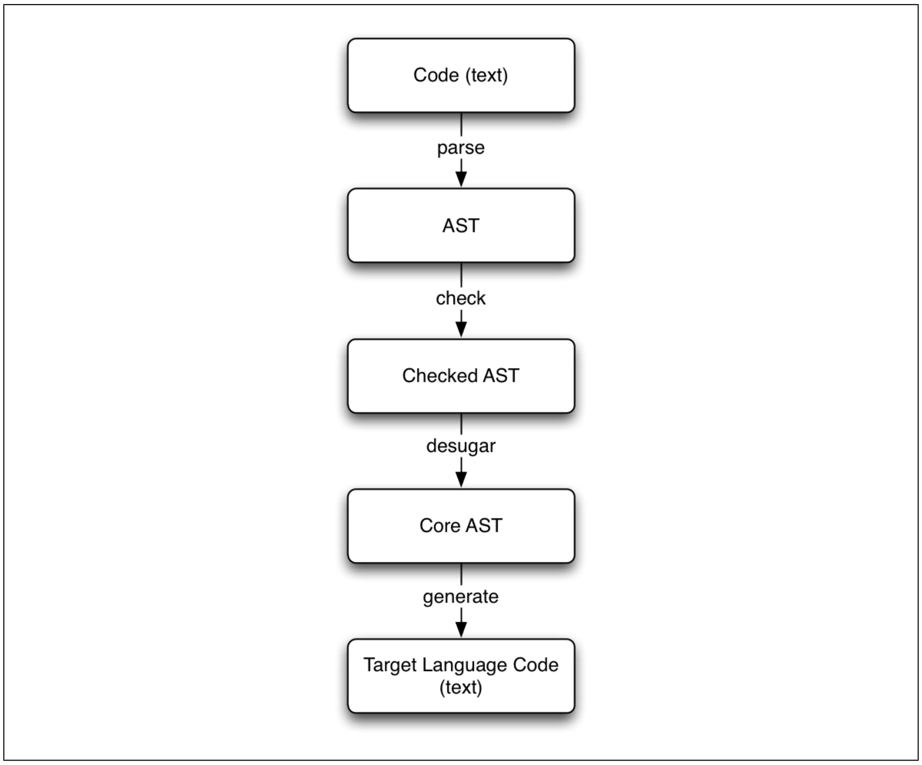


Figure 1.1 Compiler transformation steps

RESEARCH QUESTION 2

> *How can the level of abstraction in a DSL be raised without reducing its coverage?*

Chapter 3 describes *WebWorkFlow*, an extension of WebDSL that adds concise workflow constructs. To prevent reduced coverage of the resulting language, workflows can be expressed at three levels of abstraction, enabling developers to escape to a lower level for more flexibility.

1.7 COMPILER MODULARITY AND SEPARATION OF CONCERNS

A DSL compiler typically compiles a program in roughly four steps (Figure 1.1):

1. During the *parsing* step, the textual representation of the program is turned into a model of the program referred to as an *abstract syntax tree* (AST), a tree-shaped data structure that forms a structured, abstract representation of the program that can be easily analyzed.

2. The *checking* phase analyzes the resulting AST for consistency and reports errors and warnings as needed.

3. *Desugaring transformations* are transformation that simplify the checked AST. These transformations can be simple normalizations as well as more complex transformations.

4. *Code generation* translates the core language AST that is produced by the previous steps to the language of the target platform, typically a general-purpose language e.g. Java or C.

Over the years, many students contributed to the WebDSL compiler and grew it rapidly to over 30,000 lines of code. The code base has become increasingly difficult to extend.

One of the code smells (indicators that code is not in a good state) in the compiler are so-called "God rules". Analogous to "God classes" [Deligiannis et al., 2004] in object-oriented programming, "God rules" are large transformation rules in the compiler that dispatch numerous sub-rules that generate parts of a single, monolithic artifact. As a result, separate aspects of the language that are responsible to generate part of a single artifact have to be invoked from a God rule — a typical case of mixing concerns within the compiler.

WebDSL internally annotates the AST with type information that is used by transformation rules. However, as soon as the AST is transformed, those type annotations are often invalid or disappear. Therefore, type analysis has to be performed after every transformation iteration. As a result, the compiler uses an increasingly long pipeline of stages that first perform a transformation, followed by a reanalysis of the entire AST. Not only is this inefficient, it also requires the extension of the transformation pipeline every time a new transformation is added to the compiler.

The described problems raise the following research question:

RESEARCH QUESTION 3

How to maintain separation of concerns and efficiently combine analysis with transformation in a DSL compiler?

Chapter 4 introduces the *code generation by model transformation* approach to building DSL code generators. The approach is based on the idea of generating code as a *model* (AST) rather than plain text, supporting further transformation of code before being written to files. The chapter describes how transformation of generated code can be used to improve separation of concerns in the compiler. Part of the approach is a novel method combining type analysis with transformations. The implementation of the code generation by model transformation approach in the WebDSL compiler is described.

1.8 RETARGETABILITY

The initial prototype of WebDSL generated code that uses the JBoss Seam framework. The resulting applications had to be deployed on a Java Application Server such as JBoss[6]. The time between invoking the WebDSL compiler and the application being available to test was usually multiple minutes. In addition, once the application ran, it ran slowly. This was partly due to the heavy-weight framework used and partly due to all the extra code that needed to be generated to let the framework behave according to WebDSL semantics [Groenewegen et al., 2008].

Therefore, we decided to replace the JBoss Seam back-end with two lighter-weight back-ends: one targeting Java Servlets and one targeting Python running on Google AppEngine[7]. Both back-ends generate low-level code, rather than relying on heavy-weight frameworks. In addition to resulting in better performance, the new back-ends test whether WebDSL is a *platform-independent language*, i.e. if it has an implicit dependency on the underlying platform. Generally speaking, a DSL should abstract from the underlying platform and thus be *portable*.

The two new back-ends enable users to develop WebDSL applications and deploy on either a Java infrastructure or Google's AppEngine infrastructure. However, the two back-ends slow down the development of WebDSL. Every language feature that requires changes in the back-end takes twice as much work, because the change has to be implemented twice. In practice this does not always happen and as a result, over time, the back-ends slowly diverge in their feature sets.

While being developed for two quite different platforms, the two back-ends are very similar in structure. Since neither of the back-ends rely heavily on frameworks (besides object-relational mapper libraries), and essentially generate `print` statements that produce HTML code, the differences in generated code are largely cosmetic.

To remedy similar issues, traditional compilers rely on *intermediate languages* to simplify retargeting compilers to multiple machine architectures [Steel, 1961, Richards, 1971, Peyton Jones et al., 1999, Benitez and Davidson, 1988, George, 1997]. Instead of directly emitting machine instructions for each targeted machine arcitecture, the compiler emits intermediate language instructions, which in turn are mapped to specific instructions for each machine architecture. However, these intermediate languages are not easily reused in DSL compilers. DSL compilers generate code at the level of *software platforms* rather than machine architectures. A software platform consists of one or more programming languages with a set of libraries and frameworks, deployable on one or more operating systems. Targeting software platforms enables compiler builders to implement DSLs more quickly than having to emit low-level machine instructions. This raises the following question:

[6]`http://jboss.com`

[7]`http://appengine.google.com`

RESEARCH QUESTION 4

> *How can the implementation and maintenance effort of supporting multiple software platforms with DSLs be reduced?*

Chapter 5 introduces PIL, a Platform Independent Language to be used as a high-level intermediate language for DSL compilers to cheaply support multiple software platforms. Instead of generating target platform code directly, it emits PIL code that is subsequently compiled to code for multiple platforms by the PIL compiler. Consequently, only a single compiler back-end needs to be built and maintained.

1.9 AN INTEGRATED DSL FOR THE MOBILE WEB

In order to test the ideas and techniques developed in Chapter 2–5 we perform a second case study by developing a language for a new domain. The domain of *mobile web applications* is an obvious choice because it exposes many of the same problems as the web in general.

The market for mobile touch devices such as the iPhone, iPad, Android and BlackBerry devices is rapidly growing. At the time of writing, over a million of these devices are sold every day[8]. These devices come with fast and modern web browsers that enable its users to access the web from anywhere.

However, there are important restrictions to take into account when developing web applications for mobile devices. Mobile devices are used in different contexts and have different features and constraints than the personal computers that are typically employed to access the web. For instance, on mobile devices, Internet access is not always available, reliable or fast, screen estate is limited, expected user interaction patterns are different, such as touch controls and gestures such as tapping, swiping and pinching and applications are expected to respond to changes in context, e.g. device orientation and changes in location.

Due to the restricted screen size and limited forms of input (finger touches), mobile applications use different user interface styles than regular web applications. For instance, the navigation through mobile user interfaces is typically hierarchical: At the top level are tabs, the roots of the hierarchy. Each tab typically has a list of items the user can choose from. When the user picks one, the current screen slides to the left, the new screen slides in from the right. The user can then drill down further, or go back, effectively moving up and down a tree structure. The mobile web space is still young and rapidly changing. There is a lot of innovation in user interface and interaction patterns ongoing requiring mobile toolkits and frameworks to quickly adapt.

WebDSL is designed with a particular implementation in mind — stateless server-side web applications — imposing restrictions on the language. For instance, WebDSL applications are stateless and HTTP request-based. Therefore, changes in data are only persisted and become visible after a request to

[8] `http://mashable.com/2011/05/19/smartphone-sales-q1-2011-gartner/`

the server has taken place. The expectation for mobile applications is different. Mobile applications typically do not have "Save" buttons or form submits, instead data is persisted continuously. Nevertheless, mobile web development exposes many similar problems as regular web development including the need to combine multiple loosely-coupled languages in a single application (for instance HTML, CSS, SQL and JavaScript) and the boilerplate code that needs to be written (such as cache manifest files). This raises the following question:

RESEARCH QUESTION 5

> *How can the language design and implementation techniques developed for WebDSL be applied to the mobile web domain?*

Chapter 6 describes mobl, a new language to rapidly develop mobile web applications. Mobl implements a number of previously discussed techniques, including design for static verification (Chapter 2), syntactic language extensions (Chapter 3) and certain code generation by model transformation techniques (Chapter 4).

1.10 RESEARCH METHOD

The MoDSE project aims to develop a systematic approach to model-driven software development using DSLs by developing methods, techniques, and tools that support DSL development. In order to evaluate the developed methods, techniques and tools we apply them in practice. The development of WebDSL and mobl contribute both toward the development of a systematic approach to DSL design as well as evaluating the tools developed as part of MoDSE (including SGLR, Stratego and Spoofax) in a real-world setting.

Developing languages beyond the level of prototypes enables them to be used in practice, resulting in valuable feedback that supports our research. Beside using them ourselves, we attempt to develop communities of users around our DSLs that test the language and provide feedback to steer further development.

In addition, we post preliminary research results on-line to obtain input and share our findings with non-academics that would otherwise not read our papers. For instance, the survey of the state of practice for Chapter 2 had been posted online as a series of blog posts[9] and generated a fair amount of attention and input for the version prepared for publication.

1.11 ORIGIN OF CHAPTERS

The core chapters (Chapter 2–6) in this dissertation are slight adaptations of peer-reviewed papers at programming language and software engineering conferences and journals. Since these papers were published independently,

[9] `http://zef.me/2308/when-rails-fails`

they can also be read independent of each other. Since all papers have their own, individual contributions, there is some redundancy in the background material, motivation, and examples. In addition, most chapters end with a postscript section presenting our updated view on the chapter since publication, as well as putting it in a chronological perspective.

- Chapter 2 is an updated version of the Journal of Symbolic Computation 2011 paper *Static Consistency Checking of Web Applications with WebDSL*. [Hemel et al., 2011]
- Chapter 3 is an updated version of the MODELS 2008 paper *WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications*. [Hemel et al., 2008b]
- Chapter 4 is an updated version of the Software and System Modeling journal paper *Code Generation by Model Transformation. A Case Study in Transformation Modularity*. [Hemel et al., 2010]

  A previous version of this paper was presented at ICMT 2008 [Hemel et al., 2008a].
- Chapter 5 is an updated version of the SLE 2009 paper *PIL: A Platform Independent Language for Retargetable DSLs*. [Hemel and Visser, 2009]
- Chapter 6 is an updated version of the OOPSLA 2011 paper *Declaratively Programming the Mobile Web with Mobl*. [Hemel and Visser, 2011]

Papers contributed to during this research, but not directly included in this thesis are:

- The DSM 2008 paper *When Frameworks Let You Down: Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages*. [Groenewegen et al., 2008]
- The IEEE Software 2010 paper *Separation of Concerns and Linguistic Integration in WebDSL*. [Groenewegen et al., 2010]

Static Consistency Checking of Web Applications with WebDSL

2

ABSTRACT

Modern web application development frameworks provide web application developers with high-level abstractions to improve their productivity. However, their support for *static verification of applications* is limited. Inconsistencies in an application are often not detected statically, but appear as errors at run-time. The reports about these errors are often obscure and hard to trace back to the source of the inconsistency. A major part of this inadequate consistency checking can be traced back to the lack of linguistic integration of these frameworks. Parts of an application are defined with separate domain-specific languages, which are not checked for consistency with the rest of the application. Examples include regular expressions, query languages and XML-based languages for definition of user interfaces. We give an overview and analysis of typical problems arising in development with frameworks for web application development, with Ruby on Rails, Lift and Seam as representatives.

To remedy these problems, in this chapter, we argue that domain-specific languages should be designed from the ground up with static verification and cross-aspect consistency checking in mind, providing linguistic integration of domain-specific sub-languages. We show how this approach is applied in the design of WebDSL, a domain-specific language for web applications, by examining how its compiler detects inconsistencies not caught by web frameworks, providing accurate and clear error messages. Furthermore, we show how this consistency analysis can be expressed with a declarative rule-based approach using the Stratego transformation language.

2.1 INTRODUCTION

Web applications are complex software systems that combine many technical concerns, such as database querying, input handling, user interface design, and navigation. Web application frameworks are often used to simplify web development and improve web developer productivity. A web framework consists of a set of APIs built on a general-purpose programming language. Popular web frameworks include JBoss Seam, Lift, Ruby on Rails, and Django. These frameworks enable abstraction over many low-level details of normal web application development, avoiding handwritten boilerplate code, thus increasing developer productivity.

While web frameworks improve the clarity of the application and expressivity of developers that use it, applications containing inconsistencies (faults) often fail late, i.e. at run time or deployment time instead of at compile time. Even inconsistencies in applications written using a framework based on a statically typed language such as Java or Scala are often only revealed at deployment time or at run time. The errors produced when the application fails are often difficult to trace back to their origin and error messages are typically not domain-specific, exposing framework implementation details.

2.1.1 *Causes of Late Failure*

Web frameworks use a combination of high-level APIs, meta-programming techniques, and domain-specific languages to achieve higher developer expressivity. Meta-programming techniques used range from reflection in Scala and Java-based frameworks to extension and adaptation of classes and objects at runtime in frameworks based on dynamically typed languages such as Ruby and Python. Domain-specific languages (DSLs) are used for user interface construction (ASP.NET, JSF), access control policies (rule files), pattern matching (regular expression) and database queries (SQL, HQL).

Domain-specific languages, as used by web frameworks, are not *linguistically integrated* with the rest of the framework. Therefore, in practice, very few consistency checks are performed on *connections* between the application aspects defined in different domain-specific languages, resulting in late failure. Web frameworks based on statically typed general purpose languages can report a limited class of application inconsistencies at compile-time. Modern frameworks, such as JBoss Seam and Scala Lift, cannot identify all inconsistencies during compilation, because the static checks they provide are limited to the type checker of their host language (Java and Scala respectively). Other errors, often inconsistencies between application components defined in separate DSLs, are only reported at deployment time or at run time, resulting in the same issues web frameworks based on dynamically typed languages have.

Frameworks based on dynamically typed languages, such as Ruby on Rails and Python's Django only provide *runtime* consistency checks. Typically, consistency in these frameworks is not explicitly checked, but rather manifests itself when the faulty code is executed. Consequently, errors are not always easily traced back to the source of the problem, and the messages are often unclear and confusing, relating to the framework implementation and not the actual web application. Many errors – not all – include a stack trace directing the developer to the point in the source code (either the framework's code or the developer's) where the failure occurred. Reported error messages often expose underlying implementation details. For instance, when routing to a non-existing controller in Ruby on Rails, an "uninitialized constant" error is reported that refers to a name-mangled version of the application's controller name.

2.1.2 *Design for Consistency Checking*

One solution to late failure and bad error reporting is to build static verifiers for existing web frameworks. However, developing verifiers is very complicated because the framework was never intended to be statically verified.

In this chapter we propose a different solution: web languages should be *designed* to enable static verification of its applications for consistency. We show that linguistic integration of the languages is essential for effective checking of consistency properties that span multiple aspects of the application. Linguistic integration entails that different technical concerns, typically expressed using completely separate languages, are instead expressed using a single language integrating the syntax and semantics of multiple sub-languages as described by Visser [2007a].

We illustrate this approach with WebDSL, a web language integrating a number of sub-languages for different concerns related to the construction of web applications with a rich data model, such as a data modeling language, a user interface language, an action language, and an access control language. Based on linguistic integration, consistency properties that span multiple technical domains can still be statically checked in WebDSL. Important domain concepts, such as entities, pages and templates are first-class language elements in WebDSL ensuring that error messages for consistency violations are always expressed in a domain-specific manner, e.g. "entity not found" rather than "undefined constant".

2.1.3 *Contributions*

This chapter identifies early, accurate consistency checking of web applications as a problem. It is an important problem since it directly affects the productivity of web developers: with better, more accurate static checks, maintenance of source code can be simplified. Existing frameworks based on general-purpose programming languages provide only a limited number of consistency checks. External tools that provide additional checks are hard to construct and maintain, especially when targeting linguistically separate languages. We argue that only an integrated solution allows for an efficient implementation of static consistency checking.

The contributions of this chapter are as follows:

1. An analysis of areas where consistency checks are typically lacking within current web frameworks.
2. An analysis of the quality of failure of three state-of-the-practice web frameworks.
3. A declarative, rule-based approach to linguistic integration and consistency checking.
4. A demonstration of this approach with an implementation in the Stratego transformation language of consistency checking for a (subset of) WebDSL.

This chapter focuses on consistency checking, relating it to consistency checks in other frameworks, providing a detailed description of the different static checks performed by the language, showing novel, non-trivial ways a web application can be checked, and describing the rule-based architecture in which these checks are implemented.

We begin this chapter with a study of different classes of inconsistencies in web applications, showing how these are checked and reported in major web frameworks. In many cases, these consistency checks are lacking in accuracy and in quality of the error reports. In Section 2.3 we analyze why this is the case, looking at the implementation of the different frameworks. In Section 2.4 we explain how to address the discovered problems, and describe solutions applied in WebDSL. In Section 2.5 we demonstrate how a static checker for a subset of WebDSL can be implemented using rewrite rules in Stratego. Section 2.6 handles discussion points and describes differences with previous work.

2.2 FAILURES IN WEB APPLICATIONS

Modern web applications comprise a number of aspects, often expressed using different domain-specific languages, e.g. HTML for user interfaces and data models using annotated Java code. Our experience with mainstream web development frameworks has been that faults, especially across aspect boundaries, manifest themselves late, e.g. only when the application is run and the specific page is loaded, often resulting in developer annoyance and a decrease in productivity. Not only do failures occur late, they are often difficult to trace back to their origin and provided error messages are not domain-specific and expose implementation details of the framework.

To analyze failures in web application frameworks, we have conducted an experiment investigating the problems in fault manifestation and reporting in the current state of practice. We evaluate four aspects of mainstream web frameworks (data model, user interface, application logic and access control). Through *fault seeding* we register when and how applications built using these frameworks fail. Subsequently, the next section will examine the reasons of failure and how they can be mitigated.

2.2.1 *Web Application Aspects*

Typical modern web applications comprise multiple aspects. Application aspects include the data model, user interface and business logic. To simplify development, frameworks offer specialized languages and APIs for these aspects. For instance, user interfaces are defined using an extension of HTML, data models are defined by annotating classes with persistence annotations, and a rule language is used to declaratively specify access control rules. While the use of specialized languages and APIs enable separation of concerns, the application aspects are not completely independent. Each aspect contains links to other application aspects. These inter-aspect links are an important cause of the late detection of web application failures.

For our study we selected four common application aspects, which are listed below. This list is not meant to be exhaustive, but we believe it is a representative list of aspects that are typically covered by web application frameworks. Other application aspects have similar issues. For each application aspect we list some common internal and inter-aspect faults.

- **Data model**, web frameworks typically have APIs to define the data model of the web application in a declarative manner. The data model represents the data structures that need to be persisted. Common faults:
  - *Properties of non-existing types*, the data model defines properties of types that do not exist.
  - *Invalid inverse properties*, inverse properties refer to non-existing properties.
  - *Invalid data validation*, rules to validate the values of data model properties are invalid, e.g. the regular expression that checks the zip code format contains a syntax error.
- **User interface** is typically defined using a separate DSL, usually an extension of HTML. Common faults:
  - *Invalid page elements*, the use of tags and controls that do not exist or are used incorrectly.
  - *Invalid element nesting*, incorrectly nesting tags and controls in an invalid manner, e.g. nesting list items outside a list.
  - *Invalid references to data model*, the user interface often presents data from the data model, references to the data model, e.g. entity properties, may be incorrect.
  - *Invalid links to pages*, links to pages within the application do not exist or are linked to with wrong parameters.
  - *Invalid links to actions*, actions to be triggered, e.g. when pushing a button, do not exist or are invoked incorrectly.
- **Application logic** defines the business logic of the application. Common faults:
  - *Invalid references to data model*, properties and types that do not exist.
  - *Invalid redirect from actions*, the user is redirected to pages within the application that do not exist.
  - *Invalid data binding*, form data is bound to entities incorrectly.
- **Access control** defines who can access what parts of the application in a declarative manner. Common faults:
  - *Invalid references to data model*, access control rules link to non existing data model entities and properties

2.2.2 *Moment of Failure*

Application faults should manifest themselves as soon as possible; the sooner the developer knows, the sooner he or she can resolve the problem. Thus, the *moment of manifestation* is an important quality of fault detection in frameworks. Once a fault has manifested itself, the developer has to resolve the problem. Therefore, the *retraceability* of the problem to its source is important; the location of the fault should be clearly indicated in the code. Once the source of the problem has been pin-pointed, the reported error message should indicate what the problem is *in terms of the application domain* and should reveal as little about the underlying implementation as possible. For instance, when a link to a non-existing page within the application is found, the error should use domain terminology such as "page" and "link" rather than "constant" or "method".

Thus, we can determine the quality of fault detection in frameworks and DSLs by considering three aspects:

1. The *moment of manifestation*, i.e. the moment the developer is presented with an application inconsistency:
   - *compile time*, detected during compilation of the application;
   - *deployment time*, detected when the application is started or deployed to an application server;
   - *runtime*; detected at the server while the application is running, e.g. when loading a page;
   - or in the *browser*, when an error is only detected when a page is loaded by the client (e.g. mistakes in Javascript, HTML etc.).
2. Is the error *retraceable* to its origin? Is a source code filename and line number clearly indicated?
3. *Clarity* and specificity of error message. Are domain-specific terms used in error messages, or do they uncover the underlying implementations?

2.2.3 *Frameworks*

We evaluate three mainstream, available web application frameworks that represent the state of the practice in web application development. We discuss other web frameworks and languages in Section 2.6. We base our study on parts of example applications and tutorials from the websites of the different frameworks. We apply the technique of *fault seeding* by introducing small inconsistencies in parts of the application (often in the form of simple typing errors, simulating what happens when an application is changed or a developer makes a mistake) and observe how the errors manifest themselves.

The selected frameworks are:

- *Ruby on Rails*[1], representing dynamically typed language frameworks. We chose Rails as a representative of frameworks based on dynamic languages. Other frameworks such as Django for Python are similar in terms of implementation techniques and error handling.

- *JBoss Seam*[2], a framework based on Java, combining a number of existing Java technologies such as the Java Persistence API (JPA) and JavaServer Faces (JSF). We selected JBoss as a representative of Java-based frameworks. A comparable framework is Spring.

- *Lift*[3], a web framework based on Scala, a highly expressive object-oriented and functional programming language with a sophisticated type system. Scala is a statically typed language with a very flexible syntax, distinguishing Lift from the two other categories.

In the remainder section we highlight two faults related to the data model and the user interface. A full overview of the cases we studied is given in Appendix A. We summarize our results in tables that rank the three quality aspects of moment of manifestation, retraceability, and clarity (labeled *M*, *R*, and *C*).

2.2.4 *Case 1: Consistency of References to the Data Model*

	M	**R**	**C**
Rails	Runtime	+	−
Seam	Runtime	−	+
Lift	Runtime	+	+

User interfaces are typically used to present data from a database. Therefore user interface code contains references to the data model, for instance to show the value of a certain property, or binding a control to a certain entity property.

In Ruby on Rails, references from the user interface to data model properties are constructed through embedded Ruby code. The following example displays the value of the `name` property of the `post` entity, encoded to be displayed in HTML:

```
<td><%=h post.name %></td>
```

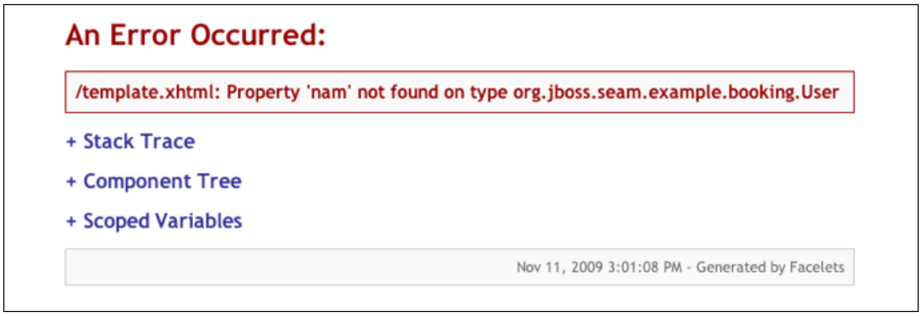
Although references to undefined properties, such as `post.nam` instead of `post.name`, are easily traced back to their source, the reported "undefined method" message is not domain-specific and only reported at runtime.

In Seam, values of entity properties can be injected into a page using the `#{...}` syntax:

```
Welcome #{user.name}
```

When invalid property names are used, a domain-specific runtime exception is reported when the page is loaded ("Property 'nam' not found on type ..."), but no indication of the source of the problem is supplied (see Figure 2.1).

[1]We evaluated version 2.3.4 of Ruby on Rails, `http://www.rubyonrails.org/`.
[2]We evaluated version 2.2.0.GA of Seam, `http://www.jboss.com/products/seam`.
[3]We evaluated version 1.0 of Lift, `http://www.liftweb.net/`.


An Error Occurred:

/template.xhtml: Property 'nam' not found on type org.jboss.seam.example.booking.User

+ Stack Trace

+ Component Tree

+ Scoped Variables

Nov 11, 2009 3:01:08 PM - Generated by Facelets

Figure 2.1 Seam exception when using an undefined property `nam`

In Lift, the `name` property of an entity `user` is referenced as follows:

```
<user:name>User name</user:name>
```

When misspelling `name` as `nam`, Lift gives a clear, domain-specific error ("no such property") and reports the line and column number of the error.

All of the tested frameworks report faults in references to the data model only at runtime, when the specific page is loaded.

2.2.5 *Case 2: Consistency of Links to Pages*

	M	**R**	**C**
Rails	Runtime	+	–
Seam	Browser	–	–
Lift	Browser	–	–

Creating hyperlinks between pages is a fundamental part of the web. While broken links to external websites are hard to avoid, broken links *within* a single web applications should be avoided and, at least in principle, be automatically detected.

Ruby on Rails provides a `link_to` helper for user interfaces:

```
<%= link_to 'Edit', edit_post_path(post) %>
```

The `edit_post_path` method that is called is generated on the fly by convention, the convention taking the form of `<action>_<controller>_path (<args>)`. When the name of this method is constructed incorrectly, a generic "undefined method" error is reported, with accurate code and line and column numbers. This means that the framework is able to detect broken, internal links before they are displayed to the user. However, the error message is not domain-specific.

Seam uses a `s:link` tag to create links to arbitrary URLs. These URLs are not checked by the framework:

```
<s:link id="register" view="/register.xhtml"
        value="Register New User"/>
```

When the linked page does not exist, the user is presented with a "page not found" error when the link is clicked.

	Manifestation			Retraceability			Clarity		
Category	**Ra**	**Se**	**Li**	**Ra**	**Se**	**Li**	**Ra**	**Se**	**Li**
Data model									
Properties of non-existing types	R	C	C	-	+	+	-	+	+
Invalid inverse properties	R	D	C	-	+	+	+/-	+	+
Invalid data validation	R	C/D	C/D	-	+/-	+/-	-	+	+
User interface									
Invalid page elements	R	R	R	+	+	-	-	+	-
Invalid element nesting	B	B	B	-	-	-	-	-	-
Invalid references to data model	R	R	R	+	-	+	-	+	+
Invalid links to pages	R	B	B	+	-	-	-	-	-
Invalid links to actions	R	R	R	-	+	-	+	-	-
Application logic									
Invalid references to data model	R	C	C	+	+	+	-	+	+
Invalid redirect from actions	R	R	R	-	-	-	-	-	-
Invalid data binding	R	NA	NA	-	NA	NA	-	NA	NA
Access control									
References to data model	R	R	C	+	-	+	-	-	+

Ra = Ruby on Rails, Se = Seam, Li = Lift
B = Browser, C = Compile, D = Deploy
NA = Not applicable, R = Runtime

Figure 2.2 A summary of consistency checks in Ruby On Rails, JBoss Seam, and Lift.

Lift does not have a special construct to define internal links, instead simple `<a href="...">` tags are used. Similar to Seam, links to non-existing pages go undetected until they are clicked.

2.2.6 *Summary*

A summary of our results is shown in Figure 2.2. Rather than tally the specific scores of the individual frameworks, we conclude that there are many cases where errors are not reported at the earliest possible opportunity, where errors are not easily traceable to their source, and where error messages are unclear or confusing. In the next section we discuss reasons in the design and implementation of the frameworks that cause these deficiencies.

2.3 FRAMEWORK DESIGN AND FAULT DETECTION

In this section we analyze *why* faults in web applications manifest themselves late in the development process and why failures often have poor retraceability and clarity. The examples of web application inconsistencies in the previous section illustrate that there are many cases where inconsistencies lead to late failure. They may only be reported or otherwise manifest themselves once a definition is *used*, not when it is first compiled or interpreted. In many cases, reported error messages are very generic, revealing details about the implementation of the framework (i.e., revealing leaky abstractions). Error messages also do not always show the origin of the error, as they are reported in various ways and definitions are not directly checked.

The frameworks in our survey have been implemented using different programming techniques and based on different programming languages. In the following subsections we analyze different properties of the frameworks that impact the manifestation of faults.

2.3.1 *Reflection and Run-time Code Manipulation*

Reflection and run-time code generation are common techniques for integration and deployment of components in web application frameworks. Based on the dynamic language Ruby, Rails in particular makes heavy use of these techniques to provide convenient, high-level abstractions. JBoss Seam makes use of reflection techniques to process annotations, particularly to describe the data model.

Ruby on Rails

As a typical example of how the dynamic programming approach of Ruby interacts with how failure manifests itself, consider a one-to-many relationship declaration in an entity:

```
has_many :comments
```

This declaration *implies* there is a `Comment` entity defined elsewhere. When the property is used, the Rails framework simply takes the `comments` symbol, strips off the `s` and capitalizes the first character. If no such entity is defined, the developer will receive a "constant not defined" error related to `Comment`, while the application code does not contain any reference to this entity anywhere directly. These indirect error messages can be confusing to the user of the framework. If entity declarations were instead verified directly when the entity was declared, the error could be detected earlier, and would be more easily traced back to the source. The dynamic programming approach taken by Ruby on Rails involves a trade-off between the performance of not checking such properties and ease of use.

Many features of the Rails framework make use of methods which are passed a map with named arguments. This way, arbitrary key/value pairs can be used as arguments for these methods. When a key is mistyped or there is no definition for such a key (as seen with `:confirmation` in Section A.4.4), such faults remain undetected unless the contents of the map is explicitly verified by the framework. In the current implementation of Rails, this is often not the case.

JBoss Seam

After a JBoss seam application is compiled, framework-specific tools are used to deploy it onto a server environment. Typically, application servers enable web application verification code to be invoked while the application is being deployed. This provides frameworks with the opportunity to perform additional checks that were not already performed by the compiler.

An example of a post-compilation time consistency check is Seam's verification of entity classes and their annotations and embedded regular expressions. Any faults detected in the data model are reported by throwing exceptions. Unfortunately, in practice this seems to cause a domino effect of exceptions being thrown by various components of the application server. This causes enormous stack traces to be recorded in the server logs, in which it is very hard to find the originating error message. Still, by performing these checks while the application is being deployed, Seam avoids run-time failures resulting from certain classes of faults in the data model.

2.3.2 *Linguistic Separation*

The three frameworks each employ one base language: Java, Ruby, or Scala. They also employ a number of other languages, such as XHTML, regular expressions, or query languages. These languages are linguistically separated in the sense that the compiler for the base language is not aware of the definitions made in the other languages and whether or not they are consistent and correct. Because the compiler cannot pick up these inconsistencies, they can lead to failures as an application is running.

Conceptually, it is appealing to use different languages that each address different technical concerns: each language can be more or less suited for that particular domain. Unfortunately, as these languages have been designed and have evolved separately, there can be redundancy and inconsistency among them. The EL expression language used in JBoss Seam, for example, does not support all features of standard Java expressions, yet it adds some features of its own.

Separate languages also introduce a problem for programming tools, as tools that support one language lack awareness of other languages that are used in a web application. Editors and compilers generally only have a limited "view" of a web application, constrained by the boundaries of a particular language. They do not check inside strings, determine the meaning of annotations, or analyze accompanying XML or XHTML files. Consistency checking for concerns that cross the boundaries of a language – understanding-in-the-large of a web application – is very hard when different languages are used. Only tools that are specialized to work with a particular set of languages and frameworks (such as IntelliJ IDEA, discussed in Section 2.6.2) can check for some of these consistency issues. However, as the different languages, frameworks, and tools involved are developed by different groups of people, such a solution is very hard to maintain and even harder to make complete.

Links and redirects in the three frameworks are constructed as simple URL strings. Only in Rails, where links can be constructed using helper methods, are internal links checked for correctness at run-time. The other frameworks do not support any form of consistency checking: bad links only manifest themselves when the user tries to follow them.

2.3.3 *Limited Static Type Checking*

Faults manifest themselves at a variety of different stages: at compile time, deployment time, run time, or sometimes only in the browser. Failures early in the development cycle typically require less effort to resolve. Faults that are detected directly at compile time do not require failure-to-fault tracing or running the application to be detected.

Seam and Lift benefit from their statically typed base languages with respect to compile-time detection of faults, while Rails can only provide developers with feedback about faults at runtime. In our study we found that there are a number of negative performance trade-offs when delaying checks until run-time, and that accurately discovering and reporting the origin of errors can be difficult. Still, there were many cases where the Seam and Lift frameworks did not score much better at providing early feedback.

Since Rails is based on Ruby, there is no compilation step, and consistency errors that are reported are always detected at run time. Still, we can distinguish between errors reported when a definition is interpreted and when the definition is used. In many cases, errors are only reported when definitions are used. In our experience, the framework performs very few checks when definitions are made, before they are used elsewhere. When errors are reported, the messages are usually generic Ruby messages (typically, a `NoMethodError`).

Based on compiled, statically typed languages, Lift and Seam can report many errors before an application is deployed. Errors detected by the Java and Scala compiler always clearly indicate their origin. Using an IDE such as Eclipse, compile-time errors can be conveniently marked in the source code using a marker in the editor. Still, the reported error messages are always generic Java or Scala error messages, as the compiler and IDE only follow the static semantics of the host language. Because of this limitation, any language features encoded in strings, such as embedded queries or regular expressions, cannot be checked. Likewise, any references to other elements of an application in the form of strings (such as in the Seam `@OneToMany` annotation) cannot be statically checked. The Java and Scala host languages also do not offer a way to statically constrain the placement of annotations on the right elements of an application, or to avoid conflicting annotations.

A problem with relying on the static type system of the base language is that the errors reported are not specific to the domain of web programming. For instance, instead of reporting an error about an entity property, reported errors may complain about the field of a class. Since Seam and Lift are frameworks and not true languages on their own right, reporting domain-specific error messages is very difficult. Only by the construction of extensions to the already elaborate Java or Scala compilers would it be possible to check such frameworks. Building such extensions is generally a difficult, laborious undertaking, especially for frameworks that rely on reflection techniques and linguistic separation. In Section 2.6.2 we discuss tools that follow this approach in more detail.

2.3.4 *Run-time consistency checking*

Most faults not detected by the compiler or at deployment time are reported at runtime. Some errors are reported directly when a definition is processed by the runtime, others only in particular use cases of the application, manifesting themselves only when a particular action is performed by the user. Such delays in detection are detrimental for developer productivity and, as regressions may go undetected when not covered by the test suite, the maintainability of an application.

From a framework implementation point of view, run-time consistency checks – at least in principle – make it easy to report accurate, highly specific error messages. However, in practice, traceability of these errors is often lacking, as source location information at run time is scarce, usually limited to the point in the application where the check was performed. There are often many framework calls in between the location of the error and the point where the error is detected, resulting in runtime traces that can be misleading or confusing. Our survey in Section 2.2 showed that the quality of runtime error messages and their traceability varies widely and is typically worse than for compile-time reported errors.

Seam and Lift perform static checks at compile-time using the standard Java and Scala compilers and perform a limited set of consistency checks at deployment time. This leaves it up to the runtime to perform the remainder of the checks. Thorough, often domain-specific checks that are not performed earlier are performed at run time. These checks guarantee the correctness of any strings in annotations and of string-embedded languages. Both frameworks run on the Java Virtual Machine and use the Java exception tracing mechanism for reporting the origin of such errors. For run-time checks, some of these reported origins relate to the usage sites of inconsistent definitions, but as a last resort they are still helpful in determining the root cause of an error.

Location information provided by exceptions is ineffective for checks that are not performed at the definition site where an error is triggered. This makes it particularly difficult to report accurate location information for errors in annotations, which are heavily used especially in Seam. The Java language provides few means to provide exact location information when the annotations are reflected over at runtime. At most, a class and method name can be provided in any annotation errors that are reported.

2.3.5 *Summary*

Providing accurate, static checks at compile-time avoids failures at deployment-time or at run-time. Statically detected faults do not require failure-to-fault tracing and can be reported directly inside an IDE. Still, there are many classes of faults that are not statically detected by the frameworks in our survey. Reasons for this include that they use reflection and run-time code manipulation techniques, linguistically separated languages, and can only use static typing

UI	Language for defining HTML user interfaces
Data models	Language to define persistent data models
Action	Simple language for defining application logic
Access control	Access control rules for specifying the access control policy
Validation	Data validation language
HQL	Database query language
Workflow	Language for defining workflows

Figure 2.3 WebDSL sublanguages

provided in the base language compiler. Instead, many faults are reported at run time, introducing a (small) performance penalty and often resulting in errors that are vague or hard to trace back to their originating fault.

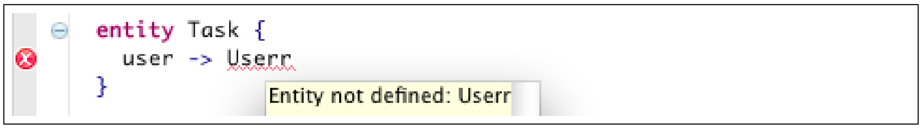
2.4 DESIGNING FOR STATIC VERIFIABILITY

In the previous sections we demonstrated the problems of weak static verification of web applications. We concluded that the cause of this weakness is in the design of the programming languages and frameworks. Static verifiability is an afterthought, delegated to third party tool developers or coped with by test-driven design methodologies. Because static verifiability is not a criterion during design, the resulting language will end up being hard to verify. Our solution is designing a web programming language with static verifiability in mind as exemplified in WebDSL.

WebDSL embraces the notion of having different, specialized languages to address separate concerns. WebDSL provides specialized languages for data modeling, user interface design, and basic data operations. However, through *linguistic integration*, these different languages are combined into one large integrated language. Figure 2.3 illustrates the key domain-specific languages that together form WebDSL. The languages are seamlessly integrated, follow the same style of syntax and share common elements, and can be used together in one module, if required.

WebDSL and its sublanguages have been designed as statically checked languages: the *moment of detection* of all consistency checks is at compile time. In fact, using the new WebDSL Eclipse plug-in, errors are detected as the developer writes his code. As the checks are performed directly on the source code, rather than on a deployed application, any reported errors directly relate to the source code, ensuring proper *retraceability*. Finally, since errors relate to the domain-specific WebDSL language – and not a general-purpose language with a web framework on top – all errors are domain-specific and are explained in terms of the web application domain rather than in terms of the underlying implementation.

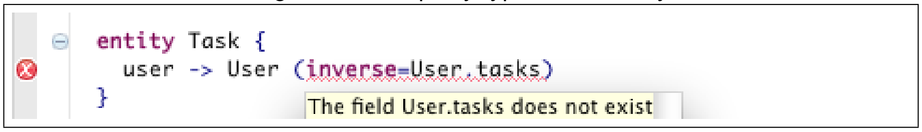
For a general description of WebDSL, we refer the reader to our previous work [Visser, 2008, Groenewegen and Visser, 2009], Chapter 4 and Chapter 3. This section will highlight design decisions where static verifiability was taken into account, in particular the categories from Figure 2.2 will be addressed.



```
entity Task {
  user -> Userr
}
```

Entity not defined: Userr

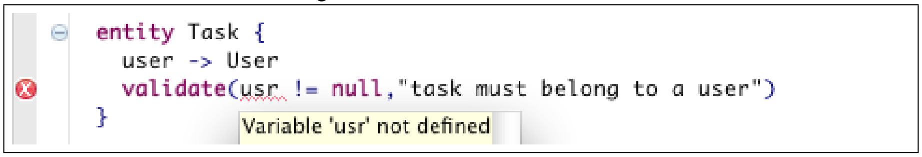
Figure 2.4 Property type consistency



```
entity Task {
  user -> User (inverse=User.tasks)
}
```

The field User.tasks does not exist

Figure 2.5 Inverse annotation



```
entity Task {
  user -> User
  validate(usr != null,"task must belong to a user")
}
```

Variable 'usr' not defined

Figure 2.6 Data validation

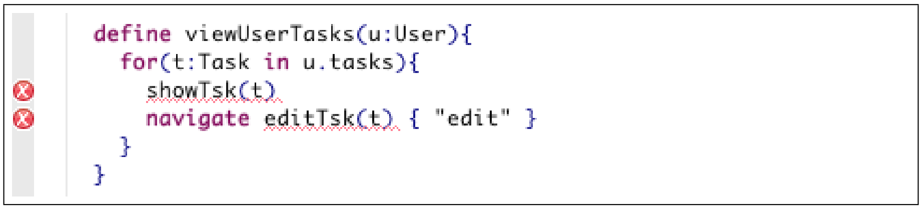
2.4.1 *Data Model*

Data model entities are *first-class language elements* in WebDSL. They are defined as uniquely named top-level elements. The properties of data model entities are statically typed, they can refer to built-in simple types or to defined entities. A shared, static type system across WebDSL sub-languages enables static verification of the use of existing types and properties. Designing the language with entities as first-class language elements enables reporting of domain-specific error messages.

Figure 2.4 illustrates the editor feedback when a non-existing type is referenced in a property in WebDSL. Similarly, Figure 2.5 shows that this check also holds for inverse relations. Figure 2.6 shows checking of references to entity properties from validation rules.

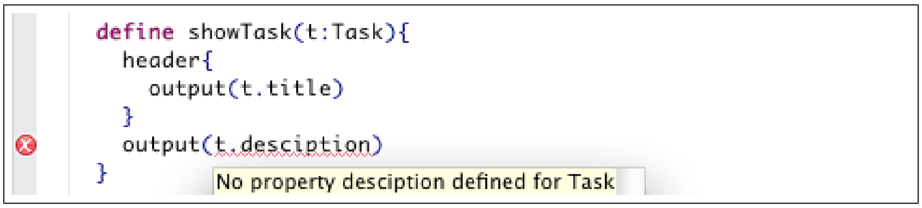
2.4.2 *User interface*

User interfaces in WebDSL are defined using page template definitions. Like data model entities, template definitions are declarative first-class language elements in WebDSL. Templates can call other built-in or user-defined templates. Navigation between pages is expressed using `navigate` elements which create links to other pages within the application. Rather than constructing links through string concatenation, links are defined as typed page calls, for which can be verified that they exist and that the number and type of their arguments are correct. Figure 2.7 shows how mistakes in template calls and navigates are reported. Output elements form references to the data model for displaying data (the `output` template name is overloaded for each type). Figure 2.8 illustrates that such references are checked as well.



```
define viewUserTasks(u:User){
  for(t:Task in u.tasks){
    showTsk(t)
    navigate editTsk(t) { "edit" }
  }
}
```

Figure 2.7 Template call and navigation



```
define showTask(t:Task){
  header{
    output(t.title)
  }
  output(t.desciption)
}
```

No property desciption defined for Task

Figure 2.8 Template reference to data model

2.4.3 *Application Logic*

While the model-view-controller pattern is generally considered good style, WebDSL does not impose the use of this pattern in the language. Instead, WebDSL applications typically encapsulate small snippets of application logic directly in user interface code as page actions. Larger pieces of logic can be defined separately in functions. The sublanguage used in these page actions and functions is a Java-like imperative language with a simple API, fully checked by the WebDSL typechecker. Figure 2.9 shows a small template that will result in a form with two input fields. Data binding is automatic, any input in the form will update the data model before the action is executed. Redirecting the user to a different page after an action has succeeded is done using the built-in `return` construct. A `return` construct, similar to a `navigate` in the user interface language, takes a page call as its argument.

The incorrect action reference `sav` is reported, as is the page reference `showUserTsks` inside the action.

2.4.4 *Access Control*

The access control policy of a WebDSL application is defined in access control rules. The access control language reuses the expression language (and its checks) also used in the user interface and application logic. In addition, the page signature syntax is the same as for defining pages, enabling the verification that a rule in fact matches an existing page with correct signature.

Figure 2.10 shows how a missing property of the data model is reported.

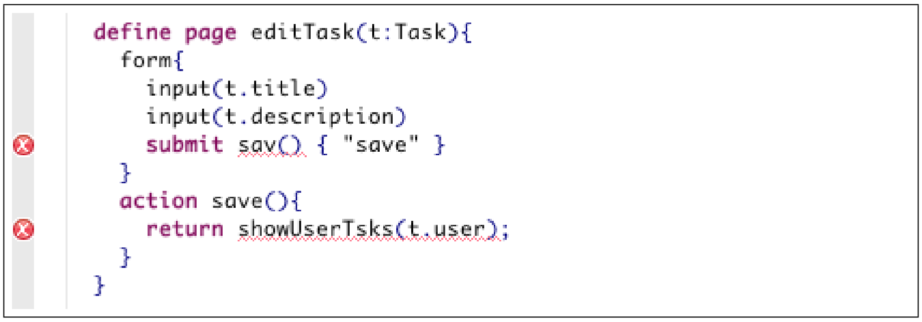

```
define page editTask(t:Task){
  form{
    input(t.title)
    input(t.description)
    submit sav() { "save" }
  }
  action save(){
    return showUserTsks(t.user);
  }
}
```

Figure 2.9 Action logic

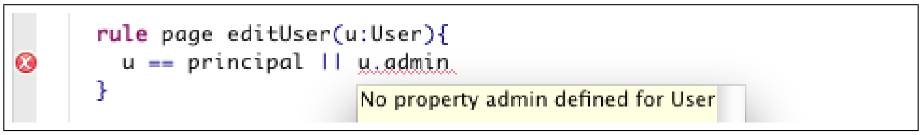

```
rule page editUser(u:User){
  u == principal || u.admin
}
```
No property admin defined for User

Figure 2.10 Access control

2.4.5 *Verifiability versus Flexibility*

Designing for verifiability requires a trade-off with flexibility. Verifiability should be part of the language design considerations, but may impede coverage, i.e. the range of programs that can be expressed. As an example, consider verification of navigation in WebDSL. The interaction between page definitions and navigate statements is verified by controlling the URLs that are generated for pages, and thus required for links to those pages. That is, a URL for a page consists of the name of the page followed by the (identities) of the arguments separated by slashes. For most applications that results in nice readable URLs. If a developer wants to implement a more dynamic scheme this can be realized by creating a single page definition that interprets the URL parameters and dispatches to some appropriate template definitions. However, this results in a loss in the effectiveness of static verification; navigates become calls to the generic dispatch page, rather than to a specific page, which requires the developer to deal with parameter encoding/decoding and verifying consistency. For applications where such flexibility is a requirement, the current WebDSL design is not optimal; it would be better to generalize the current page/navigate paradigm to declaratively specify dispatch schemes that are verifiable.

In practice, WebDSL's verifiability does not impede coverage. The language is used for several web applications that are in production. The largest and most complex WebDSL application to date is researchr[4]. Researchr is a digital library with over a million publication records, including BibTeX import and export, bibliographies, reviewing, tagging, a reputation system, groups, and a messaging system. Researchr's data model consists of over a hundred entities

[4]`http://researchr.org`

(represented by 140 database tables) and the complete application consists of about 18,000 lines of WebDSL code. The static consistency checking scales to the size of the application, and is even a pre-condition for its maintainability; making changes is not scary since consistency faults introduced are detected at compile-time.

YellowGrass[5] is a free web-based issue tracker. Internally we use it to track WebDSL bugs and other projects within our group use it as well. The official WebDSL website[6] has been built using WebDSL. It features an editable manual with revision control.

2.5 RULE-BASED CONSISTENCY CHECKING

In the previous sections we have argued that static consistency checks for a linguistically integrated web programming language provide better and earlier feedback to developers. In this section we show that this can be realized using a high-level rule-based specification. We give a formal definition of automatic consistency checking for a subset of WebDSL using rewrite rules in Stratego [Bravenboer et al., 2008] following the style developed for type checking introduced in Chapter 4. We give a brief introduction to Stratego and the style of consistency checking employed in the chapter. The concrete syntax of WebDSL is defined using SDF grammars [Visser, 1997b], but in this chapter we focus only on the abstract syntax and semantics of the language.

2.5.1 *Language Definition*

We illustrate static consistency checking in WebDSL using a subset of the full language focusing on the two examples from Section 2: references to the data model in user interface templates, and consistency of references to user interface templates and pages. Figure 2.11 defines the abstract syntax of the subset of WebDSL we are considering using an algebraic signature, which consists of typed term constructors corresponding to language constructs. Figure 2.12 illustrates the definition with the concrete and abstract syntax of a fragment of a WebDSL program.

The data model of a WebDSL program is defined using *entity declarations* (`Entity`), which consist of a name and a list of properties (`Property`), each having a name and a type. Expressions are constants (`StringLit`), variables (`Var`), or access to the values of properties of objects (`PropertyAccess`).

The user interface of a WebDSL program consists of *template definitions* (`TemplateDef`), which have a name, list of parameters, and list of template elements. The elements compose the output of the template from the objects passed as parameters. This is mostly achieved by reference to other templates. Some of these templates are primitives. For example, the `output` template presents the value of an object, and the `input` template is used to create input form elements.

[5]`http://yellowgrass.org`
[6]`http://webdsl.org`

```
signature
  constructors
    Module        : ID * List(Definition) -> Module

    // data model
                  : Entity -> Definition
    Entity        : ID * List(Property) -> Entity
    Property      : ID * Type -> Property
    SimpleType    : ID -> Type

    StringLit     : STRING -> Exp
    Var           : ID -> Exp
    PropertyAccess : Exp * ID -> Exp

    // user interface templates

                  : Template -> Definition
    TemplateDef   : List(Mod) * ID * List(Param) *
                    List(Element) -> Template
    Page          : Mod
    Param         : ID * Type -> Param

    String        : STRING -> Element
    Navigate      : PageRef * List(Element) -> Element
    Call          : TemplateRef * List(Element) -> Element
    TemplateRef   : ID * List(Exp) -> TemplateRef
    PageRef       : TemplateRef -> PageRef
```

Figure 2.11 Signature for NWL, a subset of WebDSL

Template *page* definitions have the `Page` modifier and produce a complete web page. Non-page template definitions define partial pages that are used to compose pages. There are two ways in which template definitions refer to other template definitions. A template call (`Call`) *inlines* the body of a referenced template in the calling template. A page reference (`PageRef`) is used to produce a link to *navigate* to the corresponding template (which must be a page definition).

2.5.2 *Static Consistency Checking*

The language is designed to support static consistency checking. References to other elements of a program are *explicitly* encoded in the syntax of the language. For example, instead of encoding an expression retrieving the value of a property of an object as a string literal, the user interface language can use expressions to produce such values. The identifiers used in these expressions are typed and property accesses can be checked against the data model. Similarly, references to user interface templates are explicit calls that can be checked for existence of the called template and the proper typing of the ar-

```
module blogpost
entity Post {
  title   : String
  text    : WikiText
  author  : User
}
define page post(p : Post) {
  header{ output(p.title) }
  output(p.text)
  navigate editpost(p) { "Edit" }
}
```

```
Module("blogpost",[
  Entity("Post",
    [Property("title",  SimpleType("String")),
     Property("text",   SimpleType("WikiText")),
     Property("author", SimpleType("User"))]
  ),
  TemplateDef([Page()], "post", [Param("p",
                                   SimpleType("Post"))],
    [Call(TemplateRef("header"),
              [Call(TemplateRef("output", [
                     PropertyAccess(Var("p"),
                     "title")]))]),
     Call(TemplateRef("output", [PropertyAccess(Var("p"),
                                  "text")])),
     Navigate(PageRef(TemplateRef("editpost", [Var("p")])),
       [String("Edit")])]
  )
]
```

Figure 2.12 Concrete and abstract syntax for fragment of a WebDSL program.

guments passed; in contrast to the composition of URLs from strings (which is akin to pointer manipulation in C).

The WebDSL compiler translates WebDSL programs to Java programs. Before code generation, the source code is statically checked for consistency violations. WebDSL is also supported by an Eclipse editor plugin, which displays error messages and warnings in the editor, providing immediate feedback about consistency errors to the developer (see previous section). Code generation and static checking in the compiler and in the Eclipse plugin are implemented in the Stratego transformation language.

Static checking is divided into three parts. *Name resolution* determines which identifier uses refer to which declarations. *Type analysis* computes types (and other properties) of composite expressions. *Consistency checking* applies constraints to sub-terms, producing error messages when violations are encountered. In the next subsection we give a brief introduction to Stratego. In the following subsections, we discuss the definition of name resolution, type analysis, and consistency checking.

2.5.3 *Stratego*

Stratego is a language for program transformation based on the paradigm of term rewriting with programmable rewriting strategies introduced by Visser et al. [1998]. Stratego transformations operate on first-order terms of the form

```
t ::= x                  // variables
    | "..."              // string literals
    | i                  // integer constants
    | c(t1,...,tn)       // constructor applications
    | [t1,...,tn]        // lists of terms
    | (t1,...,tn)        // tuples of terms
```

Basic transformations are defined by means of conditional *term rewrite rules* of the form

```
r : t1 -> t2 where s
```

with `r` the name of the rule, `t1` and `t2` first-order terms, and `s` a *strategy expression*. A rule applies to a term when its left-hand side t1 matches the term, and the condition `s` succeeds, resulting in the instantiation of the right-hand side pattern `t2`. Otherwise the application *fails*.

In addition to checking applicability constraints, the condition of a rule can perform computations the results of which are used in the right-hand side of the rule. For example, in the rule schema

```
r : t1 -> t2 where t3 := <s> t4
```

the term `t4` possibly containing variables from `t1` is transformed by the application of a strategy `s` and the result is matched against the pattern `t3`, possibly binding variables, which may be used in the right-hand side `t2`.

More complex transformations can be created by composing rules using *strategies*. A strategy is essentially a partial function from terms to terms. If a strategy is not defined on a term it is said to *fail*. Failure arises from the failure of rewrite rules to apply to terms. Strategies are composed from basic combinators such as the identity transformation `id`, sequential composition `s1; s2` and deterministic choice `s1 <+ s2`. From these basic combinators new combinators can be defined using (parametric) strategy definitions. For example, the definitions

```
try(s) = s <+ id
repeat(s) = try(s <+ repeat(s))
```

define the combinator `try(s)` that attempts to apply a strategy `s` to a term, and restore the term if `s` fails, and `repeat(s)` that applies a transformation `s` as often as possible to a term. While the strategies above apply a transformation to the root of a term, *term traversal* strategies apply transformations to sub-terms. The basis of term traversal strategies are *one-level* traversal operators such as `all(s)`, which applies a strategy `s` to each direct sub-term of a term. For example, the definitions

```
bottomup(s) = all(bottomup(s)); s
alltd(s) = s <+ all(alltd(s))
```

introduce the `bottomup(s)` strategy that applies `s` to each sub-term in a bottom-up (post-order) fashion, while `alltd(s)` applies `s` to an outermost frontier for which `s` succeeds.

Context-sensitive transformations can be expressed by means of *dynamic* rewrite rules [Bravenboer et al., 2006b], which are instantiated at run-time, as illustrated by the following schema:

```
r : t1 -> t2
    where rules( dr : t3 -> t4 )
```

The dynamic rule `dr` is defined when `r` is applied to a term matching `t1`. Any variables that `t3` and `t4` share with `t1` are then inherited by the instantiation of `dr` (concrete examples follow below).

2.5.4 *Name Resolution*

In textual software languages, program units are identified by name — hence, names are known as identifiers. Declarations introduce names and definitions bind names to meanings — often declarations and definitions are combined in one construct. Definitions are applied by invoking their name. In the language of Figure 2.11 there are four kinds of identifiers. Entity declarations introduce named entities. Properties identify the attributes of entities. Template definitions identify user interface components. Template parameter names identify their arguments. Corresponding to these declarations, we have the following uses of identifiers. Type expressions are references to entities (and primitive types). Variables are references to entity objects (or primitive values). Property access expressions retrieve the value of a property of an object. Template references invoke a template.

An important source of inconsistencies is the use of names that do not correspond to definitions, or the use of names of existing definitions in the wrong place or in the wrong way. Thus, the first task of a consistency checker is to resolve the use of names, identifying for each application which declaration it invokes. We distinghuish two types of identifiers, i.e. identifers with global scope and identifiers with local scope. We can distinghuish further layers, associating name spaces with modules, but we will ignore such layers here, but note that they can be expressed with the same approach.

The rules in Figure 2.13 define name resolution for the top-level definitions in our language, that is entity declarations and template definitions. The `declare-def` rules introduce the dynamic rules `EntityDeclaration` and `Template`, mapping identifiers to definitions. The `EntityDeclaration` rule maps the name of an entity to the complete abstract syntax representation of the corresponding entity declaration. Note that `x@t` denotes a simultane-

```
strategies

  declare-all = alltd(declare-def); rename-all

rules

  declare-def :
    ent@Entity(x, prop*) -> Entity(x, prop*)
    with rules( EntityDeclaration : x -> ent )

  declaration-of :
    SimpleType(x) -> <EntityDeclaration> x

  declare-def :
    def@TemplateDef(mod*, x, param*, elem*) ->
    TemplateDef(mod*, x, param*, elem*)
    with sig := <signature-of> def;
         rules(
           Template : x -> def
           Template : sig -> def
         )

  signature-of :
    TemplateDef(mod*, x, param*, elem*) ->
    (x, <param-types>param*)

  param-types :
    TemplateDef(mod*, x, param*, elem*) ->
    <param-types> param*

  signature-of :
    TemplateRef(x, e*) -> (x, t*)
    where t* := <map(type-of)> e*

  declaration-of :
    ref@TemplateRef(x, e*) -> def
    where def := <signature-of; Template> ref
```

Figure 2.13 Name resolution for top-level declarations

ous match to a variable (x) and a term pattern (t). The declaration-of rule maps a type expression to the corresponding entity declaration, *provided* the EntityDeclaration rule is defined for the type name. If not, the declaration-of rule simply fails. Note that the order in which rules are important: the rule that defines the EntityDeclaration rule has to be executed before the declaration-of rule is executed.

Similarly, the Template dynamic rule maps the name of a template definition to its complete AST representation. Since non-page template definitions can be overloaded there is also a mapping from the *signature* of a template

```
strategies

  rename-all = alltd(rename)

rules

  rename :
    Param(x, t) -> Param(y, t)
    with y := <rename-var>(x, t)

  rename-var :
     (x, t) -> y
     with y := x<new>
     with rules(
            RenameId : x -> y
            TypeOf   : y -> t
          )
  rename :
    Var(x) -> Var(y)
    where y := <RenameId> x

  rename :
    TemplateDef(mod*, x, param1*, elem1*) ->
    <declare-def> TemplateDef(mod*, x, param2*, elem2*)
    with {| RenameId:
            param2* := <rename-all> param1*;
            elem2*  := <rename-all> elem1* |}
```

Figure 2.14 Name resolution for local identifiers

to its definition. The signature of a template definition is a pair of its name and the list of its parameter types. The `declaration-of` rule produces the template definition corresponding to a template reference by computing its signature. Computing the signature of a template reference requires type analysis (`type-of`) to determine the type of the argument expressions. The `declare-all` strategy applies the `declare-def` rules to all top-level definitions, using the `alltd` strategy, thus creating dynamic rule mappings for each.

For the identifiers with global scope we have assumed that for each identifier (or signature) there is a single declaration that corresponds to it. Identifiers with local scope are different in that an identifier can be used in multiple scopes, corresponding to different declarations. In the language of Figure 2.11, the only local identifiers are the names of template parameters. The same parameter name can be used in multiple template definitions. To distinghuish multiple uses of the same identifier, name resolution of locally scoped identifiers is implemented as a *transformation* that *renames* these identifiers to a unique name.

Figure 2.14 defines the `rename-all` strategy defining renaming for our web language, applying a top-down traversal looking for terms that it can apply the `rename` transformation to. The `rename` rules transform identifier declarations and uses to use unique names. The rule for `Param` renames a template parameter to a unique name using the `rename-var` rule, which given an identifier `x` and a type `t`, creates a unique new name `y`, which is `x` with as annotation a freshly created string. Thus, we create a new unique term, but retain the original name of the identifier for use in error messages. Furthermore, `rename-var` defines dynamic rule `RenameId` to rename the original identifier to its new name, and `TypeOf` that maps the new identifier to its type `t`. The `rename` rule for variables (`Var`) uses the `RenameId` rule to replace a variable `x` with the corresponding unique name `y`.

To actually distinghuish identifiers defined in different scopes, the `rename` rule for `TemplateDef` uses a dynamic rule scope (`{|R:s|}`) to limit the bindings of the `RenameId` dynamic rule to the traversal of the template elements in the body of the definition.

2.5.5 *Type Analysis*

After name resolution we can map identifiers to their declarations (or types). Expressions compose new things (values, templates) from basic things (constants) and the things represented by identifiers using composition operators. Type analysis computes the type of such expressions so that we can determine if these compositions are consistent with the internal or user-provided definition of operators. The language of Figure 2.11 has only simple expressions, consisting of string literals, variables, and property access. The other kind of expressions are the template `Element`s. Their composition is checked directly by consistency checking rules below.

```
rules

  type-of :
    StringLit(x) -> SimpleType("String")

  type-of :
    Var(x) -> t
    where t := <TypeOf> x

  type-of :
    PropertyAccess(e, f) -> t2
    where t1 := <type-of> e
    where ent := <declaration-of> t1
    where Property(f, t2) := <lookup-property(|f)> ent

  lookup-property(|f) :
    Entity(x, prop*) -> <fetch-elem(?Property(f,_))> prop*
```

Figure 2.15 Type analysis

Figure 2.15 defines the `type-of` rule, which computes the types of expressions. The type of a string literal is `String`; other constants are treated similarly. The type of a variable is the type from its declaration, which we obtain using the `TypeOf` rule. The type of a property access `e.f` is determined by first computing the type `t1` of `e`. The declaration of that type is some entity `ent`, which should have a property `f` with type `t2`, which is the type of `e.f`. Any of the steps in this computation may fail; `e` itself may not have a type, the type `t1` may not be declared, or the corresponding entity may not have a property named `f`. In all these cases the application of `type-of` fails.

2.5.6 *Consistency Checking*

Name resolution and type analysis set the stage for definition of consistency checking rules. The `check` rules in Figure 2.16 and Figure 2.17 define the main constraints for our language, *and* produce an error message explaining the failure to comply to a constraint. For brevity we have omitted rules that check unique definitions, e.g. that a name can be used for at most entity, or that an entity may not have two properties with the same name.

A constraint checking rule is a regular Stratego rule of the following general form:

```
check :
  context → (target, message)
  where assumption
  where assumption
  where require(constraint)
```

The rule applies to some `context`, i.e. a subterm of the program we are checking. The `where` clauses first test some (zero or more) assumptions about the context. If these assumptions hold, the `constraint` is tested. If the constraint *fails*, the check rule *succeeds*, i.e. an error has been detected — `require` is an alias for `not`. If an error is found, the rule returns a pair of the `target`, a subterm of `context`, and an appropriate error message. The `analysis` strategy in Figure 2.16 defines the static consistency checking for our language. It first applies the `declare-all` name resolution strategy to the program, and then collects all consistency violations by applying the `check` rules using the `collect-all` strategy. Note that check rules can be defined without dependency on a particular traversal or order of application; all context information needed to check the assumptions and constraint are provided by name resolution and type analysis rules.

Rules 1–4 define definedness of types, variables, property access, and template references. The remaining rules check further consistency properties of template references. Rules 5–7 check the types and arity of the arguments of template references. Rule 8 checks that links (`PageRef`) are to page definitions and not to internal templates. Rule 9 gives a warning if a template inlines a page definition. Rule 10 checks that the parameter of a call to the primitive `input` template is an l-value, i.e. an assignable expression.

```
strategies

  analysis = declare-all; collect-all(check)

rules

  check :                                                     // 1
    t@SimpleType(x) → (x, $[Type '[x]' is not defined])
    where require(<is-simple-type> t)

  check :                                                     // 2
    e@Var(x) → (<id>, $[Variable '[x]' not declared])
    where require(<type-of>e)

  check :                                                     // 3
    e1@PropertyAccess(e2, f) →
    (f, $[[<pp>t] has no property '[f]])
    where t := <type-of> e2
    where require(<type-of>e1)

  check :                                                     // 4
    TemplateRef(x, e*) →
    (x, $[Reference to undefined template '[x]'])
    where not(<is-primitive-template> x)
    where require(<Template> x)

  check :                                                     // 5
    ref@TemplateRef(x, e*) → errors
    where not(<declaration-of>ref)
    where def := <Template> x
    where errors := <zip; filter(check-arg); not(?[])>
                      (e*, <param-types> def)

  check-arg :                                                // 6
    (e, t) →(e, $[Argument of type '[<pp>t]' expected
                 (not of type '[<pp>t2]')])
    where t2 := <type-of> e
    where require(<eq>(t, t2))

  check :                                                    // 7
    ref@TemplateRef(x, e*) →
    [(x, $['[x]' expects [l] arguments; [k] provided])]
    where not(<declaration-of>ref)
    where def := <Template> x
    with k := <length>e*
    with l := <param-types; length> def
    where require(<eq>(k, l))
```

Figure 2.16 Consistency checking rules.

```
check :                                                     // 8
  PageRef(ref@TemplateRef(x, e*)) →
  [(x, $[Navigation to template (not a page)])]
  where def := <declaration-of> ref
  where require(<is-page-def> def)

constraint-warning :                                        // 9
  Call(ref@TemplateRef(x, e*), elem*) →
  [(x, $[Page definition is used as template])]
  where def := <declaration-of> ref
  where require(not(<is-page-def> def))

check :                                                     // 10
  Call(TemplateRef("input", [e]), []) →
  (e, $[Argument of input should be variable or
        property access])
  where require(<is-lvalue> e)

is-lvalue = ?Var(_) <+ ?PropertyAccess(_,_)
```

Figure 2.17 Consistency checking rules (cont.)

Note that checking of terms is *context-free*, i.e. all occurrences are checked irrespective of their context. For instance, the use of expressions as arguments of template calls is covered by rules for expressions. It is not necessary to define a rule checking that arguments to a template reference are well-typed expressions; only the interaction between the expression and the template reference needs to be checked.

2.5.7 *Summary*

We have illustrated how a language design that integrates sub-languages covering different (technical) domains allows checking of their consistent use. A key property of the language design is to choose explicit representations of elements, instead of programmatic encodings; e.g. explicit page references instead of string manipulation to construct URLs make it possible to check that only links to existing page definitions are created.

Given such a language design, the verification of the consistency of a web application can be expressed using declarative consistency checking rules comprising of name resolution, type analysis, and check rules composed by strategies.

2.6 DISCUSSION AND RELATED WORK

2.6.1 *Consistency Checking Capabilities Integrated Into Languages and Frameworks*

Cooper et al. [2006] describe Links, another domain-specific language for the web. Similar to WebDSL, it consists of a number of sublanguages that are linguistically integrated and are compiled to a combination of server and client-side code. Although the language is statically typed, the paper does not describe static verification of Links applications.

Meijer et al. [2006] developed LINQ for the .NET platform. Language INTegrated Query is an extension of C# and VB.NET that provide a generic query syntax that aims to replace string-encoded SQL queries and other types of query languages such as XPath for XML. LINQ queries are statically verified by the compiler. While LINQ is a good first step, other string encoded languages remain on the .NET platform, such as regular expressions. Other general purpose languages with powerful type systems are powerful enough to add database query support as an internal DSL, type-checked by the host language. Spiewak and Zhao [2009] demonstrate how this can be achieved with Scala and Bringert et al. [2004] how it can be done with Haskell. However, error messages of the latter two frameworks are be expressed in terms of Scala and Haskell type errors, rather than domain concepts.

Brabrand et al. [2002] introduced Bigwig, a domain-specific language for developing interactive web applications, which they call web services. One of the core ideas of Bigwig is that its services are session based. The services are not viewed as a collection of pages but as sequences of interactions between client and server. Such an abstraction avoids the broken page link issue discussed in Section 2.2, while limiting URL flexibility. The Bigwig compiler provides a number of static guarantees. Particularly interesting are the guarantees about dynamically created documents. The compiler checks that input fields always match the code that receives the input, i.e. each name property of an `<input>` tag should be handled by server-side code [Sandholm and Schwartzbach, 2000]. This particular problem does not apply to WebDSL, because such input names are generated by the compiler. Besides guarantees for form inputs, Bigwig also guarantees that all documents being generated dynamically are valid XHTML 1.0, as described by Brabrand et al. [2001]. WebDSL enforces consistency checks for many HTML elements, but not a strict XHTML compliance, which is future work (see Section 2.6.5). The successor to Bigwig, JWIG [Möller and Schwarz, 2009], does not add additional types of analysis. The difference is that the analysis is applied in the context of a Java embedding instead of an external DSL.

Thiemann [2002] describes WASH/CGI, a Haskell library to build web applications. The Haskell type system is used to statically verify certain application properties, such as navigation links. This is easy to do, because pages in WASH are just functions, and navigation links are function calls. We downloaded WASH, but were not able to compile and test it. However, we suspect that not all application code is checked statically. For instance, callback attributes contain Haskell expressions embedded in strings. In addition, because the Haskell compiler does not know about domain-specific concepts such as pages, the error messages will not be expressed in domain terminology, but rather in terms of the Haskell type system.

In 1996 already Atkins et al. [1999] discussed the advantage of domain-specific languages in terms of static verification of web applications. The language they proposed, MAWL, enables the definition of form-based applications and performs static checks between the definition of views and the application's logic. However, Mawl is very limited in the aspects it covers, it only covers logic and user interface definitions. It does not cover aspects such as access control, data modeling, data validation and workflows with multiple participants.

The WebDSL language we described is designed to generate full-featured web applications from a single, high-level specification. In contrast, several model-driven methodologies for creating web applications have been proposed in recent years, including OOHDM [Schwabe et al., 1996], SHDM [Lima and Schwabe, 2003], WebML [Ceri et al., 2000], UWE [Koch et al., 2001], OOWS [Pastor et al., 2003], and Hera [Vdovjak et al., 2003]. Many of these model-driven methodologies have evolved into tools that provide partial code generation, for example UWE4JSF [Kroiss et al., 2009] for UWE, HyperDe [Nunes and Schwabe, 2006] for SHDM, WebRatio [Brambilla et al., 2007b] for WebML, OOWS [Valderas et al., 2007], and Hera-S [van der Sluijs et al., 2006] for Hera. These solutions generate only a skeleton application that targets a conventional web application platform. Developers can edit these, relying on these frameworks (as discussed in Section 2.2) to perform consistency checking for the application as a whole. The model-driven solutions used to design the skeletal application can only perform partial consistency checking, and are oblivious of any handwritten code added to it.

Comai et al. [2002] describe a tool for statically verifying consistency properties on XML-based WebML models. Although WebML is a visual language, the models are stored in an XML-based textual representation. The tool can be used to report erroneous patterns in those XML-based models. To verify correctness of an application, syntactic and semantic checks are performed. As WebML is a graph-based language, certain consistency properties are a natural part of the syntax: for example, links to other pages in the application can be checked by checking the syntax. The semantic checks discussed in the paper are addressing issues specific to the WebML language. Like many other model-driven approaches, WebML generates only skeletal applications, and cannot perform full consistency checking once custom code is added to an application.

In previous work Bravenboer et al. [2007] described StringBorg, a generic approach to embedding a DSL in a host language, for instance by adding SQL and regular expression support to Java. The host and embedded language become linguistically integrated and therefore static verification can be performed on the newly created combination of the languages. StringBorg is a specialization of the MetaBorg approach of Bravenboer and Visser [2004] for embedding languages using SDF grammars [Visser, 1997b], which has also been used for the construction of WebDSL, as described by Visser [2007a].

2.6.2 *External Consistency Checkers*

For many frameworks, it is technically feasible to provide better consistency checks and better feedback to developers than provided by the reference implementations, as observed in Section 2.3. External third-party tools can sometimes improve consistency checking and feedback, often integrating into IDEs and providing cross-language consistency checks not performed by the reference implementation.

JetBrains develops IDEs for a number of different languages. With IntelliJ IDEA, they support the Java language, but also provide specialized support for frameworks such as the JBoss Seam framework, Struts, and GWT [JetBrains, 2009a]. The IDE provides features such as content completion and error checking in JSP (Java Server Pages) definitions and provides consistency checks and feedback not available with the reference implementation. Another JetBrains IDE is the Web IDE [JetBrains, 2009b], which provides support for a variety of languages that are commonly used together, including PHP, HTML, CSS, JavaScript, and SQL. While it provides only limited static checking for these languages, it provides an integrated environment for all these languages together, even though they are independently developed and maintained.

Tatlock et al. [2008] describe Quail, a tool for deep type checking of queries embedded in strings. The tool specializes on the Java language and performs safety checks of queries embedded in string literals, rather than introducing an embedded language. The authors showed that their tool can check most types of queries constructed as strings, but a small category of runtime-constructed (concatenated) strings remains unchecked. The embedded language approach applied for WebDSL and StringBorg does not have this limitation, but cannot be used with the embedded strings in the standard Java language as it was not designed for those checks.

External consistency checker tools can improve consistency checks of frameworks, and have one major advantage over the integrated consistency checkers discussed in Section 2.6.1 as well as those of WebDSL: they can be used with existing, industry-accepted frameworks. However, as these checkers are developed independently from the framework they analyze, they do have a number of disadvantages in terms of completeness and correctness:

- *Uncoordinated development by independent teams can lead to inconsistencies.* In addition to keeping up with the latest versions, maintaining correctness and providing complete support is increasingly difficult as more components developed by independent teams come into play.

- *Thorough framework-level consistency checking is never complete.* Whereas our approach checks a single *language*, these tools check *frameworks*. Frameworks can interact with other frameworks of external parties (e.g., a unit testing framework), new language features, and data types. The tool vendor cannot anticipate all these interactions. As a result, some of the more sophisticated consistency checks can only be implemented as heuristics.

Furthermore, these independently developed checkers pose a number of challenges to their developers, requiring significantly more effort to develop and maintain than built-in consistency checkers:

- *The language and frameworks are complex.* The complexity of the language and frameworks make their analysis very complex. Domain specific languages are typically much smaller and simpler and consequently easier to analyze.
- *The source language and framework are not designed to enable checking.* This makes it considerably harder to implement many classes of consistency checks. An example of this is the string-embedded queries of Java, checked by Quail: only by a sophisticated data-flow analysis can these queries be checked, and completeness cannot be guaranteed.
- *Supporting and keeping up with multiple versions of languages and tools requires considerable effort.* These tools must support different, independently developed versions of languages and frameworks, and combinations thereof. This places a large burden with the tooling developers, even if the goal is only to support the most recent versions.
- *Reuse of the reference compilers and interpreters is very hard.* It takes a lot of effort to effectively reuse the reference compiler and interpreter implementations (say, the Java compiler and JSF/XML processors). These tools already implement components required for consistency checks, but they have not been designed for reuse by external consistency checkers.

2.6.3 *Finding Faults by Unit Testing*

To manage the lack of static checking in web applications, unit testing is often proposed as a way to check different consistency properties in web applications. However, while unit tests are a highly effective, indispensable way of identifying regressions in an application, they do not provide the same level of accuracy, completeness, and the swiftness of static consistency checks. There are two approaches to unit testing: either strictly testing a single unit of code, making heavy use of mock objects; or writing tests that cross more than one unit of code (sometimes called cross-tests or integration tests). Strict unit tests implicate one unit of code: if a test fails, the offending code is easily identified. Writing strict, explicit unit tests for basic consistency properties is laborious and impractical. Only cross-tests, testing more than one unit, are effective at checking consistency between different modules. Still, these tests are typically not complete in testing all consistency properties. They also do not clearly implicate a particular piece of source code, like static checks or even strict unit tests can do. By applying strict test-driven development it is possible to implicate the *most recent edit* of an application as the cause of the failure of a test, but not a particular line or statement.

Static consistency checks, more so than unit tests or other runtime checks, excel at rapid and accurate error reporting. Found inconsistencies can be reported before deploying or running an application, and are always associated with a particular location in the source code. When used with an integrated development environment (IDE), any constraint violations can be reported by displaying error markers in the source code editor. This allows developers to quickly adapt their code to fix mistakes, or can guide them through the process of making larger changes, when the application may be in a state where it cannot be deployed or executed.

2.6.4 *Previous Work*

Key abstractions provided by the WebDSL language are in the areas of data modeling, user interface specification, and data operations. For a detailed overview of higher-level abstractions, built on top of these core concepts, we refer the reader to later chapters and earlier papers: Visser [2007a] and Chapter 4 give an overview of the basic design and implementation of WebDSL, Groenewegen and Visser [2008] described the access control language, Groenewegen and Visser [2009] described data validation, and Chapter 3 describes the workflow language. In contrast to these earlier papers, the present chapter focuses on consistency checking, showing how it compares to consistency checking in other languages, and describing how consistency checking is implemented.

Static consistency checking and IDE integration are a powerful combination: an IDE that supports a statically checked language can report any errors directly in the editor. In previous work, Kats et al. [2009] and [Kats and Visser, 2010a] reported on the construction of IDE plugins for the Eclipse environment using SDF [Visser, 1997b] and Stratego [Bravenboer et al., 2008], particularly focusing on the constructing of an IDE for WebDSL. In the present chapter, we focus on the semantic checks of the WebDSL language and the underlying semantic (Stratego) rules.

2.6.5 *Future work*

While the WebDSL compiler checks a lot of properties, it is not yet complete. WebDSL applications are not currently guaranteed to produce validating HTML, for instance. This is something we intend to investigate. Also, declarative rules could describe nesting restrictions of user-defined templates.

WebDSL is optimized for the construction of form-based interactive web information systems. It is currently not very well suited for building applications that mainly rely on heavy client-side JavaScript work. Improving support in this area will provide an opportunity for verification of Rich Internet Applications. We also intend to investigate how we can further simplify the definition of compilers with static verification in Stratego, e.g. by even more declaratively defining scoping rules.

2.7 CONCLUSION

In this chapter we demonstrated that timely, accurate and adequate error reporting is problematic in current state-of-practice web frameworks, such as Ruby on Rails, Lift and Seam. While certain frameworks report some application inconsistencies at compile-time, many are only discovered later, at deployment or run time. The lack of consistency checking in otherwise statically checked languages can be contributed to the linguistic separation of these frameworks. Aspects of the applications are defined in separate DSLs whose consistency is not checked with the rest of the application.

In this chapter we argued that DSLs should be designed from the ground up to enable static verification by linguistically integrating its sub-languages. Based on static verification and linguistic integration, the WebDSL language provides consistency checks that are reported at compile-time, can directly be traced back to their source, and provide clear, domain-specific error messages. We showed examples of error messages given by the WebDSL compiler. Subsequently we detailed the architecture and implementation of a consistency checker for a simplified version of the WebDSL language.

Postscript: Static Consistency Checking of Web Applications with WebDSL

While "Static Consistency Checking of Web Applications with WebDSL" [Hemel et al., 2010] was not the first paper we published, it does provide a strong motivation for the rest of the thesis, which is why it is the first core chapter. The ability to statically verify an entire application is a quality of WebDSL and mobl that should not be underestimated. In web and mobile development, the use of dynamic languages is rapidly becoming the norm, greatly complicating the ability to statically verify applications, resulting in late failure.

For a long time WebDSL's compiler and verifier could only be run from the command line. Although the compiler clearly reported file names and line numbers alongside errors, such output does not make much of an impression during demos. When Spoofax [Kats and Visser, 2010b] became stable enough, Danny Groenewegen created an Eclipse plug-in for WebDSL, integrating the type checker into the IDE. Using the plug-in, errors are highlighted *as the user types* with accurate error messages, as demonstrated in Section 2.4. This makes a big difference in user experience and error detection time, because the compiler does not have to be explicitly called to run the static verification process.

Before publishing this chapter as a paper, we published an early version of Section 2.2 online as a series of blog posts[7]. The response, especially to the criticism on Ruby on Rails, was intense. A small group of people acknowledged the problem, although they did not feel the problem was big enough to move away from Rails. However, a larger group simply rejected our claims. We must be *stupid* to make such errors and not to be able to interpret the "perfectly clear" error messages that the framework produced. At one point we had to ban a person who proceeded to call us names on all subsequent (unrelated) blog posts as well. Clearly, this was a sensitive issue. At times it seemed like criticizing a web framework was like attacking a religion.

[7]`http://zef.me/2308/when-rails-fails`

WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications

3

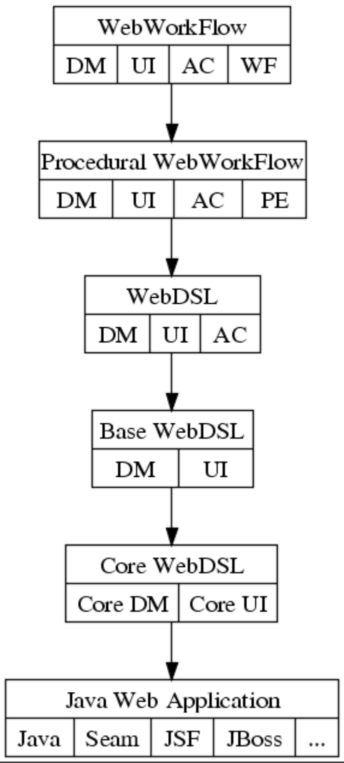
ABSTRACT

Workflow languages are designed for the high-level description of processes and are typically not suitable for the generation of complete applications. In this chapter, we present WebWorkFlow, an object-oriented workflow modeling language for the high-level description of workflows in web applications. Workflow descriptions define procedures operating on domain objects. Procedures are composed using sequential and concurrent process combinators. WebWorkFlow is an embedded language, extending WebDSL, a domain-specific language for web application development, with workflow abstractions. The extension is implemented by means of model-to-model transformations. Rather than providing an exclusive workflow language, WebWorkFlow supports interaction with the underlying WebDSL language. WebWorkFlow supports most of the basic workflow control patterns.

3.1 INTRODUCTION

Workflow is concernced with the coordination of *activities* performed by *participants* involving *artifacts* [Hollingsworth, 1995, WfMC, 1999]. Workflow and business process modeling languages such as UML activity diagrams [Dumas and ter Hofstede, 2001], BPEL [Curbera et al., 2003], and YAWL [van der Aalst and ter Hofstede, 2005], are designed for the high-level description of a wide variety of workflows or *business processes* ranging from the documentation of the operating procedures for a factory, the administrative processes involving (paper) documents of a business, or the procedures carried out by medical staff with patients in a hospital. Thus, participants in a workflow may be people, machines, or machines operated by people, and artifacts may be electronic data or physical artifacts. A worklow description may be just the documentation of a procedure to be carried out by humans, or it may be the specification of an interactive automated process. If automated, a workflow may be coordinated by a central machine (e.g. a web server), or it may consist of a network of collaborating (web) services. To cover this wide range of applications, workflow languages are restricted to modeling processes and not complete applications. That is, using a workflow engine for the execution of a process definition requires external applications or code to implement individual activities.

Web applications are concerned with presenting information to, and obtaining information from users interactively through a web browser. There are many types of web applications that contain workflow elements, i.e. the coordination of activities performed by participants. Consider for instance the following three examples. (1) An issue tracker coordinating the activities of the members of a project through registration, assignment and monitoring progress on issues. (2) A conference management system coordinating the activities of authors, program chairs, program committee members, external reviewers, meeting planners, and attendees to produce, review, select, and present a collection of scientific publications. (3) A user registration component, creating an account for a new user by subsequently registering, checking of credentials and confirming by email, involving a user and administrator. Thus, workflow concepts can be used as organizing principle for the engineering of many web applications, supporting the high-level implementation of the administration and monitoring of a process.



DM = data model, UI = user interface, AC = access control, PE = procedure events, WF = workflow

Figure 3.1 WebWorkFlow is implemented as extension of WebDSL

Rather than deriving an incomplete skeleton or boilerplate application from a process definition, a customized application with workflow requires *integration* of a workflow description language with a web engineering language.

In this chapter, we present WebWorkFlow, an object-oriented workflow modeling language for the high-level description of workflows in web applications. WebWorkFlow is an *embedded language* [Bravenboer and Visser, 2004] extending WebDSL [Visser, 2008], a domain-specific language for web application development, with workflow abstractions (Figure 3.1). From the definition of procedures operating on objects, and a control flow description to connect these procedures, complete custom web applications can be generated.

The WebWorkFlow generator is designed and built using a number of best practices for *domain-specific language engineering* [Visser, 2008]. Rather than providing an exclusive workflow language, WebWorkFlow supports interaction with the underlying WebDSL language. This approach enables the use of workflow abstractions where possible, and the use of the regular web modeling facilities where needed. This practice is called *language integration and separation of concerns [Groenewegen and Visser, 2008]*.

The target language (WebDSL) is a subset of the source language. The high-level language is more expressive (more concise models), but may not have the same coverage. For example, process expressions support only structured control-flow, while the underlying procedure event model supports unstructured control-flow. Thus, this approach makes it possible to use high-level abstractions where possible, but allows escaping to the next lower level where needed, thus increasing coverage of the language. This approach is called *compilation by normalization [Kats et al., 2008]* and is sketched in Figure 3.1. At the top level is WebWorkFlow, a rich DSL with sub-languages for data, user interface, access control and workflow modeling. WebWorkFlow is translated to lower-level procedural WebWorkFlow where workflow process descriptions have been translated to procedure events. Procedures are translated to a combination of data models, user interface elements and access control rules. This chain of transformations continues until all that is left is core WebDSL, a relatively low level model that can be easily mapped to the target platform, in this case Java/SEAM.

3.1.1 *Contributions*

The contributions of this chapter are as follows:

1. The design of the WebWorkFlow language, a language enabling the construction of complete applications involving one or more workflows.

2. A demonstration of how enabling access to multiple layers of abstraction supports coverage.

3. An evaluation of the coverage of WebWorkFlow by demonstrating how they can be used to implement a common set of *workflow patterns*.

3.1.2 *Outline*

The rest of this chapter is structured as follows. In the next section we introduce WebWorkFlow procedures by means of an example. In Section 3.3 we describe our implementation approach, explaining the procedure event model underlying the implementation of high-level process descriptions, the transformation of process expressions to procedure events, and the transformation of procedures to WebDSL, building on its high-level data model, user interface, and access control abstractions. In Section 3.4 we evaluate the coverage of WebWorkflow by examing the encoding of the control-flow patterns of van der Aalst et al. [2003]. In Section 3.5 we discuss the relation of WebWorkFlow to other process modeling approaches.

Workflows in WebWorkFlow are defined by means of *workflow procedures* that operate on *workflow objects*. In this section we introduce the high-level language constructs for defining objects and procedures, using as running example a simple workflow for organizing 'progress meetings' between managers and their employees. More precisely, rather than organizing the meeting itself, the workflow organizes the organization surrounding the meeting. Prior to the meeting, the manager and employee provide their own view on the progress of the employee. After the meeting the manager writes a `report` about the meeting. The employee may approve the report or may provide `comment`s on the report, which may cause the manager to revise the report. When the report is approved, the manager finalizes it. The complete WebWorkFlow implementation of this `ProgressMeeting` workflow is defined in Figures 3.2 and 3.4.

Workflow Objects

WebWorkFlow is an *object-oriented* workflow language. Central to the definition of a workflow is a workflow object that accumulates the data produced in the process and documents its progress. Typically, a workflow object is a domain object in the domain model of the application. For example, in a conference management system natural workflow objects are `Paper` and `Review`. If the only purpose of a workflow is to schedule a number of steps without a natural domain object, a special entity can be created to represent the instances of the workflow.

Workflow objects are instances of entities described using WebDSL data models [Visser, 2008]. Figure 3.2 describes the data model for the progress meeting workflow. A data model consists of *entity declarations* such as `User` and `ProgressMeeting`. An entity declaration has *properties*, which associate data with entity instances. A property has a *name* and a *type*, which may be either a *value type* indicated by `::` (e.g. `String`, `Text`, `Secret`) or a reference type, indicated by `->`, referring to other entities or collections of entities (e.g. `Set<User>`).

```
entity User {
  username  :: String
  password  :: Secret
  name      :: String
  manager   -> User
  employees -> Set<User>
}
entity ProgressMeeting {
  employee     -> User
  employeeView :: Text
  managerView  :: Text
  report       :: Text
  approved     :: Bool
  comment      :: Text
}
```

Figure 3.2 WebDSL data model for progress meeting workflow

Workflow Procedures

A workflow in WebWorkFlow is formalized by means of a *procedure*, which describes activities to be performed by one or more participants in a particular order. A procedure may consist of a single step, or may be a composition of procedures. A procedure may be automatic or may require a user to provide input, which may require a simple button click or filling in a complete form. Figure 3.4 defines the procedures for the progress meeting workflow. The `meeting` procedure defines the overall process of the workflow by composing the other procedures, which each define a single step. The screenshots on the right of Figure 3.4 are snapshots from a workflow conducted by 'Joe Manager' and 'Jane User'. The name in the menubar indicates the logged in user.

Figure 3.3 defines the syntax of high-level procedure definitions. Thus, a procedure definition has a *name* (`f`), exactly one typed parameter (`x:A`) indicating the workflow object to which the procedure applies, and a number of optional clauses, `who`, `when`, `view`, `do`, and `process`, which are discussed below. In the next section, the list of procedure clauses is extended to cater for the definition of *procedure events*.

```
procedure f(x : A) {
  who       { who }
  when      { when }
  view      { elem* }
  do        { stat* }
  process   { pexp }
}
```

Figure 3.3 Procedures

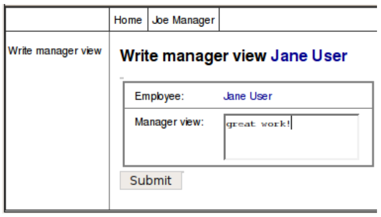
Actor The `who` clause determines which participants can apply the procedure by means of an access control predicate, based on the declarative access control model of WebDSL [Groenewegen and Visser, 2008]. The expression is a constraint on the current session and the workflow object, and any objects reachable from those via properties. The session includes a pointer to the *principal*, i.e. the logged in user associated with the session. For example, the `writeReport` procedure requires that the `principal` corresponds to `p.employee.manager`, that is, the manager of the employee for which the meeting is organized.

Activation The `when` clause provides additional constraints on the applicability of a procedure. This is used for enforcing the ordering of procedures, as we will see in the next sections. However, in the high-level language, ordering of procedures is achieved by means of process expressions. Thus, further utility of the `when` clause is to test for preconditions on the workflow object. For example, the `finalizeReport` procedure tests that the `report` has actually been written by requiring that it is not the empty string, and that the report has been approved. Another application of the `when` clause is to test timing constraints, e.g. the deadline for a `submitPaper` procedure. A procedure is only applicable when the actor and activation constraints are satisfied. Thus, the page for applying a procedure and the links to that page are not accessible if these constraints are not satisfied.

User interface The `view` clause defines the user interface for applying a procedure. This may be an arbitrary WebDSL page definition, allowing a completely customizable user interface. The page definition can display any relevant information accessible through the workflow object and session, and will typically display a form for user input required by the procedure. It is often convenient and appropriate to derive a page definition from the data model. The WebDSL `derive` construct

```
derive style from e for (p1,...,pn)
```

```
procedure meeting(p : ProgressMeeting) {
  process {
     (writeEmployeeView(p)
        |AND| writeManagerView(p));
    repeat {
      writeReport(p);
      (approveReport(p)
          |XOR| commentReport(p))
    } until finalizeReport(p)
  }
}
procedure writeEmployeeView(
    p : ProgressMeeting) {
  who  { principal = p.employee }
  view {
    derive procedurePage from p
       for (view(employee), employeeView)
  }
}
procedure writeManagerView(
    p : ProgressMeeting) {
  who  { principal = p.employee.manager }
  view {
    derive procedurePage from p
     for (view(employee), managerView)
  }
}
procedure writeReport(p : ProgressMeeting) {
  who  { principal = p.employee.manager }
  view {
    derive procedurePage from p
     for (view(employee), view(employeeView),
            view(managerView), report)
  }
}
procedure approveReport(p : ProgressMeeting){
  who  { principal = p.employee }
  do   { p.approved := true; }
}
procedure commentReport(p : ProgressMeeting){
  who  { principal = p.employee }
  view {
    derive procedurePage from p
       for (view(employee), view(report),
            commments)
  }
  do   { email(commentNotification(p));
  }
}
procedure finalizeReport(
    p : ProgressMeeting) {
  who  { principal = p.employee.manager }
  when { p.report != "" && p.approved }
}
```


Home | Joe Manager

Write manager view

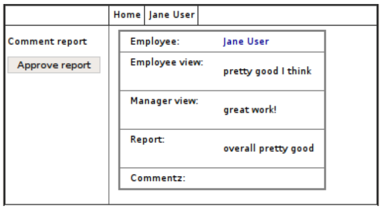
Write manager view Jane User

Employee: Jane User

Manager view: great work!

Submit

(a) `writeManagerView(p)`


Home | Jane User

Comment report

Approve report

Employee: Jane User

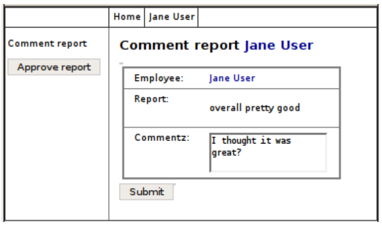
Employee view: pretty good I think

Manager view: great work!

Report: overall pretty good

Commentz:

(b) `approveReport(p) |XOR| commentReport(p)`


Home | Jane User

Comment report

Approve report

Comment report Jane User

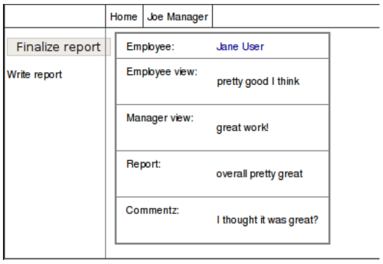
Employee: Jane User

Report: overall pretty good

Commentz: I thought it was great?

Submit

(c) `commentReport(p)`


Home | Joe Manager

Finalize report

Write report

Employee: Jane User

Employee view: pretty good I think

Manager view: great work!

Report: overall pretty great

Commentz: I thought it was great?

(d) `repeat{ writeReport(p) ... } until finalizeReport(p)`

Figure 3.4 Progress meeting workflow procedure with screenshots

provides a flexible mechanism for deriving a page from an entity declaration [Hemel et al., 2008a]. The `style` argument declares the style of the page, the expression `e` indicates the object and thus the type for which to generate the page, and the `pi` properties indicate which properties of the object should be displayed (`view`) and which should be edited. The `commentReport` procedure in Figure 3.4 demonstrates how a `procedurePage` can be gener-

```
extend entity ProgressMeeting {
  meetings → Set<ProgressMeeting>
}
procedure meeting(p : ProgressMeeting) {
  process {
    employeeMeetings(p);
    (employeeView(p) |AND| managerView(p));
    ... as before ...
  }
}
procedure employeeMeetings(p : ProgressMeeting) {
  who { principal = p.employee }
  do {
    for(u : User in p.employee.employeesList) {
      p.meetings
       .add(ProgressMeeting{employee := u})
    }
  }
  process {
    AND(q : ProgressMeeting in p.meetingsList) {
      meeting(q)
    }
  }
}
```

Figure 3.5 Recursive workflow procedure.

ated for two view properties (`employee` and `report`) and one edit property (`comments`).

Action The `do` clause describes the action taken when the procedure is applied. Actions can be described using a simple imperative language. For a standard `procedurePage` the default action is to save the changes for the edit fields in the form, and no further action is needed. Additional actions may be taken to implement business logic, to send a notification email as in the `commentReport` procedure, or to create sub-workflows as described below.

Process The `process` clause contains a *process expression* defining the composition of procedures to apply after invoking the containing procedure. For example, the `meeting` procedure in Figure 3.4 defines the composition of the individual steps in the `ProgressMeeting` workflow. Process expressions are composed from procedure invocations with several combinators. The sequential composition `e1 ; e2` of two expressions first applies `e1` and then `e2`. The parallel composition `e1 |AND| e2` applies `e1` and `e2` in parallel waiting for both to terminate. The iteration `repeat{e} until f(o)` applies `e` until procedure `f` is applied. The parallel composition `e1 |XOR| e2` enables the application of `e1` and `e2`, but cancels one if the other has terminated.

Recursive Procedures

The meeting example defines a workflow on a single workflow object `ProgressMeeting`. However, there can be multiple instances of this workflow in different stages of execution in parallel. For each user there can be a `ProgressMeeting` instance, or even several, say if an employee has more than one manager, or one for each year of employment. Thus, a workflow instance corresponds to an instance of the corresponding workflow object. Procedures can instantiate new sub-workflows by invoking procedures on linked objects (through properties). Such sub-workflows can also be *recursive* in the sense that a procedure may call itself on another workflow object. The example in Figure 3.5 illustrates recursion by extending the progress meeting example. In this workflow users do not only have managers, they can also *be* managers. Before evaluating the progress of a manager, all his or her own employees should be evaluated first. Thus, the `meeting` workflow procedure is adapted to invoke `employeeMeetings(p)`. In the `do` clause, a `ProgressMeeting` object is created and added to the set of `meetings` for each employee of the user. Then, in the `process` clause, the `meeting` workflow is initiated for each employee meeting in parallel so that all employees can start working on their `employeeView` simultaneously. The `employeeMeetings` procedure needs to be finished (all reports approved and finalized) before the managers meeting can proceed.

3.3 TRANSFORMING PROCEDURES

In this section we explain how we implement the compilation by normalization approach to realize WebWorkFlow. We describe how WebWorkFlow procedures are implemented by means of model-to-model transformations to the underlying WebDSL language. The conceptual design of WebWorkFlow may suggest that it is an object-oriented language that can be directly translated to a regular object-oriented language such as Java. However, due to the statelessness of the HTTP protocol, state has to be kept in between requests. Traditionally, sessions have been used for this purpose, however sessions typically only last a few hours whereas a workflow can last years. Furthermore sessions are bound to one particular user, whereas many users can participate in a workflow. So rather than using sessions, the workflow state is persisted in the database through extension of the application's data model. Furthermore, page definitions are used to implement the user interface for applying a procedure, and access control rules to regulate the applicability of a procedure. The transformations rely on the data, user interface and access control modeling languages of WebDSL.

Procedures to Pages

The basic idea for the implementation of a procedure is illustrated in Figure 3.6 with the transformation of `procedure` *`f`*`(`*`a`*`:`*`A`*`){...}`. To record the state of a procedure, the workflow entity *`A`* is extended with a property with the name of the procedure referring to a `ProcedureStatus` object.

The basic implementation of `ProcedureStatus` provides an `enabled` property, which indicates whether the procedure may be applied, and an `enable()` function, which can be used to set this property to true.

The user interface for the procedure is realized by means of a page definition with the name of the procedure and the workflow object as argument. The *view* from the procedure is used as specification of the presentation and the `do` action is performed on submit and disables the procedure by resetting the `enabled` flag. Finally, an access control rule uses the *who* and *when* expressions to regulate access to the procedure. The `enabled` property of the status object is used as an additional requirement for applicability of the procedure.

```
procedure f(a : A) {
  who  { who }
  when { when }
  view { elem* }
  do   { stat* }
}
```

⇓

```
extend entity A {
  f -> ProcedureStatus
}
entity ProcedureStatus {
  enabled :: Bool
  function enable() {
    enabled := true;
  }
}
define page f(a : A) {
  elem*
  action do() {
    a.f.enabled := false;
    stat*
  }
}
access control rules {
  rule page f(a : A) {
    who && when &&
    a.f.enabled
  }
}
```

Figure 3.6 Transformation of procedure

Task Lists and Navigation

In addition to the basic page for applying a procedure, further elements for the user interface of an application can be derived from procedure definitions. In particular, a definition of a task list with links to pages for applicable procedures for a particular workflow object, or a list with all available procedures for a particular user. The access control mechanism of WebDSL ensures that links in such work lists are only displayed if the pages they point to are accessible. Thus, the user interface is dynamically adapted to the state of the application. This is illustrated in Figure 3.4, where links to applicable procedures can be seen in the sidebar.

Procedure Event Model

A procedure has a life cycle that starts with the creation of the workflow object it is associated with and ends with its destruction. WebWorkFlow provides an event model for observing the changes in the life cycle of a procedure. Observation of events is realized using the following event handling clauses in a procedure definition (Figure 3.7):

```
procedure f(a : A) {
  enabled   { stat* }
  who       { who }
  when      { when }
  view      { elem* }
  do        { stat* }
  done      { stat* }
  process   { pexp }
  processed { stat* }
  disabled  { stat* }
}
```

Figure 3.7 Procedure events

- `enabled` is triggered after a call of `enable()`

- `disabled` is triggered after a call of `disable()`
- `done` is triggered after the execution of the `do` clause
- `processed` is triggered after the procedure's process has terminated; in case a procedure has no `process` clause the `processed` event follows directly after the `done` event.

To support modular subscription to the events of a procedure, the `extend procedure` mechanism can be used to add additional statements to an event handler. For example, the definition

```
extend procedure f(a : A) {
  processed { stat* }
}
```

extends the `processed` event handler with the *`stat*`* statements.

Encoding Procedure Dependencies

The procedure event model provides a general mechanism for encoding a wide variety of policies for ordering procedures. For example, the definition

```
extend procedure writeReport(p : ProgressMeeting){
  processed {
    p.approveReport.enable();
    p.commentReport.enable();
  }
}
```

enables the procedures `approveReport` and `commentReport` after the `write-Report` procedure has terminated, which corresponds to a parallel split.

Process Expressions

While the procedure event handlers provide a flexible mechanism for composing procedures, it is also a rather tedious mechanism. A large number of procedure composition patterns can be captured using concise process expressions from which the correct event handling code can be generated automatically. For example, the sequential composition of two procedures can be encoded as in the `writeReport` definition above. That is, the sequential composition `f(x); g(x)` is encoded by calling `x.g.enable()` in the `processed` clause of `f`. This direct enabling of the successor of a procedure works fine provided that procedures are only called from one call site. Since workflow procedures are intended for human consumption, it is generally not a good idea to require the same activity in many different contexts. However, this constraint is typically violated in the case of recursion, which requires an initial call and the actual recursive call.

Dealing with multiple call sites requires a more dynamic approach to sequencing of calls. In order to return control to the proper callee, it is necessary to record the 'return address'. The return address of a workflow procedure call can be represented by the identity of the caller, i.e. its `ProcedureStatus`

```
entity ProcedureStatus {
  caller      → ProcedureStatus
  returnstate :: Int
  function enable(c : ProcedureStatus, r : Int) {
    this.enabled := true;
    this.caller := c;
    this.returnstate := r;
    this.enabled();
  }
  function disable() {
    this.enabled := false;
    this.disabled();
  }
}
```

Figure 3.8 Re-definition of `ProcedureStatus` for recording return address of procedure call.

```
procedure f(a:A){ process{ g(a); h(a) } }
```

$\Downarrow$

```
entity FStatus : ProcedureStatus {
  a -> A
  function next(state : Int) {
    if(state = 0) { a.g.enable(this, 1); }
    if(state = 1) { a.h.enable(this, 2); }
    if(state = 2) { this.processed(); }
  }
}
extend procedure f(a : A) {
  enabled { this.next(0); }
  processed {
    this.caller.next(this.returnstate);
  }
}
```

Figure 3.9 Sequential composition with state machine.

object, and its state. Figure 3.8 redefines `ProcedureStatus` with an `enable` function taking the caller identity and its state as arguments.

To determine the next step to take after a procedure returns, a process expression is transformed to a finite state machine encoded by the `next` function of the *F*`Status` entity, which specializes `ProcedureStatus` for a procedure *f*. To compute the state machine, all positions in the process expressions are assigned a unique number. For each combinator there are special rules for computing the transitions. We illustrate the computation with two examples, sequential composition (Figure 3.9) and parallel split (Figure 3.10). Procedure `f` in Figure 3.9 applies the sequential composition of `g` and `h`. It is transformed

```
procedure f(a : A) {
  process {
    p(a); [1]
    ((g1(a); e1*; h1(a) [3])
      |AND| (g2(a); e2*; h2(a) [5]));
    q(a)
  }
}
```

⇓

```
extend entity A { count :: Int }
extend entity FStatus {
  extend function next(state : Int) {
    if(state = 1) {
      a.g1.enable(this, 0);
      a.g2.enable(this, 0);
    }
    if(state = 3 || state = 5) {
      if(a.count = 1) {
        a.count := 0;
        a.q.enable(this, 0);
      } else {
        a.count := a.count + 1;
      }
    }
  }
}
```

Figure 3.10 Implementation of the split and join transitions of the `|AND|` parallel combinator.

to the `next` function in the `FStatus` entity declaration and an extension of the procedure event handlers. When `f` is enabled, the transition from the start state (0) is taken, which will lead to `g` being enabled with state 1. When the `next` transition is taken from `g`, `h` is enabled with state 2. On return from `h` the process is completed and the `processed` event handler is called, which itself returns to the caller of `f`.

Figure 3.10 defines the implementation of the `e1 |AND| e2` combinator, which applies procesess `e1` and `e2` in parallel and waits for both to complete before proceeding. The transformation assumes a normalized process expression in which the expression is preceded and succeeded by a simple procedure call. This assumption is also made for the branches of the split. Expressions that do not match this pattern (e.g. `(e1 |AND| e2); (e3 |AND| e4)` can be transformed to this form by insertion of automatic identity procedures. Note that the `[i]` expressions are state labels. When the split is reached (state 1), the heads of the two branches are enabled. When the first of the branches returns, the counter `count` is incremented. When the next branch returns the counter is 1 and the continuation `q` is enabled.

3.4 ENCODING PATTERNS

A lot of research has been conducted on the assessment of workflow languages. van der Aalst et al. [2003] describe an extensive set of workflow patterns from the process perspective ranging from simple patterns such as sequential execution to complicated patterns such as loops and cancellation patterns. Recently, a revised version of these patterns was published [Russell et al., 2006]. Patterns have also been devised for the resource perspective [ter Hofstede et al., 2004] and the data perspective [Russell et al., 2005]. In this chapter, we focus on the use of control patterns for evaluating workflow languages from the control perspective.

Figure 3.11 shows the coverage of these patterns for WebWorkFlow. In this table `+` means this pattern directly supported by WebWorkFlow, `+/-` means it is possible to implement through a workaround, and `-` means the pattern is not supported by WebWorkFlow. A number of patterns that are particularly noteworthy for WebWorkFlow are patterns 10, 12-15 and 22.

Pattern	Support
1. Sequence	+
2. Parallel split	+
3. Synchronization	+
4. Exclusive Choice	+
5. Simple merge	+
6. Multi choice	+
7. Synchronizing merge	+
8. Multi-merge	+/-
9. Discriminator	+
10. Arbitrary cycles	+/-
11. Implicit termination	-
12-15. Multiple instance patterns	+
16. Deferred choice	+
17. Interleaved parallel routing	+/-
18. Milestone	+/-
19. Cancel activity	+
20. Cancel case	+/-
21. Structured Loop	+
22. Recursion	+
23. Transient Trigger	+
24. Persistent Trigger	+

Figure 3.11 Control flow pattern coverage of WebWorkFlow

Although *arbitrary cycles (10)* are not often needed, they can be implemented using the lower-level procedure events `processed` or `done`, in which an arbitrary procedure can be enabled. This is an example of why it useful to have a high-level process description language, while at the same time still having access to a lower level of abstraction where not directly supported process features can be implemented.

Multiple instances (12-15) of procedures in WebWorkFlow are supported through multiple instances of the objects they operate on. One instance of a process can run on each object that the process is defined on. By creating new object instances, any number of instances of a process can be created. Multiple instances without synchronization (pattern 12) can be implemented by simply calling `enable()` on them, which instantiates them in a non-blocking manner. Synchronizing on a number of instances (pattern 13 and 14) can be realized by instantiating them using another process expression:
`AND(a : A in o.aList){proc}`. This construct can spawn a variable number of parallel instances of a process, with the same semantics as `|AND|`. Pat-

tern 15 (Multiple instances without a priori runtime knowledge) can be implemented as follows:

```
multiproc(o).enable(); stopmultiproc(o)
```

In the `done` clause of `multiproc`, the procedure instantly re-enables itself using `enable()`, creates a new object, and starts a process with `enable()`. Finally, `stopmultiproc` disables `multiproc` through a call to `multiproc.disable()`. To synchronize all processes created, a `when` clause can be added to the `stopmultiproc` procedure, to require for all instances that their processes are finished.

For *recursion (22)* we can distinguish three cases. First, tail recursion can be implemented using `repeat` and `while`. Second, recursion on properties of the object is supported by simply calling a procedure recursively on the property (e.g. Figure 3.5). Recursive self calls on an object `a` from a process defined on `a` are not supported. We have not yet found a use-case for non tail-recursive self recursion, so it does not seem much of an issue. (But examples are most welcome!)

Data patterns

Russell et al. [2005] discuss a number of workflow data patterns. WebWorkFlow defines its processes directly on top of data entities. This gives the procedures in the process direct access to the data entity and all the data that is linked from it. Through this mechanism many of the data patterns are naturally supported. WebWorkFlow is mostly lacking in data hiding, for which it provides no explicit support. Lack of space prevents us from providing a thorough evaluation of the data patterns that are supported by WebWorkFlow.

3.5 DISCUSSION

During the design of WebWorkFlow a number of design decisions were made that distinguish it from other workflow systems. We compare our approach to other workflow approaches and discuss opportunities for future work.

Evaluation

Most workflow systems (YAWL, JBPM, BPEL) dynamically load a workflow description and *interpret* it. The advantage of this approach is the ability to adapt the workflow while the system is running. The workflow description can then simply be reloaded. However, this ability is often limited to process descriptions, which means that no new tasks or procedures can be defined at runtime. In WebWorkFlow on the other hand, workflows are *compiled* to WebDSL (which in in its turn compiles to Java/JSF). As a consequence, runtime adaptations of the workflow process are not supported. However, the compiled application is much more light-weight than an interpreted worfklow management system.

The main concern of most workflow management systems is *controlling* the process. User interfaces and the rest of the application are handled in separate systems and are thus outside the scope of the workflow system. WebWorkFlow is an embedded DSL in WebDSL and therefore integrates well with the rest of WebDSL. This integration enables the developer to more easily develop and *generate* complete user interfaces, automated tasks and control flow. At the same time, it gives a maximum of flexibility as one can always resort to a lower level of abstraction in case the ultimate abstraction layer does not support a certain construction.

Following the tendency to design languages that are understandable by both business analysts and technical developers, and the shift from workflow to business process modeling, most workflow approaches use a *graphical language* for specifying processes. WebWorkFlow is specifically aimed at web developers and uses a *textual language*. The first reason to use a textual language is because it is an embedded DSL within WebDSL, which is a textual language. The second reason is that it turns out to be a very efficient and expressive way of expressing workflows.

Related Work

The approach that is taken by Brambilla et al. [2007a], is that of adding workflow support to any domain model by performing a model transformation, by which the original model is extended with the necessary domain model elements to support workflow. WebWorkFlow applies these ideas to WebDSL. WebWorkFlow is more expressive than the approach discussed by Brambilla, because it does not cover nested sub-processes, which precludes recursion. Brambilla et al. [2006] themselves applied these ideas to WebML , where they describe how processes described in BPMN can be used to enact workflow processes in WebML applications. The authors envision that their explicit design styles in the future could be automated to generate skeleton applications based on process descriptions. WebWorkFlow realizes this vision; it does not only generate a skeleton application, but a complete application that can be *customized at the model level* to better suit application-specific needs.

YAWL is a graphical worklow language and system designed by Van der Aalst and Ter Hofstede, authors of the work on workflow patterns [Russell et al., 2006, ter Hofstede et al., 2004, Russell et al., 2005]. YAWL is designed to support almost every workflow pattern. Its formalization is based on high-level Petri nets [van der Aalst and ter Hofstede, 2005]. Although YAWL is a graphical language, a lot of configuration needs to be done by setting parameters in property boxes. The process diagram does not always shows a complete picture because so much information is hidden inside properties. While YAWL is very expressive, it does not have the layered implementation that WebWorkFlow has, which allows to use lower level constructs to implement workflows not directly supported by the workflow language.

Panta Rhei [Eder et al., 1997] is a web-based workflow system that interprets workflows specified in a textual language. Workflow data in Panta Rhei is persisted in a database, which, like in WebWorkFlow, makes it possible to alter the execution of workflows at run-time by changing this data. A web browser is used as interface to the user. Communication with other systems is possible using an internal form representation, which remote systems must be able to interpret. Panta Rhei also has support for timing and transactional features, both of which are future work for WebWorkFlow.

iTasks [Plasmeijer et al., 2007] is an internal workflow DSL based on the functional programming language Clean. iTasks shares many goals with WebWorkFlow, it enables workflows to be defined at a high level of abstraction, it supports the creation of fully working applications based on the workflow definition and avoids a lot of boiler plate code. However, the approach taken is different. iTasks takes a functional programming approach to creating and composing workflows, based on a small set of combinators. WebWorkFlow takes the object-oriented approach of implementing processes like methods on objects, resulting in a language that is more familiar to users accustomed to the object-orient paradigm.

WebWork is a web-only implementation of the workflow management system Meteor-2 [Miller et al., 1998]. It uses a graphical designer to specify a workflow, from which HTML and CGI scripts are generated. Automated tasks are performed using a socket connection with a web server and invoking CGI programs. WebWork is not as expressive as BPEL or BPMN, as only somewhat more than half of the first 20 workflow pattern are supported [van der Aalst et al., 2003]. Also, the use of CGI adds flexibility, but a custom task is not easily constructed, as opposed to WebWorkFlow, where all code can be specified in one language.

BPMN (Business Process Modeling Notation) is a business process modeling notation language designed by the Business Process Modeling Initiative [White, 2004] and now maintained by the Object Management Group [Recker and Strategy, 2006]. Its goal is to provide a notation for describing business processes understandable for both business analysts and technical developers. WebWorkFlow on the other hand is much more technical and mainly aimed at developers. In contrast to WebWorkFlow, BPMN is not directly executable. It is possible however, to derive executable workflow specifications from workflows specified using a subset of BPMN.

Future Work

An interesting area of research for WebWorkFlow is making procedures accessible through a web service interface (as opposed to the current HTML interface) and accessing procedures of remote servers. WebWorkFlow could then be used for web service orchestration, similar to BPEL. Because of the user-initiated nature of WebDSL (and WebWorkFlow by extension), using timed events is not yet possible. Conceivable applications of timing features are scheduling tasks and using deadlines to automatically influence the execution of procedures.

As WebWorkFlow is used to generate a fully functional workflow system instead of specifying a workflow that is interpreted by a WFMS, regenerating and deploying the workflow system could potentially break ongoing workflows. This is a data model evolution problem and is future work. The automatically generated navigation based on procedure definitions is useful, but can still be enhanced. Process descriptions can be more fully utilized to generate navigation. Although it is possible to disable (parts of) procedures in the current version of the language, it is not straightforward to roll back the state of an application when errors occur. A transaction system similar to Panta Rhei might be helpful.

3.6 CONCLUSION

In this chapter we introduced WebWorkFlow, an embedded DSL extending WebDSL with object-oriented workflow abstractions. Based on the definition of workflow procedures a full fledged executable application can be generated, including navigation and work lists. Following the 'code generation by model transformation' (Chapter 4) approach, WebWorkFlow achieves great flexibility and customizability by making the lower abstraction levels of the procedure event model and WebDSL web application modeling accessible next to the high-level workflow abstractions. WebWorkFlow covers most of the workflow control patterns. The patterns that are not directly expressible through the process expression language can often be implemented on the procedure event level, demonstrating the advantages of building a (workflow) language as an abstraction on a lower-level language.

Postscript: WebWorkFlow

Workflow seemed like an obvious addition to WebDSL, since little workflows such as user registration seem to appear everywhere. However, in practice WebWorkFlow has not seen a lot of use. In this postscript we discuss two issues that came up during the design of WebWorkFlow that we did not anticipate.

GENERATED USER INTERFACES

While WebWorkFlow solves the problem of concisely and flexibly defining and enforcing workflows in web applications, it does not provide a satisfying solution to building a user interface around this workflow. Workflows in WebWorkFlow center around objects, but how should that idea manifest itself in the user interface? As a solution we came up with the idea of using a toolbox metaphor. When selecting an object in a vector graphics drawing program, or a word processor, a toolbox appears that enables the user to perform operations on it. In WebWorkFlow workflows the idea is similar: on the page that represents the object, there is a list of tasks that appears along the side of the object that shows tasks that can be performed at that time. While useful from a prototyping perspective, these user interfaces are rarely acceptable to be used in practice.

Another example of this issue, independent of the workflow component of WebDSL, is the automatic derivation of view and edit (CRUD) pages for entities. While a very useful feature for rapid prototyping, the generated user interface is rarely the desired user interface. Similarly, the user interfaces elements generated by WebWorkFlow *work*, but in practice they need to be tweaked which is something that the language does not support well.

Many high-level extensions of WebDSL rely on the automatic generation of templates and pages that can be used in the application. The question is how to customize such generated templates from within the application. Minor adjustments can be made with CSS styles, but beyond the trivial, styles are not sufficient. *Automatically generating user interfaces* that are aesthetically pleasing is difficult and we have not yet found a way to support this in WebDSL.

STRICTNESS

Another issue that we found after developing WebWorkFlow is that it is, like workflow tools in general, very strict. Web applications typically allow the user to do many things simultaneously in arbitrary orders. Workflows are intended to impose a specific order in which tasks have to be completed. They work well in bureaucratic workflows, but this type of workflow is not so common in the type of web applications that we typically build.

Code Generation by Model Transformation

4

ABSTRACT

The realization of model-driven software development requires effective techniques for implementing code generators for domain-specific languages. This chapter identifies techniques for improving separation of concerns in the implementation of generators. The core technique is *code generation by model transformation*, that is, the generation of a structured representation (model) of the target program instead of plain text. This approach enables the transformation of code after generation, which in turn enables the extension of the target language with features that allow better modularity in code generation rules. The technique can also be applied to 'internal code generation' for the translation of high-level extensions of a DSL to lower-level constructs within the same DSL using model-to-model transformations.

This chapter refines our earlier description of code generation by model transformation (as described by Hemel et al. [2008a]) with an improved architecture for the composition of model-to-model normalization rules, solving the problem of combining type analysis and transformation. Instead of coarse-grained stages that alternate between normalization and type analysis, we have developed a new style of type analysis that can be integrated with normalizing transformations in a fine-grained manner. The normalization strategy has a simple extension interface and integrates non-local, context-sensitive transformation rules.

We have applied the techniques in a realistic case study of domain-specific language engineering, i.e. the code generator for WebDSL, using Stratego, a high-level transformation language that integrates model-to-model, model-to-code, and code-to-code transformations.

4.1 INTRODUCTION

Model-driven software development aims at improving productivity and maintainability of software by raising the level of abstraction from source code in a general purpose language to high-level, domain-specific models such that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details as described by Stahl and Völter [2005], Schmidt [2006], Kelly and Tolvanen [2008]. The essence of the approach is to shift the knowledge about these implementation details from the minds of programmers to the templates of the *code generators* that automatically translate models into implementations.

While model-driven software development should make developing end-user applications easier, the effort of understanding and appropriately using the implementation platforms is shifted to *domain-specific language engineering*, that is, the design and implementation of domain-specific languages (DSLs). DSLs must incorporate the implementation patterns of a complete application domain rather than those sufficient for a single application, possibly amounting to a significantly larger effort. This effort can be amortized by using a language in many projects, and/or by reducing the cost of creating and maintaining domain-specific languages and their generators. Thus, effective methods and techniques for domain-specific language engineering are crucial for realizing model-driven software development. A DSL engineering approach should not only reduce the effort of the initial creation of a DSL, but also the effort needed for maintenance tasks such as the adaptation to a new implementation platform, or the extension with new abstractions.

As a realistic case study in the application of transformation techniques to the systematic development of domain-specific languages for model-driven software development, we are developing WebDSL, a domain-specific language for modeling web applications with a rich data model. In earlier work [Visser, 2008] we described the *process* of designing WebDSL, to contribute to a method for systematic development of new DSLs.

In this chapter, we describe the *techniques* for *improving separation of concerns* in the implementation of code generators for domain-specific languages, based on our experience with the implementation of WebDSL. The core technique is *code generation by model transformation*, that is, the generation of a structured representation (model) of the target program instead of plain text. This approach enables the transformation of code after generation, which in turn enables the extension of the target language with features that allow better modularity in code generation rules. The technique can also be applied to 'internal code generation' for the translation of high-level extensions of a DSL to lower-level constructs within the same DSL using model-to-model transformations.

This chapter refines our earlier description of the *code generation by model transformation* approach [Hemel et al., 2008a] with an improved architecture for the composition of model-to-model normalization rules. In particular, in this earlier description we discussed the difficulty of combining type analysis and rewriting. Instead of coarse-grained stages that alternate between normalization and re-application of type analysis, we have developed a new style of type analysis that can be integrated with normalizing transformations in a fine-grained manner. The new normalization strategy has a simpler interface, which makes it easier to extend with rules for new language extensions. Furthermore, the strategy integrates non-local, context-sensitive transformation rules for aspect weaving to support separation of concerns *in WebDSL*.

4.1.1 *Contributions*

The contributions of this chapter are as follows:

1. An approach to code generation that enables both *modularity* and *separation of concerns* in code generators.

2. Techniques to combine transformation and type analysis in a compiler.
3. A description of how these techniques are applied in the WebDSL compiler.

4.1.2 *Outline*

In the next section we give a brief introduction to WebDSL. In Section 4.3 we discuss the architecture of the implementation of WebDSL. The core of the approach is based on *code generation by term rewriting* (Section 4.4), which employs term rewrite rules with concrete object syntax to transform DSL models to code models. The implementation of the approach is based on Stratego/XT, a language and toolset for program transformation as described in Visser [2004], Bravenboer et al. [2008]. Stratego is a high-level transformation language that integrates model-to-model, model-to-code, and code-to-code transformations. The language provides *rewrite rules* for the definition of basic transformations, and *programmable strategies* for building complex transformations that control the application of rules.

The use of *concrete object syntax* [Visser, 2002] in the definition of transformation rules improves the readability of rules, guarantees syntactic correctness of code patterns, *and* supports the subsequent transformation of generated code. In Section 4.5 we show how this can be used to *extend the target language* to make it better suited for code generation. For example, we have created an extension of Java with identifier composition, interface extraction, and partial classes and methods to simplify code generation rules.

The WebDSL language described in Section 4.2 provides basic abstractions for data models, user interfaces, and actions. While this provides a significant abstraction from the implementation platform, the language still requires repetive code and does not provide optimal separation of concerns. In Section 4.6 we discuss high-level abstractions for web applications that can be expressed by transformation to the base language. We illustrate this using examples of modules, modular data model definitions, user interface templates, high-level polytypic user-interface elements, declarative access control rules [Groenewegen and Visser, 2008], and the workflow procedures described in more detail in the previous chapter (Chapter 3), which can all be implemented using model-to-model transformations.

The extended language is implemented by means of *compilation by normalization*, a transformation process in which high-level constructs are gradually transformed to constructs of the base language. The implementation of these transformations often requires more than simple local-to-local rewrite rules. Section 4.7 discusses the implementation of model-to-model transformations using rewrite rules, and discusses techniques for realizing local-to-global and global-to-local transformations such as needed for aspect weaving.

The implementation of the range of language extensions requires the composition of the model-to-model transformations that implement them. A natural approach is to stage the transformations according to level of abstraction, that is, starting with the highest level constructs, gradually transform down to the base language level. However, it turns out that interaction between transformations makes staging of normalization rules cumbersome. In particular, the interaction between type analysis and normalization requires many re-applications of type analysis in a staged setting. In Section 4.8 we present an approach for combining model-to-model transformations into a single normalization strategy that is modularly extensible with rules for new language constructs. In Section 4.9 we present an approach for fine grained combination of type analysis and transformation, based on a refactoring of the type analysis strategy.

4.2 WEBDSL

WebDSL is a textual domain-specific language for the implementation of web applications with a rich data model. The language provides sublanguages for the specification of data models, for the definition of custom pages for viewing and editing objects in the data model, and for the manipulation of data. Manipulation of data is defined by actions which are contained in pages. This section describes each of these sublanguages using the implementation of a small blogging application as illustration (Figures 4.1, 4.2, and 4.3).

4.2.1 *Data Model*

A data model specification introduces *entity* definitions (e.g., `Blog`, `BlogEntry`, and `User` in Figure 4.1[1]), consisting of *properties* with a name and a type. Types of properties are either value types (indicated by `::`) or associations to other entities defined in the data model. Value types are basic data types such as `String`1 and `Date`5, but also domain-specific types that carry additional functionality. For example, the `WikiText`6 type implies the use of a wiki rendering engine on display of a value of the type. Associations are composite (the referer owns the object, indicated by `<>`) or referential (the object may be shared, indicated by `->`). One-to-many and many-to-many relationships between entities are defined through the use of the `Set`3 and `List`2 types. The `inverse`4 annotation on a property declares a relation with automatic synchronization of two properties.

4.2.2 *User Interface*

Page definitions consist of the name of the page, the names and types of the objects used as parameters, and a presentation of the data contained in the parameter objects. For example, the definition `blog(b : Blog)` in Figure 4.2 defines a page showing all blog entries for blog `b`. WebDSL provides basic markup operators such as `section`8 and `header`9 for defining the structure of a page. Data from the object parameters (and the objects they refer to) are injected in the page by data access elements such as `outputString`7. Collec-

[1]The code examples in this chapter contain callouts to help the reader. These underlined superscripts connect code fragments in the text to code fragments in the example.

```
entity Blog {
  title   :: String[1] (name)
  entries <> List[2]<BlogEntry>
  authors -> Set[3]<User>
}
entity User {
  username :: String
  password :: Secret
}
```

```
entity BlogEntry {
  blog    -> Blog
     (inverse[4]=Blog.entries)
  title   :: String (name)
  author  -> User
  created :: Date[5]
  content :: WikiText[6]
}
```

Figure 4.1 Blog data model

```
define page blog(b : Blog) {
  title { outputString[7](b.title) }
  section[8] {
    header[9] { outputString(b.title) }
    for[10](entry : BlogEntry in b.entries
         order by entry.created desc) {
      section {
        header { outputString(entry.title) }
        par { "by " outputString(entry.author)
              "at " outputDate(entry.created) }
        par { outputWikiText(entry.content) }
        navigate(editBlogEntry(entry))[11] { "Edit" }
      }
    }
    navigate(createBlogEntry(b)) { "New Blog" }
  }
}
```

Figure 4.2 Blog page definitions.

tions of data can be presented using the iterator construct `for`[10], which can filter and sort the elements of a collection. Navigation among pages is realized using the `navigate`[11] element, which takes a page with parameters and a link text as arguments.

4.2.3 *Data Operations*

Data can be manipulated by declaring page actions. The sublanguage used in these page actions is a Java-like imperative language with a simple API. The `createBlogEntry` page in Figure 4.3 shows a page that allows the user to create blog entries for a specific blog. Data input elements inside a `form`[13], such as `inputString`[14], are used to enter information. The `save`[16] action stores the blog entry. Execution of an action can result in page navigation, initiated by the `return`[17] statement. An `action`[15] element in a page is used to connect a button with the caption in its first argument to the action call in its second argument. This page also shows that pages can have variables to be used for initialization of data, in this case the var `be`[12] of type `BlogEntry`.

```
define page createBlogEntry(b : Blog) {
  title { "Create Blog Entry for " outputString(b.title) }
  var BlogEntry be[12] := BlogEntry {
    author := securityContext.principal
    be.blog := b
  }
  section {
    header { "Create Blog Entry for " outputString(b.title) }
    form[13] {
      table {
        row { "Title:" inputString(be.title)[14] }
        row { "Content:" inputWikiText(be.content) }
      }
      action[15]("Save", save())
    }
  }
  action save[16]() {
    be.created := now();
    be.save();
    return blog[17](b);
  }
}
```

Figure 4.3 Blog data manipulation.

4.3 IMPLEMENTING WEBDSL BY CODE GENERATION

The architecture of the WebDSL generator follows the four-level model organization presented by Bézivin [2005] as illustrated in Figure 4.4. At the M_3 level we define the SDF metametamodel, which is the grammar of the Syntax Definition Formalism SDF, which is defined in (and thus conforms to) itself [Visser, 1997b]. At the M_2 level we define the WebDSL metamodel, i.e., the grammar of WebDSL defined in SDF. At the M_1 level we define WebDSL models of web applications, consisting of entity and page definitions. At the M_0 level we define the web applications consisting of Java classes and XHTML pages, which represent the models at the M_1 level.

In the implementation of WebDSL that we have realized [Visser, 2008], the M_0 systems were initially based on the Java/Seam architecture, consisting of high-level application frameworks, such as the Java Persistence API (JPA) [DeMichiel and Keith, 2006b], JavaServer Faces (JSF) [Burns and Kitain, 2006], and the Seam web framework [Kittoli, 2008]. Currently, alternative back-ends generating plain Java servlets [Coward and Yoshida, 2003] and Python (for the Google AppEngine platform[2]), are under development. In this chapter we will only consider the original Java/Seam back-end. The other back-ends use similar techniques even if the details of the transformations are different.

[2]http://code.google.com/appengine

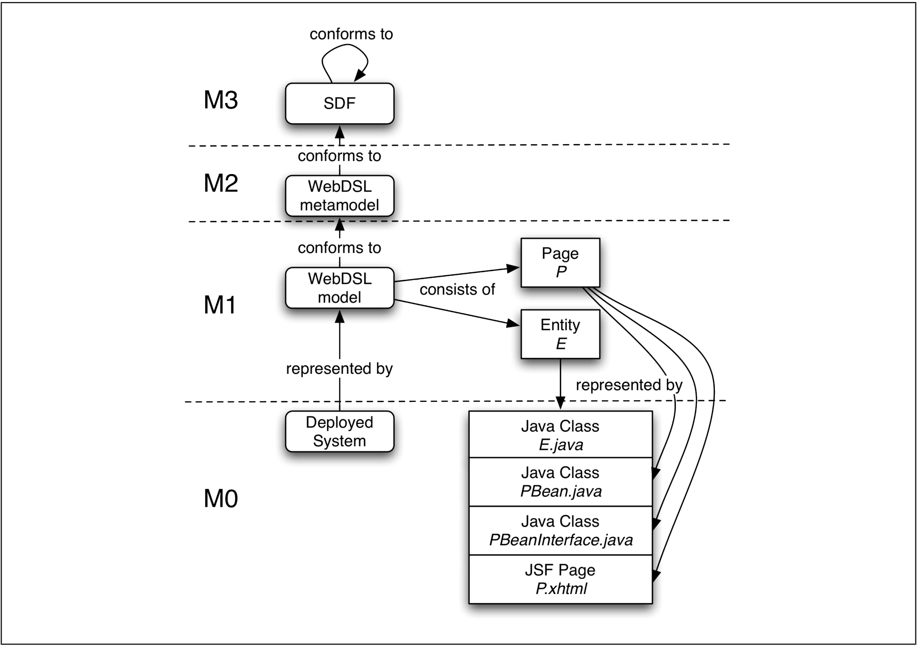


Figure 4.4 Organization of models and artifacts of the WebDSL generator.

For each entity definition, a corresponding entity class is generated with fields, accessors, and mutators for the properties of the entity, annotated for object-relational mapping (ORM) according to the JPA. Figure 4.5 shows an example transformation of a WebDSL entity `Blog`[18] to a JPA entity Java class `Blog`[21]. The `title`[19] property is transformed to a class property `_title`[22], the accessor `getTitle`[23], and the mutator `setTitle`[24]. The second property, `entries`[20], also generates annotations to define static mapping to tables (`@OneToMany`[25]) and dynamic behaviour such as save cascading (`@Cascade(SAVE_UPDATE)`[26]).

For each page definition, a JSF XHTML page, an Enterprise JavaBeans (EJB) [DeMichiel and Keith, 2006a] session bean, and the required interface are generated. Figure 4.6 shows the WebDSL page definition `editBlogEntry`[28] with its transformation to JSF and Java. The page's arguments become properties of the session bean which make them available to the JSF Expression Language (EL). In this example the WebDSL argument `e`[27] is transformed to the java property `e`[34] which can be used in JSF EL expressions, e.g. to show its title with `editBlogEntry.e.title`[31]. The connection to the correct session bean is made by referring to the name specified in the `@Name`[33] Java annotation, in this case `editBlogEntry`. The `save`[30] action defined in the page becomes the `save`[35] method of the session bean. The action can be executed using the `actionLink`[32] JSF element which is generated from the `action`[29] call in the WebDSL page. Finally, an interface is generated for the session bean which makes the methods available and specifies that the bean can be

```
entity Blog[18] {
  title[19]:: String
  entries[20]<> List<BlogEntry>
}
```

$\Downarrow$

```
@Entity public class Blog[21] {
  protected String _title[22] = "";
  public String getTitle()[23] {
    return _title;
  }
  public void setTitle(String value)[24] {
    _title = value;
  }
  @OneToMany[25] @Cascade(SAVE_UPDATE)[26]
  protected List<BlogEntry> _entries;
  public List<BlogEntry> getEntries() {
    return _entries;
  }
  public setEntries(List<BlogEntry> b) {
    _entries = b;
  }
}
```

Figure 4.5 Transformation of a WebDSL entity definition to a JPA class.

accessed locally[36].

WebDSL is a textual, domain-specific language and its M_2 metamodel is a grammar describing the valid sentences of that language. From the grammar, we automatically generate a parser that transforms the textual representation of a model to an abstract syntax tree (AST). The AST conforms to a regular tree grammar, another M_2 metamodel that defines a set of valid trees, and which is obtained automatically from the grammar. All subsequent transformations are applied to the AST corresponding to the textual representation of the model. The WebDSL generator transforms high-level models into low-level Java code and XML files. These target languages are also described by a grammar and a derived abstract syntax definition. All transformations are expressed in Stratego/XT [Bravenboer et al., 2008], which can apply transformations to any models with an abstract syntax definition.

In the following sections we discuss the organization of the generator as a pipeline of model-to-model transformations, and the idioms used to realize these transformations.

4.4 CODE GENERATION BY TERM REWRITING

Most model-driven engineering approaches accomplish code generation by simply writing strings to text files. Sometimes template engines are used to

```
define page editBlogEntry(e[27] : BlogEntry)[28] {
  form {
    table {
      row { "Title:" inputText(e.title) }
      row { "Content:" inputWikiText(e.content) }
      action("Save", save())[29]
    }
  }
  action save()[30] { e.save(); }
}
```

$\Downarrow$

```
<html><body>
<h:form><table>
 <tr><td><h:outputText value="Title: "/></td>
   <td><h:inputText
   value="#{editBlogEntry.e.title}"[31]/></td></tr>
   <tr><td><h:outputText value="Content: "/></td><td>
   <h:inputTextarea value="#{editBlogEntry.e.content}"/>
 </td></tr></table>
 <h:actionLink[32] action="#{editBlogEntry.save()}"/>
</h:form>
</body></html>
```

```
@Stateful @Name("editBlogEntry")[33]
public class EditBlogEntryBean
     implements EditBlogEntryBeanI {
  @In @Out private BlogEntry e;[34]
  public void setE(e) { this.e = e; }
  public BlogEntry getE() { return this.e; }
  public String save()[35] { entityManager.persist(e); }
}
```

```
@Local[36]
public interface EditBlogEntryBeanI {
  public void setE(e);
  public BlogEntry getE();
  public String save();
}
```

Figure 4.6 Transformation of a WebDSL page to a JSF page with a backing bean.

make this process easier. In contrast, in the *code generation by model transformation* approach, code generation is just another model transformation step. Rather than printing strings to a file, source and target models are represented by means of first-order terms, and code generation is expressed by means of term rewriting. For the implementation of WebDSL we use Stratego [Visser, 2004, Bravenboer et al., 2008], a transformation language with a combination of features that makes it suitable for code generation by model transformation. This section explains code generation by term rewriting and compares it with code generation using template engines as used in other MDE approaches.

4.4.1 *Representing Models and Code with Terms*

String template engines such as Velocity [The Apache Foundation, 2007], StringTemplate [Parr, 2004], and xPand [Voelter and Groher, 2007] use templates to generate fragments of plain text, which cannot be checked statically for syntactic correctness. Only after a complete output file has been generated for a particular input is it possible to determine if the generated code is syntactically correct.

Stratego rewrite rules operate on a structured representation, using first-order terms to represent models as a tree structure. Any patterns and fragments using this representation can be statically checked for syntactic correctness. For example, consider the following WebDSL entity declaration and its term representation:

```
entity Blog {
  ident :: String
}
```

$\Rightarrow$

```
Entity("Blog", [
  Property("ident", SimpleSort("String"))
])
```

This term corresponds to a tree structure, with as root an `Entity` node, with as children the term representations of the name and properties of the entity declaration. Using the WebDSL meta-model, the structure of this tree can be statically checked for well-formedness. Similarly, Java code can be represented using a term representation. For example, consider the following (tiny) Java class and its term representation:

```
@Entity
public class Blog {
  public Blog()
}
```

$\Downarrow$

```
ClassDec(
  ClassDecHead(
    [MarkerAnno(TypeName(Id("Entity"))), Public()]
    , Id("Blog")
    , None(), None(), None()),
  ClassBody(
    [ConstrDec(
       ConstrDecHead([Public()],None(),Id("Blog"),[],None()),
       ConstrBody(None(), []))]))
```

The essence of code generation by model transformation is to represent both the input model and the generated code as structured terms. This re-

```
entity-to-class :
  Entity(x, prop*) ->
  ClassDec(
    ClassDecHead(
      [MarkerAnno(TypeName(Id("Entity"))), Public()], Id(x),
       None(), None(), None()),
    ClassBody(
      [ConstrDec(ConstrDecHead([Public()],None(),Id(x),[],
        None()),ConstrBody(None(),[]))]))
```

Figure 4.7 Rewrite rule generating Java code with abstract syntax.

quires meta-models for the source language (WebDSL) as well as for the target languages (Java and XML). Using structured representations for input and output makes it possible to divide a large transformation into several, smaller transformation steps. We elaborate on this technique in Section 4.5.

4.4.2 *Rewrite Rules*

The elementary operations of a transformation are rewrite rules of the form `L : p1 -> p2 where s`. The name `L` of a rule can be used to invoke it in a strategy. When applied, the left-hand side pattern `p1` is matched against the subject term, binding any variables in the pattern to corresponding sub-terms of the subject term. When the match of `p1` and the condition `s` succeed, the subject term is replaced with the instantiation of the right-hand side pattern `p2`. Rewrite rules are used for code generation by translating a fragment of the source language on the left-hand side (represented as a term pattern) to a fragment of the target language on the right-hand side (represented as a term pattern). For example, the `entity-to-class` rewrite rule in Figure 4.7, rewrites a WebDSL entity declaration to a Java class, using terms to specify both the left-hand side and the right-hand side of the rule. The rewrite rule rewrites the input entity to a class where the name *x* of the entity is used as the name of the class and its constructor. For brevity, this rule does not consider the inclusion of generated properties.

4.4.3 *Concrete Object Syntax*

While we will argue that it is useful to have a structured representation of generated code, the right-hand side pattern of the rule in Figure 4.7 is not very easy to produce or understand, due to the complexity of the abstract syntax of Java. A language's concrete syntax is usually more concise and more familiar than its abstract syntax. Therefore, Stratego supports *concrete object syntax patterns* [Visser, 2002] in the definition of transformation rules. That is, textual patterns in the syntax of the language concerned that are compiled to the corresponding structured representation pattern. This provides the same level of readability as template engines, while guaranteeing syntactic correctness of code patterns. For example, using concrete object syntax, the

```
entity-to-class :
  |[ entity x_ent { prop* } ]| ->
  |[ @Entity class x_ent {
       public x_ent() { }
       cbds*
       ...
     } ]|
  where cbds* := <map(property-to-java)> prop*
      ...
property-to-java :
  |[ x :: srt ]| ->
  |[ private t x_field = e;
     public t x_get() { return x_field; }
     public void x_set(t value) { x_field = value; } ]|
  where t := <java-type> srt
      ; e := <initialization-expression> srt
      ; x_field := <concat-strings> ["_", x]
      ; X := <capitalize> x
      ; x_get := <concat-strings> ["get", X]
      ; x_set := <concat-strings> ["set", X]
```

Figure 4.8 Rewrite rules for generating Java entity class from WebDSL entity declaration.

`entity-to-class` rule in Figure 4.7 can be written as follows:

```
entity-to-class :
  |[ entity x { prop* } ]| ->
  |[ @Entity public class x { public x() { } } ]|
```

Note that *x* and *prop** are recognized as *meta-variables* for identifiers and lists of properties, respectively. In this chapter, we will indicate meta-variables using italics, to distinguish them from identifiers in the subject language. The implementation details are explained elsewhere [Visser, 2002] and are not in the scope of this chapter.

A more realistic translation of entity declarations is defined in Figure 4.8. Rule `entity-to-class` generates a Java class for an entity declaration. The rule uses the auxiliary rule `property-to-java` to generate the class body declarations (*cbds2**) defining the field and accessors for a property of an entity. Only the case of properties with value type is shown here.

4.4.4 *Rewriting versus Template Engines*

The table in Figure 4.9 summarizes the differences between the use of rewriting with concrete syntax and template engines such as Velocity [The Apache Foundation, 2007] and xPand [Voelter and Groher, 2007] for code generation. The approaches have in common that the concrete syntax of the target language is used to define (parameterized) code fragments, which makes it easier to define such fragments than with the use of abstract syntax. The approaches

Rewriting		**Template Engine**	
+	use syntax of target language	+	use syntax of target language
+	pattern matching	+	object accessors
-	requires grammar	+	flexible: no grammar needed
+	static syntactic checks	-	no static syntactic checks
+	structured representation	-	generation of text
+	automatic pretty-printing	+/-	use existing pretty-printers
+	transformation after generation	-	no transformation after generation

Figure 4.9 Comparison of rewriting with concrete syntax and textual template engines.

use different methods to instantiate the holes in a code fragment. Rewriting uses pattern matching and meta-variables, whereas template engines typically use object accessors inside anti-quotations. This is mainly a difference in programming style, which is not easy to compare. In order to parse code patterns in concrete object syntax, a grammar of the target language is necessary. Since template engines just produce text, they can be applied flexibly for different languages. On the other hand, the lack of a grammar entails no syntactic checks of the code patterns. Finally, the rewriting approach produces a structured representation, while template engines produce flat text. A structured representation means that the target code can be pretty-printed automatically. However, pretty-printers are available for mainstream languages, so this is not necessarily a concern for the use of template engines. The main difference that we are interested in for the purpose of this chapter is the fact that the rewriting approach produces a structured representation of the target code, which entails that further transformations can be applied, as we will see in the following sections.

4.4.5 *Composing Generation Rules*

In Stratego, rewrite rules can be composed using *programmable strategies* that control the application of individual rules [Visser et al., 1998, Bravenboer et al., 2008]. Stratego provides a few basic combinators for composing transformations from rules. For example, the combinator `s1 ; s2` produces the *sequential composition* of the transformation strategies `s1` and `s2`, and the combinator `s1 <+ s2` produces the *deterministic choice* of `s1` and `s2`. More complex strategies can be constructed from these basic combinators. For example, the strategy definitions

```
try(s)    = s <+ id
repeat(s) = try(s; repeat(s))
```

are part of the Stratego Standard Library. The strategy `try(s)`, tries to apply transformation `s`, but succeeds by producing the original term when `s` fails. The strategy `repeat(s)`, repeatedly applies a transformation `s` until it fails. Traversal strategies are defined using generic traversal operators, and can specify a visit sequence that works on any tree rather than for a particular tree grammar. For example,

```
topdown(s) = s; all(topdown(s))
```

defines a strategy that applies a transformation `s` to all nodes in a tree during a top-down traversal. An application-specific library can collect custom strategy definitions.

Figure 4.10 shows a basic definition of the `webdsl-to-seam` transformation strategy that transforms a WebDSL application consisting of a list of WebDSL entity and page definitions to Java classes and JSF code. The strategy is a pipeline of transformations that are applied in sequential order to the input model. The `typecheck` transformation checks the consistency of declarations and their uses and annotates identifiers with (a reference to) their declaration or type. The `generate-code` transformation uses `GenerateCode` to map individual WebDSL definitions to XML- or Java class models. The `GenerateCode` strategy is defined using multiple definitions, which is a convenient way to define an extensible composition of alternatives. Thus, the definition of `GenerateCode` in Figure 4.10 is equivalent to

```
GenerateCode = entity-to-class + page-to-class
```

that is, the non-deterministic composition of the bodies of the definitions. The non-deterministic composition `s1 + s2` of two strategies entails that either the composition `s1 <+ s2` or the composition `s2 <+ s1` is used. In other words, the order in which the alternaves are tried is undefined.

Finally, the `write-to-file` strategy pretty-prints the Java class and XML models, and writes them to a file.

4.5 TRANSFORMING GENERATED CODE

Most programming languages are not designed as targets for code generation, which is manifest especially in the lack of composition operators for program fragments of all kinds. For example, identifiers are typically just strings of alphanumeric characters without operators for composing identifiers from smaller strings. Similarly, classes and methods (in Java at least)

```
strategies
  webdsl-to-seam = typecheck; generate-code; write-to-file
  generate-code  = map(GenerateCode)
  GenerateCode   = entity-to-class
  GenerateCode   = page-to-class
```

Figure 4.10 A basic code generation pipeline.

cannot be assembled from smaller class and method fragments. This lack of compositionality in target languages is reflected in a corresponding lack of compositionality in code generation rules, as is illustrated by the rules in Figure 4.8; a rule generating a Java class must produce all ingredients of that class. Since rewrite rules produce a structured representation of the target program, it is possible to apply further transformations to generated code. In this section, we show three example extensions of the target language Java with composition operators that enable better modularization of code generation rules.

4.5.1 *Identifier Composition*

Code generation often requires the creation of many new identifiers based on identifiers in the source model. Since identifiers in Java (and most programming languages) are simple alphanumeric strings, creation of identifiers requires string manipulation and concatenation. For example, the generation of a field, and a setter and getter method in the `property-to-java` rule in Figure 4.8 requires three string concatenations and one string manipulation. Even while this is straightforward code, it takes up quite a bit of real estate in the generation rules, and it is repetitive boilerplate code. To avoid this problem we extended the Java language with the # operator, which composes its two operand identifiers into a single identifier following Java's naming conventions. Thus, `get#name` becomes `getName` and `_#name` becomes `_name`. The new version of the `property-to-java` rule from Figure 4.8 using this feature is presented in Figure 4.11. The # operator is implemented by a transformation that replaces composite identifiers by regular Java identifiers. The Java extension and the transformation can be reused in all code generators that produce Java code.

4.5.2 *Partial Classes and Methods*

Since its conception, the WebDSL generator has grown considerably. Initially, the generator was constructed in a centralized fashion, with a single *"God rule"* associated with each generated artifact. Much like a "God class", an anti-pattern in object-oriented programming, such a God rule dispatches a large number of smaller transformation rules to generate a monolithic target

```
property-to-java :
  |[ x :: srt ]| ->
  |[ private t _#x = e;
     public t get#x() { return _#x; }
     public void set#x(t value) { _#x = value; }
  ]|
  where t := <java-type> srt
      ; e := <initialization-expression> srt
```

Figure 4.11 Code generation rule using identifier composition operator `#`.

artifact (e.g., a Java class). The `entity-to-class` rule in Figure 4.8 is a typical starting point for growing such a God rule. As new language extensions are added, these rules grow to a size that no longer fits on a single screen. Thus, this pattern is a code smell that hinders the extensibility and maintainability of the generator.

The employment of God rules is the result of the structure of the target meta-model: Java does not support composition of classes. Other platforms, such as C#, provide partial classes (but not partial methods), which allow subdividing classes into smaller units. The lack of such a construct makes it difficult to decompose rewrite rules that generate large classes. This platform limitation can be resolved by extension of the target language with partial classes and methods. Our extension uses Java's annotation syntax to identify partial classes and methods with the annotation `@Partial`. For example, Figure 4.12 shows a refactoring of the generation rules of Figure 4.8, in which the `entity-to-java` and `property-to-java` rules independently generate partial classes. Methods can be declared as partial using the same annotation:

```
@Partial void initialize() { stm* }
```

Since code patterns are no longer assembled by a God rule, partial code fragments are emitted to a code repository using rules such as `emit-java-class`. The code fragments are collected and assembled by the `merge-partial-classes` strategy. Partial classes with the same name (within the same Java package) are merged into a single Java class. Similarly, the bodies of partial methods with the same name (in the same class) are merged into a single method definition. The ordering of statements originating from different partial methods is non-deterministic. Hence, the generator should make no assumptions on such an ordering. However, there is a simple technique to

```
entity-to-java :
  |[ entity x_ent { prop* } ]| ->
  <emit-java-class> |[
    @Entity @Partial class x_ent {
      public x_ent() { }
    } ]|
property-to-java :
  |[ x :: srt ]| ->
  <emit-java-class> |[
    @Partial class x_ent {
       private t _#x = e;
       public t get#x() { return _#x; }
       public void set#x(t value) { _#x = value; }
     } ]|
  where x_ent := <entity>
      ; t := <java-type> srt
      ; e := <initialization-expression> srt
```

Figure 4.12 Generation rules emitting partial classes and methods.

enforce some order. If there are several classes of statements such that all statements in one class should be executed before all statements in another class, the partial method can be refactored into a regular method that calls for each stage a partial method to which the statements of the various classes can be added.

4.5.3 *Interface Extraction*

Seam and EJB require that each session bean implement an interface containing all public methods of that bean. Generating code for these types of interfaces is tedious, since it requires generation rules that shadow the generation rules for the regular class. Therefore, we extended Java to generate these interfaces automatically. The `@RequiresInterface` annotation for a class indicates that a separate interface should be derived from the class including signatures of all its public methods. Again, the extension is implemented by means of a (library) transformation that carries out the derivation of the interface. Thus, for a page bean generated with the rule in Figure 4.13, a corresponding interface `x#PageBeanInterface` is generated as well.

```
page-to-java-bean :
  |[ define page x(param*) {elem*} ]| ->
  <emit-java-class> |[
     @RequiresInterface @Partial @Stateful
     class x#PageBean {
       @In EntityManager em;
       ...
     }
  ]|
```

Figure 4.13 Annotating a generated class for interface extraction.

4.5.4 *A Revised Pipeline*

Figure 4.14 shows a revision of the generator pipeline presented in Figure 4.10, incorporating the new code generation technique. Since the `GenerateCode` rules apply not only to top-level definitions, the `map` strategy is no longer sufficient. The new `generate-code` strategy uses the generic `topdown` strategy to traverse the WebDSL model, applying the `GenerateCode` rules to each node of the input model. The successful applications of `GenerateCode` contribute to a repository of code fragments, which are assembled by `merge-partial-classes`.

4.6 MODEL-TO-MODEL TRANSFORMATIONS

The WebDSL language as described in the preceding sections provides basic abstractions for web applications. As the language evolved, we have added new features to achieve higher levels of abstractions, providing better support

```
strategies
  webdsl-to-seam =
    typecheck; generate-code; merge-partial-classes;
    write-to-file
  generate-code = topdown(try(GenerateCode))
  GenerateCode  = entity-to-java
  GenerateCode  = property-to-java
  GenerateCode  = page-to-java
  GenerateCode  = parameter-to-java
```

Figure 4.14 Code generation pipeline with partial class generation and assembly.

for particular kinds of web applications and application domains. For some of these abstractions, special support from the platform is needed. However, many can be implemented by incremental model transformation steps, transforming rich WebDSL models to more primitive WebDSL models that only support the feature set of the WebDSL *core* language as described in Section 4.2. The advantage of transforming higher level abstractions to low level core models rather than generating implementation code from them directly is that the domain-specific core language that has to be mapped to implementation target code remains small, increasing portability of the generator. The development of the Servlet and Python back-ends for WebDSL took little time because the WebDSL core language is relatively small.

This section discusses several examples of abstractions we have added to WebDSL and implemented by means of model-to-model transformations. In the next sections we discuss the issues that arise in the implementation of such transformations and the techniques developed to address them. The rationale for the introduction of a core language for WebDSL is discussed more extensively in Visser [2008].

4.6.1 *Modules*

In the WebDSL core language all definitions of an application have to be defined in a single file. This is not desirable, as it impairs maintainability and reusability of applications. A simple module system defined as an extension of the core language addresses this problem. A module is a collection of definitions and may import other modules. WebDSL modules are implemented by means of a model-to-model transformation that replaces module imports by the content of the imported module. This mechanism is similar to the `#include` pragma of the C language. The crucial difference with the C mechanism is that a WebDSL module is parsed *before* its abstract syntax tree is included in the abstract syntax tree of the importing module. Thus, it is not possible to rely on textual composition to compose WebDSL definitions, e.g. using an import in the middle of a definition.

```
module usermanagement
  entity User {
    username :: String
    password :: Secret
  }
module blogentries
  imports usermanagement
  extend entity User {
    entries -> Set<BlogEntry>
  }
```

⇒

```
entity User
  username :: String
  password :: Secret
  entries -> Set<BlogEntry>
```

Figure 4.15 Joining modular data model definitions

```
define main() {
  top()
  body()
}
define top() {
  menubar  ...
}
define page blog(b : Blog) {
  main()
  define body() {
    section {
      header {
        output(b.name)
      }
      ...
    }
  }
}
```

⇒

```
define page blog(b : Blog) {
  menubar { ... }
  section {
    header {output(b.name)}
    ...
  }
}
```

Figure 4.16 Inlining template calls.

4.6.2 *Modular Data Models*

To support separation of concerns it can be beneficial to spread entity declarations over different modules that deal with various aspects of an application. For example, for our blogging application we might want to reuse a separate, generic user management module with a standard `User` data model. For the purpose of the blogging application the `User` entity needs to have a set of blog entries. Rather than directly implementing this aspect in the `usermanagement` module, we define it in a separate module, using the `extend entity` construct to extend the existing `User` entity. The transformation merges the entity definition and its extensions, mapping it to a regular entity declaration in the core WebDSL language, as illustrated in Figure 4.15.

4.6.3 *Template Definitions*

User-defined *templates* allow the developer to define reusable chunks of user interface model. For example, the `main()` template in Figure 4.16 defines a general set-up for pages (with menubar and page body) that is shared among many pages of the application. Pages can call the `main()` template and locally override parts of it. For instance, the `blog` page overrides the default `body` template. Templates can be implemented by recursively inlining the contents of their definitions into the page they are called from. Thus, in the `blog` page, the `main` template call is replaced by a `top` and `body` template call, which are subsequently replaced by a menubar and a section.

4.6.4 *Deriving User Interface Elements from Types*

The user interface elements to be used for input and output of entity properties depend on their type. For example, input of a `String` requires a simple text input field, input of a `WikiText` requires a larger text area, while input of a `Date` requires a date picker. Thus, the core WebDSL language provides different `input` elements for different types. For example, for input of the properties of a `BlogEntry e`, we would use `inputString(e.title)`, `inputDate(e.created)`, and `inputWikiText(e.content)`. Since the input element to be used depends on the type of the property, we can simplify the specification of inputs to just `input(e)`, and derive the particular kind from the type of the expression using a model-to-model transformation. Thus, `input(e.title)` is transformed to `inputString(e.title)` and the call `input(e.created)` to `inputDate(e.created)`.

A recurring pattern in user interfaces are tabular forms for input of the properties of an entity. Such forms can be derived from an entity declaration by considering its properties. WebDSL provides a collection of `derive` statements for deriving different page configurations from data model declarations. For example, the `derive editPage` statement produces a complete edit page for a particular entity, as illustrated in Figure 4.17.

4.6.5 *Access Control*

Most web applications need to adhere to a certain access control policy that defines the permissions of the various users. Such a policy can be encoded in the application using conditional checks defined in the protected resource, such as a page. If access is not permitted the user is notified by a redirect to an error page. This solution requires that access control checks are entangled with template definitions, which makes the policy encoded in the application hard to verify or modify. The WebDSL access control sublanguage [Groenewegen and Visser, 2008] supports definition of access control policies as a separate concern. Access control rules are defined in a separate section (or module) in the application, and are woven into the corresponding WebDSL definitions during compilation.

```
define page editBlogEntry(e : BlogEntry) {
  derive editPage from e
}
```

⇓

```
define page editBlogEntry(e : BlogEntry) {
  section {
    header{"Edit " output(e.title)}
    form {
      table {
        row { "Title:"   input(e.title)   }
        row { "Created:" input(e.created) }
        row { "Content:" input(e.content) } }
      action("Save", save())
      action save() {
        e.save();
        return blogEntry(e); } } } }
```

Figure 4.17 Derivation of edit page based on data model.

The example in Figure 4.18 illustrates the weaving of access control rules. The page `editBlogEntry` is protected by a rule that matches the signature of the page (i.e. the name and type of arguments) and specifies the condition for allowing access. The condition verifies that the currently logged in user, the '`principal`', is the author of the blog entry. The condition is woven into the page definition by a transformation, creating an `init` action which performs the check and redirects if necessary. Such an action is executed before each request to the page it is defined in.

```
define page editBlogEntry(e : BlogEntry) { section  ...  }
```

```
rule page editBlogEntry(be : BlogEntry) {
  principal == be.author
}
```

⇓

```
define page editBlogEntry(e : BlogEntry) {
  init {
    if(!(principal == e.author)) {
      redirect accessDenied(); }
  }
  section  ...
}
```

Figure 4.18 Access control rule transformation.

4.6.6 *Workflow*

Another recurring aspect of web applications is workflow. Workflow is concerned with the coordination of activities performed by participants involving artifacts. Workflows can be encoded using low-level constructs in WebDSL applications, however these encodings give little insight into the structure of such workflows. Therefore, we have developed WebWorkFlow (see Chapter 3), an extension of WebDSL supporting the definition of tasks on entities and the order in which these have to be performed using process expressions. Although the specifics of this extension are beyond the scope of this chapter, it has also been implemented as a model-to-model transformation. Workflow process descriptions are translated to procedures, which are then translated to a combination of entity extensions, access control rules, actions, and page and template definitions.

4.7 IMPLEMENTING MODEL-TO-MODEL TRANSFORMS

In the previous section, we have illustrated how high-level abstractions can be implemented by means of model-to-model transformations to the core language. These transformations can be implemented by means of the same techniques we employed for code generation in Section 4.3, i.e. rewrite rules with concrete syntax and transformation of generated 'code'. Using concrete syntax, it is feasible to handle large code templates, while its underlying structured representation enables cascading transformations after generation. Thus, Stratego unifies model-to-code and model-to-model transformation, avoiding the need for different languages for different types of transformations.

van Wijngaarden and Visser [2003] give a classification of mechanisms for program transformation, distinguishing the *scope* of a transformation (which part of a program it affects or is affected by), its *direction* (whether it is triggered by the source or the target), and the number of *stages* it requires. In the WebDSL generator three classes of transformations are used that differ in *scope*.

- *local-to-local transformations*, which locally transform one term to another
- *global-to-local transformations*, which retrieve information from the surrounding (global) context to perform a local transformation
- *local-to-global transformations*, which retrieve information from a local term for use in the surrounding context

In this section, we discuss the definition of transformations in these classes. In the next section, we consider the composition of such transformations.

4.7.1 *Local-to-local*

Syntactic abstractions, also known as syntactic sugar, provide new language constructs that support expression of functionality that is already provided by the base language in a more compact manner. The implementation of such abstractions can often be realized by means of simple local-to-local transformations.

A local-to-local rewrite replaces a model fragment with another fragment without using or producing other parts of the model, as illustrated by the examples in Figure 4.19. The `normalize-text` rule normalizes applications of the `text` construct with multiple arguments to a list of applications of `text` with a single argument. More precisely, it splits off the first argument of a multi-argument application. Repeated application of the rule ensures that only singleton applications remain. For example, `text(blog.title, ": ", blog.author)` is reduced to `text(blog.title) text(": ") text(blog.author)`. Similarly, the `normalize-for` rule rewrites an occurrence of the `for` statement without a `where` clause to one with the universally valid `where true` clause. These normalizations ensure that later stages of the code generator only need to deal with one syntactic variant, i.e., singleton applications of `text` and `for` statements with a `where` clause.

4.7.2 *Global-to-local*

In a global-to-local transformation, model elements are locally transformed using (global) context information. For example, a transformation may depend on the types of identifiers declared elsewhere in the model. Similarly, template calls can be implemented by inlining a template definition defined in the global context.

As an example of a global-to-local transformation consider the derivation of edit pages and input elements in Section 4.6.4, which is implemented by the rewrite rules in Figure 4.20. The `derive-edit-page` rule transforms a `derive editPage from e` element to a complete edit page, as illustrated in Figure 4.17. The transformation is driven by the type of the expression *e*. The `type-of` strategy computes the type of an expression based on declarations of types and identifiers. Given a type expression, the `entity-properties` strategy retrieves the list of properties of an entity type. For each property a row is generated by the `derive-edit-row` rule. The return address of the

```
normalize-text :
  |[ text(e1,e2,e*) ]| -> |[ text(e1) text(e2,e*) ]|

normalize-for :
  |[ for(x : srt in e1 order by e2) {elem*} ]| ->
  |[ for(x : srt in e1 where true order by e2) { elem* } ]|
```

Figure 4.19 Local-to-local syntactic normalization rules.

```
derive-edit-page :
  |[ derive editPage from e ]| ->
  |[ section{
       header{"Edit " srt " " text(e.name)}
       form {
         table { row* }
         action("Save", save()) }
       action save() { x.save(); return x_view(x); } }
  ]|
  where prop*  := <type-of; entity-properties> e
      ; row*   := <map(derive-edit-row(|x))> prop*
      ; x_view := <type-of; view-page> e

derive-edit-row(|x) :
  |[ y relation srt ]| -> |[ row{x_text ": " input(x.y)} ]|

derive-input :
  |[ input(e) ]| -> |[ inputString(e) ]|
  where SimpleSort("String") := <type-of> e

derive-input :
  |[ input(e) ]| -> |[ select(s : t, e) ]|
  where t := <type-of> e; <defined-entity> t
```

Figure 4.20 Rewrite rules to derive edit page elements

`save()` action is the view page declared for the entity type, which is obtained with the `view-page` strategy.

The edit rows generated by this rule make use of a generic `input` element, which is specialized with regard to its type by an appropriate `derive-input` rule. For example, an `input` for type `String`, is specialized to an `inputString` template, and an `input` for a defined entity type is specialized to a drop-down menu (`select`) that allows selecting an object of that type.

4.7.3 *Local-to-global*

A local-to-global transformation can locally rewrite an element of the model, while producing elements or information for use elsewhere in the global context. This pattern can be used to implement the `extend entity` construct of Figure 4.15, or similar aspect-oriented programming (AOP) constructs.

Normal rewrite rules in Stratego are *context-free*, that is, a rule locally transforms a term to another term without access to the context in which the term occurs. To express *context-sensitive* transformations, Stratego provides *dynamic rewrite rules* [Bravenboer et al., 2006b]. Dynamic rules are defined at run-time, and inherit information in the context of their definition. The global-to-local strategy `derive-entity` makes use of strategies `type-of` and `entity-properties`, which are implemented using dynamic rules to propagate context information from declarations to uses, as we will see in Section 4.9.3.

```
extend-entity :
  |[ extend entity x { prop1* } ]| -> RemoveMe()
  where rules (
          DynamicExtendEntity :+
             |[ entity x { prop2* } ]| ->
             |[ entity x { prop2* prop1* } ]|
        )
```

Figure 4.21 Merging extended entities using a local-to-global transformation

Dynamic rules can also be used to implement local-to-global transformations, as illustrated in Figure 4.21, which defines a rewrite rule to implement the `extend entity` construct. The `extend-entity` rule is an example of a *non-preserving* local-to-global transformation. That is, the term is rewritten to a placeholder term `RemoveMe()`, which is removed from the model in a later transformation step. In addition, the rule defines a dynamic rule `DynamicExtendEntity`, which rewrites the base entity definition to include the additional properties provided by the `extend entity` definition (as illustrated in Section 4.6.2). The context-sensitivity of the dynamic rule stems from the fact that it inherits the bindings to identifiers from its context. In this case, the identifiers *x* and *prop1** are bound by the left-hand side match of `extend-entity`. Thus, the uses of *x* and *prop1** in the definition of `DynamicExtendEntity` refer to these terms. That is, the properties *prop1** are propagated from the extend entity declaration to the entity declaration for *x*. The `:+` in the definition of the dynamic rule indicates that there may be multiple dynamic rule definitions with the same left-hand side, for the same entity in this case. This allows the rule to support multiple extensions of the same entity declaration.

4.8 TRANSFORMATION MODULARITY AND EXTENSIBILITY

Separation of concerns is a key strategy in model-driven software development. High-level, declarative models support separation of essential application properties from the code patterns used to implement them, such that the effort of development and maintenance of applications is drastically reduced. Separation of concerns is also important for the maintainability and extensibility of the *implementation* of domain-specific languages. Rapid extension and adaptation of a DSL is important to keep up with evolving insights and requirements.

The use of model-to-model desugaring transformations separates the implementation of high-level abstractions from the implementation of lower-level constructs. Rather than directly bridging the semantic gap between the model and its implementation in one transformation step, the model is gradually transformed. This approach is called compilation by normalization [Kats et al., 2008]. The code generator does not have to be aware of the presence of high-level constructs. The front-end can use the base language, instead of target language implementation patterns, to implement high-level constructs. Thus, this *vertical separation of concerns* realizes information hiding between compiler stages, and reduces the impact of extending the language.

Horizontal separation of concerns is information hiding within a compiler stage, designed to reduce the dependencies between transformations for different constructs. Again, it should be easy to add transformations for new abstractions without disrupting or otherwise affecting existing rules. For code generation, horizontal separation of concerns is achieved by the generation of partial artifacts, factoring out the assembly of these artifacts into a separate transformation. The `GenerateCode` rules can be extended without affecting existing rules. It is not necessary to locate an existing 'God rule' and plug into its assembly of a Java class.

```
webdsl-to-seam =
  webdsl-to-core
  ; webdsl-core-to-seam

webdsl-to-core =
  import-modules
  ; typecheck
  ; translate-workflow
  ; weave-access-control
  ; extend-entities
  ; derive-pages
  ; typecheck
  ; inline-templates
  ; derive-elements
  ; normalize-syntax

webdsl-core-to-seam =
  generate-code
  ; merge-partial-classes
  ; write-to-file
```

Figure 4.22 Pipeline of analysis and transformation stages.

In this section, we consider two architectures for the application of normalization rules from the perspective of extensibility. In the staged approach rules are divided in clusters, which are applied sequentially. In the normalization approach all rules are applied simultaneously.

4.8.1 *Staged Normalization*

A natural approach to organize a DSL front-end is as a pipeline of transformation stages that are associated with language extensions, as illustrated in Figure 4.22. The `webdsl-to-seam` compiler is divided into two parts. The front-end, `webdsl-to-core`, transforms applications in the extended WebDSL language to WebDSL core. The back-end, `webdsl-core-to-seam`, uses the partial class generation approach of Section 4.5. This pipeline model of the transformation is similar to workflow descriptions in other model transformation solutions, such as openArchitectureWare [Efftinge and Friese, 2007].

The staging approach achieves a form of separation of concerns by combining transformations associated with a language extension in a single transformation. However, the information hiding between stages is limited. Each stage should eliminate the constructs of the language extension that it is associated with. After a stage for a particular extension has been applied, the constructs from that extension can no longer be used. Thus, the position in the pipeline determines which language constructs can be used in the generation templates of that stage. Designing the transformations for a new language extension requires an analysis of the order of the stages to determine which language constructs can be used. Furthermore, adding an extension requires modification of the central pipeline, hampering independent extensibility of the language by third party developers. For WebDSL, which composes multiple sublanguages that focus on different aspects of web applications, this phase ordering problem formed a continuous, although slight nuisance for the authors of the various language extensions.

In addition to the presence of certain language constructs, transformation stages may also depend on the results of analysis stages. For example, the application of the `derive-input` rules requires type analysis to be applied to the argument expressions of `input` elements. If such `input` elements are generated by another stage, it may be necessary to perform an additional `typecheck` stage. We consider the problem of the combination of analysis and transformation in the next section. In this section we ignore the issue and focus on an extensible architecture for normalization rules.

4.8.2 *Innermost Normalization*

```
webdsl-to-core =
  import-modules
  ; typecheck
  ; desugar-top

desugar-top =
  innermost(
    normalize-text
    <+ normalize-for
    <+ derive-edit-page
    <+ derive-input
    <+ extend-entity
    <+ ...
  )
```

Figure 4.23 Pipeline with innermost normalization.

Desugaring rules translate high-level constructs into a combination of other constructs. If the result of a desugaring rule would always be a term in the core language, there would be no phase ordering problem. However, desugaring rules for high-level abstractions often rely on transformations for lower-level abstractions that are not core language constructs. For example, WebWorkFlow definitions are translated to a combination of entity extensions, page definitions, and access control rules, which need further desugaring before the back-end can translate them. Thus, we need to reduce the model to *normal form* with respect to the set of desugaring rules.

In the staging approach we try to find an ordering of desugaring rules that ensures that we reach this normal form. However, this requires a dependency analysis on the rules, which is not guaranteed to produce a linear ordering, as illustrated by need to reapply type analysis. The phase ordering problem can be avoided by applying all desugaring rules exhaustively using a fixpoint rewriting strategy. That is, any term that is produced as result of a transformation rule, is inspected to see if other transformation rules can be applied to it. (In Section 4.8.3 we discuss termination of this process for non-confluent rules.)

Figure 4.23 shows a reimplementation of the `webdsl-to-core` strategy. Instead of a sequential composition of desugaring stages, the `innermost` strategy is used to simultaneously normalize a model with respect to a set of desugaring rules. The `innermost` strategy takes as argument a transformation, typically a list of rules composed with deterministic choice (<+), which it exhaustively applies to the subject term starting with innermost nodes. That is, a term is only considered for transformation, after all its subterms have been normalized. When a rule is applied to a term, the strategy subsequently normalizes the newly produced term. Thus, right-hand sides of rules do not have to be restricted to terms in normal form. As a result it is not necessary to consider the order in which rules are applied.

To extend the new implementation of `webdsl-to-core` with desugaring rules, is simply a matter of adding rules to the argument of `innermost`. However, this still requires changing the definition of the pipeline. Extensibility is further simplified by Stratego's rule extension facility. Instead of directly passing a composition of rules to `innermost`, the strategy `desugar` is passed, which is defined by an extensible set of clauses, in the same manner as the extension of the `GenerateCode` strategy in Section 4.4.5. Defining rules in this fashion allows different desugaring steps to be defined across different modules, each extending the `desugar` definition, as illustrated in Figure 4.24. This means that different desugaring rules can be modularly defined and implicitly composed and evaluated, without the need for explicit staging. This helps in separation of concerns, and is essential for the scalability of a generator specification.

```
desugar-top =
  innermost(desugar)
```

```
desugar = normalize-text
desugar = normalize-for
```

```
desugar = derive-edit-page
desugar = derive-input
```

```
desugar = extend-entity
```

Figure 4.24 Extensible definition of `desugar`.

4.8.3 *Normalization with Local-to-Global Rules*

Local-to-global transformations may define new dynamic rules that need to be applied elsewhere in the tree. These could be elements of the model that have already undergone normalization. To ensure that these elements are again revisited by the `desugar-top` traversal, we extend the evaluation strategy with additional logic to ensure that all elements of the tree are revisited. Figure 4.25 shows an extended version of the `desugar-top` strategy that takes this into account. Using an additional dynamic rule `InnermostApplied`, it keeps track of whether there have been any rule applications in the current traversal. Note that this rule does not have a left-hand side, and is simply restricted to the values `True()` and `False()`, similar to a (scopeable) global variable. If `InnermostApplied` is `True()`, `desugar-top` repeats the traversal after the current pass is completed. This process repeats until no rules are applicable anymore, ensuring exhaustive application of both static and dynamic rules.

A number of transformations in the WebDSL compiler directly relied on the global staging approach used previously in the generator [Hemel et al., 2008a]. These features depend on explicit ordering of rewrite rules to handle non-confluent rewrite rules, or may be applied once or a limited number

```
desugar-top =
  do-while(
    rules (InnermostApplied := False())
    ; innermost(desugar; rules(InnermostApplied := True()))
    , InnermostApplied => True()
  )
```

Figure 4.25 Fixpoint iteration of `innermost`.

of times to ensure termination. Most rewrite rules *reduce* the input term to a form closer to a normal form (e.g., core WebDSL). One such rule is the local-to-global transformation rule `extend-entity` in Figure 4.21. It rewrites a construct to a term that is removed in a different rule, thus ensuring termination of the transformation. However, it also defines a new dynamic rule `DynamicExtendEntity`, which is a preserving global-to-local transformation. Application of that rule results in a term that it can be applied to again, resulting in an infinite sequence of applications.

In general, non-reducing rules can be controlled by explicitly specifying termination criteria. For instance, an additional dynamic rule could be used to indicate that a particular transformation has successfully completed for a particular element. For dynamically defined global-to-local transformations, however, Stratego offers a convenient feature to avoid this. For each dynamic rule, an additional strategy is derived that applies the rule and then removes its definition. We can invoke this strategy using the prefix '`once-`', and integrate it into the system as follows:

```
desugar = once-DynamicExtendEntity
```

For *non-confluent rewrite rules*, the order of their application affects their result. Similar to non-reducing rules, dynamic rules can be used to register which rules may be applied at what time. Using a dynamic rule to explicitly set and check a rule that maintains a stage number, a given set of rewrite rules can be explicitly and internally staged without requiring a global staging mechanism.

4.9 COMBINING TYPE ANALYSIS AND TRANSFORMATION

While the `innermost` strategy solves the extensibility problem of the staging approach, it does not solve the interaction between type analysis and desugaring transformations. In this section, we analyze the interaction problem and present a solution for the fine grained combination of analysis and transformation.

To understand the interaction between type analysis and transformation consider the transformation of `derive editPage` in Section 4.7.2. For example, given the entity definition

```
entity NewsItem { name :: String text :: Text }
```

type analysis annotates the identifier `i` in the page definition

```
typecheck-variable :
  Var(x) -> Var(x){Type(t)}
  where if not(t := <TypeOf> x) then
          typecheck-error(|["Undeclared variable ", x,
                            " referenced"])
        end
declare-page-argument :
  |[ x : srt ]| -> |[ x : srt ]|
  where if not(<TypeExists> srt) then
          typecheck-error(|["Illegal type ", srt,
                            " for parameter ",x])
        else
          rules( TypeOf : x -> srt )
        end
```

Figure 4.26 Rules of monolithic typechecker.

```
  define page editNewsItem(i : NewsItem) {
    derive editPage from i {NewsItem}
  }
```

with its type `NewsItem` (indicated in italics). Given this type, the `derive-edit-page` rule transforms the page definition to a page definition with a table and rows (slightly simplified):

```
  define page editNewsItem(i : NewsItem) {
    table(){ row{ "Name: " input(i.name) }
             row{ "Text: " input(i.text) } } }
```

After application of the `derive-edit-page` rule, the types of the expression generated as arguments of the `input` statements are unknown. Another round of type analysis is needed to determine the types of the expressions:

```
  define page editNewsItem(i : NewsItem) {
    table(){ row{ "Name: " input(i.name {String}) }
             row{ "Text: " input(i.text {Text}) } } }
```

Given these types, the `derive-input` rules can normalize the `input` statements:

```
  define page editNewsItem(i : NewsItem) {
    table(){ row{ "Name: " inputString(i.name {String}) }
             row{ "Text: " inputText(i.text {Text}) } } }
```

Thus, after application of the `derive-edit-page` transformation, type analysis is needed before the `derive-input` rules can be applied.

4.9.1 *Integrating Type Analysis and Transformation*

In the staged approach of Figure 4.22, desugaring transformations are interleaved with invocations of `typecheck` in order to add type information to newly generated terms. While the staging approach solves the problem of the interaction between analysis and transformation, it does so at a cost to the extensibility of the compiler. Since performing type analysis to the entire model after the application of each rewrite rule does not scale, the transformations are divided into coarse grained stages. The transformation stages have to be carefully composed such that type analysis is applied after the application of transformations introducing new, untyped expressions. Fitting in new transformations requires a careful analysis of all existing transformations.

If staging is not the answer to the interaction problem, maybe maintaining a fully typed representation is a solution. That is, require all transformation rules to produce a representation that includes all type and other analysis information that is needed to perform further transformations. For example, the `derive-edit-row` rule that is responsible for the generation of the `input` statements above, can be rewritten to ensure it includes a type annotation in its result:

```
derive-edit-row(|x) :
  |[ y relation srt ]| -> |[ row{x_text ": " input(e)} ]|
  where e-untyped := |[ x.y ]|
      ; e := <add-type-information(|srt)> e-untyped
```

With this modification, the rewrite rule now explicitly adds a type annotation to the generated expression `e`. While this approach is effective, it requires additional effort in the development of desugaring rules, and leads to rules that are harder to read and maintain. Furthermore, this violates the principle of separation of concerns by introducing logic related to type analysis in transformation rules. For more complex rewrite rules, the amount of code for type analysis can be significantly larger in size, and may require passing around type information of the surrounding context.

```
typecheck =
  rename-top
  ; check-constraints
```

Figure 4.27 Typechecker composed from name resolution and constraint checks.

We considered two solutions to the combination of analysis and transformation. Staging analysis and transformation is not extensible. Integrating analysis in transformation rules breaks separation of concerns. Is there a solution that maintains separation of concerns, yet is extensible?

The `typecheck` strategy employed in Figure 4.22 is a monolithic traversal that combines three functions, as will be later illustrated by the rules in Figure 4.29 through Figure 4.31: *name resolution*, identifying the declaration to which an identifier is associated; *type analysis*, assigning types to identifiers and expressions; and, *error checking*, checking type correctness of expressions and other constructs and generating error messages in case of violations. Factoring these three operations into separate, independently applicable sets of rules, is the key to fine grained combination of type analysis and transformation. Figure 4.27 shows the composition of `typecheck` as a name resolution phase (`rename-top`) and an error checking phase (`check-constraints`). Type analysis is applied during error

```
define page editing(u^{u0} : User) {
  form {
    input(u^{u0}.name)
    var u^{u1} : Blog
    input(u^{u1}.title) } }
```

```
TypeOf: u^{u0} -> SimpleSort("User")
TypeOf: u^{u1} -> SimpleSort("Blog")
Rename: u -> u^{u1}
```

Figure 4.28 Rename example

checking, but can now also be applied *on demand* during desugaring. We discuss these components and their application during normalization in the rest of this section.

4.9.2 *Name Resolution*

In name resolution, renaming rules annotate identifiers with unique keys. These are associated with the types of the referenced declaration, using a dynamic rule `TypeOf` that rewrites the annotated expression to its declared type. The annotations added are shared for identical identifiers, taking scoping rules into account to ensure that identifiers with the same name in different scopes get different keys.

Figure 4.29 shows two renaming rules, assigning unique keys to identifier declarations[37] and arguments of page definitions[38]. Each declared identifier is annotated with a unique name by the `add-naming-key` rule[41]. A dynamic rule `Rename`[42] is then generated, which annotates all uses of the declared identifier with the new key. To ensure that the rule is only applied in the lexical scope of its declaration, the dynamic rule is scoped[40]. The `TypeOf` rule, on the other hand, is not scoped. It binds the globally unique key to its associated type, for use in the phases that follow after this analysis. The `rename` rule for page definitions[39] renames all formal arguments *`farg1*`* in a similar fashion as is done for identifier declarations; it declares them as local identifiers in the page's scope.

The naming rules are applied in a top-down fashion by the `rename-top` strategy, which is defined to apply `alltd(rename)`. The `alltd` strategy traverses the tree top-down and attempts to apply its strategy argument to each node it traverses, when the application succeeds the traversal is stopped.

The example in Figure 4.28 illustrates the application of renaming rules. The partial page definition on the left is being renamed, the traversal just passed the last line shown. For the renamed identifiers, annotations are displayed in superscript. On the right the active dynamic rewrite rules at this point in the traversal are listed. The identifier declaration u^{u1} has shadowed the page argument u^{u0}, thus there is only one active `Rename` rule.

```
rename-top = alltd(rename)

rename :[37]
  |[ var x : srt ]| -> |[ var y : srt ]|
  where y := <add-naming-key(|srt)> x

rename-page-arg :[38]
  |[ x : srt ]| -> |[ y : srt ]|
  where y := <add-naming-key(|srt)> x

rename :[39]
  |[ define page x(farg1*) { elem1* } ]| ->
  |[ define page x(farg2*) { elem2* } ]|
  where {| Rename[40]
         ; farg2* := <map(rename-page-arg)> farg1*
         ; elem2* := <rename-top> elem1*
         |}

rename = Rename

add-naming-key(|srt) :[41]
  x -> y
  where y := x {<newname> x}
      ; rules (
          Rename : Var(x) -> Var(y)[42]
          TypeOf : y -> srt
        )
```

Figure 4.29 Name resolution rules.

```
type-of :
  |[ x ]| -> srt
  where srt := <TypeOf> x

type-of :
  |[ e.f ]| -> srt2
  where srt1 := <type-of> e
      ; srt2 := <type-of-property> (srt1, f)

type-of :
  |[ e1 + e2 ]| -> srt1
  where srt1 := <type-of> e1
      ; srt2 := <type-of> e2
      ; <type-compatible> (srt1, srt2)
```

Figure 4.30 Type analysis rules.

```
constraint-error :
  |[ x ]| -> ConstraintError(["Variable ", x, " not declared"])
  where not(<type-of> |[ x ]|)

check-constraints = where(
  collect-all(constraint-error)
  ; if not(?[]) then report-errors; <exit> 1 end
)
```

Figure 4.31 Constraint error rule and check constraints strategy.

4.9.3 *Type Analysis*

After name resolution, all identifiers in the model have a unique key, and an associated `TypeOf` rule that can be used to acquire its type, without the need for contextual (e.g., scoping) information about the identifier.

Context-sensitive transformations, such as those required for deriving page elements from entity types, make use of the type information made available by the name resolution phase. The `type-of` rules can be used to acquire the type of complete expressions. Figure 4.30 shows the definition of several of such rules. The first rule resolves the type of an identifier, which utilizes the `TypeOf` dynamic rule. The second rule calculates the type of an object field access, for instance `user.name`. It does so by first calculating the type of the object expression (`user`) and then retrieving the type of the property (`name`) of the object's type. A third rule calculates the type of the addition of two expressions, where it also checks that they are type compatible.

4.9.4 *Type Constraints*

One application of name resolution and type analysis is the static checking of the type correctness of a model. The `typecheck` strategy checks the model for any violations of type constraint rules. Type constraints can be simple checks for types (e.g., conditions must be booleans), or more sophisticated checks such as constraints on the nesting of user interface elements (e.g. all `input` elements must be nested within a `form`). As an example, consider the simple type constraint in Figure 4.31 stating that every identifier used needs to be declared. Constraint rules typically specify a negative condition in their `where` clause: this rule only produces an error if the constraint is violated. If there are any constraint violations, the complete set of errors is reported to the user and the compiler terminates, as shown in Figure 4.31. This style of constraint checking rules was inspired by oAW *Check* language [Efftinge and Völter, 2006].

4.9.5 *Type Analysis during Transformation*

Since type analysis has been separated from error checking and name resolution, it can be used on the fly during transformation. Thus transformation

```
desugar-top =
  do-while(
    rules(InnermostApplied := False())
    ; innermost(
        desugar
        ; {| Rename: rename-top |};
        ; rules(InnermostApplied := True())
      )
  , InnermostApplied => True()
  )
```

Figure 4.32 Fixpoint iteration desugaring with incremental name resolution.

rules that depend on type information, such as the derive rules in Figure 4.20, can use the `type-of` rules to compute the type of an expression without the need for a re-application of type analysis to the entire model.

Moreover, the specification of the transformation rules need not be concerned with name resolution. Figure 4.32 shows an extension of the `desugar-top` strategy. Each sucessful application of `desugar` is followed by an application of the `rename-top` rule, which adds unique keys to any identifiers that have not yet been annotated. By adding a scope for the `Rename` dynamic rule, the strategy ensures that any dynamic renaming rules derived from local declarations do not "leak" out of the context of the generated code.

While the initial name resolution phase performs a global analysis, applying `rename-top` during the transformation phase is not a context-sensitive operation. It only adds annotations to the generated fragment of code, with regard to scopes defined in that fragment. For example, consider the desugaring rule of Figure 4.33. It introduces pattern matching by means of a case statement into the language. To do so, among other things, it defines a fresh variable *x* with a new name "c". Processed by the `rename-top` strategy, this name is given a new, unique key associated with the local type *`srt`*. Any non-local definitions, such as identifier uses in the expression *`e`*, are left alone by the transformation, as they must be defined in the context of the generated fragment. However, as they are defined by the context, and not local to this transformation, it is safe to assume that these identifiers are already annotated with a unique key, copied from the left-hand side of the transformation rule.

Likewise, for the `derive-edit-row` global-to-local transformation of Section 4.7.2, the generated fragment is already annotated and requires no additional effort from the `rename-top` operation. As an optimization, `rename-top` can be cached to avoid inspection of (sub)terms that are already sufficiently annotated.

Figure 4.34 illustrates the case statement desugaring with a concrete example. The `show` page definition takes two arguments, a `Blog b` and a `String s`. The content of the page depends on the type of view requested in `s`, realized through a case statement. The case statement has been desugared on the right and introduces a new identifier `c0`^{c00} which has been automatically renamed after the application of the `desugar` rule which is called by `desugar-top` (see

```
desugar :
  |[ case(e) { alt* } ]| -> |[ { var x : srt := e; stat } ]|
  where srt := <type-of> e
      ; x := <newname> "c"
      ; stat := <case-to-if(|x)> alt*
```

Figure 4.33 Desugaring rule for case statements.

```
define page show(b^b0:Blog,
                 s^s0:String) {
  case(s^s0.toLowerCase()) {
    "all"{ showAll(b^b0) }
    ...
```

```
TypeOf: b^b0 ->
        SimpleSort("Blog")
TypeOf: s^s1 ->
        SimpleSort("String")
```

⇒

```
define page show(b^b0:Blog,
                 s^s0:String) {
  {var c0^c00 : String :=
        s^s0.toLowerCase();
  if(c0^c00=="all"){
    showAll(b^b0) }
  ...
```

```
TypeOf: b^b0 ->
        SimpleSort("Blog")
TypeOf: s^s1 ->
        SimpleSort("String")
TypeOf: c0^c00 ->
        SimpleSort("String")
```

Figure 4.34 Case statement desugaring applied.

Figure 4.32). Through this mechanism analysis information is kept intact during transformations.

4.10 DISCUSSION

In this section we discuss related work, evaluate the code generation by model transformation approach, and discuss future work.

4.10.1 *Compilation by Normalization*

Normalization of a rich language to a small core language is a well-known design pattern in programming language design and implementation, made popular in particular by the functional programming language Haskell [Peyton Jones, 2003]. Haskell is a large and complex language with many 'syntactic abstractions'. These abstractions are translated by the compiler front-end to a core language, close to the lambda calculus. Furthermore, the Glasgow Haskell Compiler (GHC) uses a transformation-based approach in its optimizer that relies on the application of cascading (small) transformation rules [Peyton Jones and Santos, 1998].

A difference with the approach in this chapter is that the core language in GHC is not a subset of the input language. Using an Intermediate Representation (IR) that is a subset of the source language is useful since it allows the result of compilation to be fed into the front-end of the compiler. This approach is for example taken by Kats et al. [2008] in the extension of Java with inline bytecode, which can be used by code generators for DSLs to flexibly combine (pre-compiled) bytecode and source code. The core language design pattern has also been used in the design of Stratego [Bravenboer et al., 2006b, 2008] and SDF [Visser, 1997b], the DSLs used for the transformation and syntax definition of WebDSL.

An approach related to compilation by normalization is the *nanopass compiler infrastructure* of Sarkar et al. [2004], which advocates the design of compilers as a long pipeline of very small stages in order to enhance the understandability of the compiler in an educational context. Each stage transforms the program to an intermediate, more low-level form. In contrast, our approach does not employ a strict separation between the application of the different stages, as doing so hinders compositionality of language features. For normalization rules that are non-reducing or non-confluent, individual rules may specify dependencies or restrict their application (discussed in Section 4.8.3), without introducing a form of globally staged application.

4.10.2 *Rewriting Tools*

The main ingredients for code generation by model transformation are (1) generation rules that generate structured representation instead of text, (2) concrete object syntax to make generation rules readable and maintainable, (3) integration of model-to-code, code-to-code, and model-to-model transformations, (4) incremental normalization of high-level models to low-level ones, (5) extensible definition of transformations, and optionally (6) integration of type analysis and transformation. In the WebDSL implementation and this chapter we have used the rewriting-based Stratego/XT toolset, which supports all of these ingredients.

Examples of other rewriting-based languages are ASF+SDF [Klint, 1993] and TXL [Cordy, 2006]. Both languages support the definition of transformations using concrete syntax and can be used to realize transformations on models and code. However, these languages provide limited programmability for strategies controlling the application of transformation rules. In contrast, Tom [Balland et al., 2007] supports rewrite rules and strategies. Tom is implemented as an extension of Java, using a preprocessor approach to map the Tom language features to standard Java. The language only supports the use of abstract syntax to specify patterns. Visser's survey of strategies in rule-based transformation strategies gives an overview of approaches to user defined strategies [Visser, 2005].

4.10.3 *Model Transformation*

In this subsection we reflect on the different aspects of typical MDE systems, and discuss how these compare to the approach presented in this chapter. We first focus our discussion on the transformation of models; in Section 4.10.4 we discuss code generation techniques.

An overview of MDE Toolkits

Since the advent of model-driven engineering, several modeling methodologies and model transformation approaches have been introduced. A classification of a number of such systems is given by Czarnecki and Helsen [2006]. Various MDE toolkits provide model transformation and code generation facilities, many of which are based on OMG's MDA (openArchitectureWare [Efftinge and Friese, 2007, Efftinge et al., 2007], AMMA [Kurtev et al., 2006], AndroMDA [AndroMDA.org, 2007]). Each approach is bound to a particular metamodeling language. A number of them share a standardized foundation, such as MOF [Object Management Group (OMG), 2006], Ecore [Budinsky et al., 2003], or KM3 [Jouault and Bézivin, 2006]). Other MDE tools are based on proprietary formats (DSL Tools [Cook et al., 2007], MetaEdit+ [Kelly et al., 1996]).

The different MDE toolkits prescribe varying model transformation languages, such as ATL [Jouault and Kurtev, 2006], openArchitectureWare's xTend [Efftinge and Friese, 2007], and QVT [Bast et al., 2005]. The current crop of MDE toolkits are characterized by using a separate language for code generation, such as TCS [Jouault et al., 2006], xPand [Voelter and Groher, 2007], Velocity [The Apache Foundation, 2007]). In general, they also use a separate language to define a sequence of transformations, or to combine model transformations and code generation. Examples include openArchitectureWare's workflow language [Efftinge and Friese, 2007] and the Groovy scripting language [AndroMDA.org, 2007], employed by AndroMDA.

Representation of Models

Using a common metamodeling language can improve interoperability between tools. A number of standardized metametamodels have been developed, such as MOF, Ecore, and KM3. In Stratego/XT, SDF grammars are used as a corresponding notion. These are serialized using the ATerm format, and may be used to interoperate with other tools based on the same technology, such as ASF+SDF [Klint, 1993].

Using a common metamodeling language should not be considered a "silver bullet" for interoperability, however. In practice, metamodels designed using different tools are often incompatible. Similarly, metamodels designed using the same tool, for the same concrete syntax representation of a known language, may lack compatibility. Only when using identical metamodels can models be exchanged across tools. Alternatively, a well-defined (textual) concrete syntax representation may be used to exchange models between tools.

Model management can be based on any algebraic datastructure such as trees, graphs, hypergraphs, or categories [Bézivin, 2006]. Most current MDE toolkits are based on graphs, while Stratego/XT uses a tree-based representation.

Trees are acyclic, directed graphs. This nature allows them to be efficiently stored using maximal sharing, ensuring a significant decrease in memory usage [van den Brand et al., 2000]. Based on maximal sharing, all identical subtrees occupy the same space in memory, allowing constant-time equality tests between branches using pointers. Moreover, terms that are copied can simply be copied as pointers, while modified terms can be efficiently reconstructed (maintaining maximal sharing) rather than destructively updated. In contrast, using destructive updates parts of the tree that may be shared or used in different contexts will be modified in-line. Any local rewrite is performed using a destructive update, as is typical in graph-based rewriting.

Using a tree-based structure allows for simple, intuitive specifications of traversals with clear termination criteria. For this reason, many graph-based systems employ a spanning tree, imposing a tree structure on a graph. In contrast, in Stratego the tree structure is the principal representation. By use of dynamic rules, Stratego can conversely impose graph structures on trees. This makes it possible to model context-sensitive information that cannot easily be expressed using mere trees. For example, a dynamic rule can be used to resolve an identifier reference, essentially connecting the identifier node to the declaration node. We described how dynamic rules can provide context information for global-to-local and local-to-global transformations in Section 4.7.

An alternative approach to using dynamic rules to represent graphs as trees is Balland and Brauner's approach of using de Bruijn indices in terms implemented in Tom. [Balland and Brauner, 2008] In this approach term paths are used to point to terms. For instance in the term `f(s(a, 1.1))`, the path `1.1` will refer to `a` by first taking the first child of `f` and then the first child of `s`. Relative paths can also be represented, such as `f(s(a, -1.1))` in which `-1` indicates going one term level up and navigating from there. Again, `-1.1` refers to `a`. This approach works well in simple cases where little transformations are performed. In the case of a large number of much more complex transformations, such as the transformations demonstrated in section 4.6, it becomes tedious to keep such paths up to date during transformations.

Transformation Workflow

Stratego/XT does not employ a separate workflow language, but allows the Stratego language itself to control the application of transformations. In contrast, other approaches use a separate language such as Groovy, or a dedicated workflow language as is used in openArchitectureWare. For both examples, a lack of linguistic integration results in a lack of static checking for the validity and definedness of transformations that are specified.

Consistency management

Consistency management is an important issue in MDE [Mens and van Gorp, 2006]. In principle, models can be kept consistent as part of a transformation. In practice however, doing so tends to make transformations much more complex. In our approach we chose to separate the concern of typechecking from the model transformation at hand. The drawback of this approach is that models need to be reanalyzed after applying transformations. Incremental analysis and transformation techniques are an important research topic.

By analyzing models before any transformations are performed, we detect inconsistencies early and can report them to the developer. However, prob-

Mandatory requirement	**Supported**
1. Language for querying models	+
2. Language for transforming models	+
3. Meta-models for languages in MOF 2.0	-
4. Expressive transformations	+
5. Creation of views of meta-models	+
6. Declarative transformations enabling incremental changes	+/-
7. Meta-models specified in MOF 2.0	-
Optional requirement	**Supported**
1. Bidirectional transformations	-
2. Transformation traceability	+/-
3. Generic transformation definitions	+
4. Transactional transformations	+/-
5. Other sources of data	+
6. Model updates	+

Figure 4.35 The Stratego/XT platform and the QVT RFP requirements

lems that occur while the system is running turn out to be difficult to trace back to errors in the model. In the future, we intend to investigate the feasibility of origin tracking [van Deursen et al., 1993] to achieve code-to-model traceability.

Transformation languages such as ATL and xTend allow transformations to be separated in modules, similar to Stratego. However, extensibility of transformations is more difficult to realize, especially if transformation extensions have to operate on the same modeling elements, which is forbidden in ATL, for instance. In existing MDE toolkits, vertical modularity in transformations is often realized using a separate workflow language, such as the oAW workflow language and Groovy in AndroMDA. Stratego not only integrates model-to-model and model-to-code transformations, but also the overall generator workflow. Thus, a single transformation composition language is used for micro and macro compositions.

QVT Request for Proposals

The Query/View/Transformation (QVT) request for proposals [Object Management Group (OMG), 2003] sparked much interest in the development and comparison of different tools for model transformations. Figure 4.35 summarizes the compliance of Stratego/XT as such a tool according to the mandatory and optional requirements as specified by the OMG.

Outlining the mandatory requirements of Figure 4.35, Stratego/XT provides excellent support for specifying querying and transformations of models (1,2). In particular, it supports querying using a combination of traversal and pattern matching specifications, and provides the APath library [Janssen, 2005] for XPath-like queries. Stratego/XT does not define the abstract syntax of the query, view, and transformation languages in MOF 2.0 (3,7), but uses

ATerms instead, where SDF grammars define the meta-model. As Stratego is implemented in itself, the abstract syntax of Stratego is itself also defined in SDF. The present chapter demonstrates how the Stratego language is capable of expressing all information required to generate a target model from a source model automatically, as required by (4). Stratego can be used to implement views through transformations (5). For example, a view of all names of pages defined in a WebDSL model can be generated through a transformation that collects all page names in the model. As meta-models are defined in SDF, which can also be transformed using Stratego, views can also be constructed at the meta-model level. Technically, it is possible to incrementally apply changes to source code model into changes in a target model using Stratego (6). However, this incrementality does not come for free, and requires additional work in the implementation of the transformations.

Outlining the optional requirements of Figure 4.35, transformations cannot be applied bi-directionally (1); reverse transformations have to be specified separately. It does not provide traceability by default (2). However, with additional runtime or library support limited traceability capabilities can be added [Kats et al., 2009]. Support for reusing and extending generic transformations is where Stratego really shines (3). Using strategies it is much easier to reuse transformations and traversals than in approaches that do not have such a notion. Although there is no notion of term inheritance; generic transformations based on inheritance structures as suggested in the request for proposals can be supported by emulating inheritance using strategies that implement `is-a` behavior and accessor strategies that set and get properties of terms generically. Stratego uses copy-on-write semantics for transformations, and allows rollback using the <+ operator (4). However, applying the system in an asynchronous context is considered future work. Access is provided to other sources of data beyond the input model, such as file access and execution of external programs (5). Stratego allows (non-destructive) updates of models (6).

4.10.4 *Code Generation*

Some other approaches have generated partial artifacts through the use of partial classes, which are then combined by the regular compiler for the target language. Warmer and Kleppe [2006] describe experiences with such an approach. These approaches rely on the target language to support this features. In our approach, code is treated as a model, while most MDE approaches generate code through the use of textual template engines, which produce plain text, not amenable to further transformation. By treating generated code as a model, it is possible to extend the target language and add convenient language features such as partial classes and methods, and interface extraction.

Generation of partial artifacts has also been applied by Huang and Smaragdakis [2006] by use of Meta-AspectJ [Zook et al., 2004]. Rather than using a full-fledged AOP (meta-)programming language, our approach makes use of the standard Java syntax, with only a small semantic extension. By integration of this functionality in the generator, our approach is independent of the capabilities of the target platform.

There have been other approaches that aspect weaving at the model level rather than using this feature in the generated code [Suzuki and Yamamoto, 1999, Kulkarni and Reddy, 2008]. In contrast, in our approach we overlay the feature of partial classes and methods directly on the output language. This overlay definition can be used across different applications, i.e. other code generators that produce Java code. In contrast, using the more typical approach of strictly separating model transformation and code generation (using templates), as applied in Suzuki and Yamamoto [1999], Kulkarni and Reddy [2008], a very low-level, general-purpose model representation would have to be used to achieve the same result.

Arnoldus et al. [2007] have developed Repleo, a template engine that takes a similar approach to our CGMT approach in that it creates a new syntax, mixing the target language with the template language. However, the focus of Repleo is solely on the code generation aspect — it is a template language, not a transformation language. Rather than using the transformation capabilities of e.g. Stratego or ASF+SDF, Repleo provides its own small template language, very similar to string-based template engines, except that Repleo can guarantee that the resulting program is syntactically correct.

4.10.5 *Web Application Generators*

Many (visual) languages for modeling web applications have been developed, including WebML [M. Brambilla and Matera, 2007], MIDAS [P. Cáceres, 2003], OOWS [O. Pastor, 2003], Netsilon [Pierre-Alain Muller and Bézivin, 2005], and UWE [A. Kraus and Koch, 2007]. UWE generates JSP (Java Server Pages) code via a model representation conforming to a JSP metamodel. Netsilon uses an intermediate language for code generation in order to increase retargetability of the generator. The other approaches use textual, usually template-based code generation. WebML interprets its models rather than generating code from them.

Most approaches apply model transformations with the purpose of retargetability, or with the purpose of expressing "as many artifacts as possible using models as this allows for processing these artifacts using model transformations" [Voelter and Groher, 2007]. Only Netsilon actually models the target source code (but then only XML).

4.10.6 *Evaluation*

Throughout this chapter we have demonstrated how generator concerns can be better separated. We showed how transformation rules can be made more concise and modularized by extending the target language. We discussed several ways of combining type analysis with rewriting and introduced the approach of three-phased type analysis and transformation, in which name resolution, constraint checking, and rewriting can all be specified as strictly separate concerns.

When additional language abstractions are introduced, they can take advantage of the open extension points provided by the generator. These extension points, as described in Section 4.8 allow the extension to easily plug into the type analysis, model transformation and code generation subsystems. We built a number of language extensions into the generator, most notably the access control and workflow extensions, which are entirely built by plugging into the extension points mentioned.

4.10.7 *Future Work*

A focal point of the present chapter has been to provide an extensible mechanism for language specifications, using the Stratego language and specialized strategic programming idioms and library support. In the future, we would like to further investigate this area, in particular by providing specialized tool and language support for such specifications. Stratego is, as a strategic programming and term rewriting language, a very flexible platform for these endeavors, but specialized tooling could simplify the implementation of these idioms and provide additional static checks.

A number of other tools use attribute grammar to specify analyses, which provide a high-level, declarative, and effective way to specify analysis on trees [Paakki, 1995]. Modern attribute grammar systems such as Eli [Gray et al., 1992], JastAdd [Ekman and Hedin, 2004], and Silver [Wyk et al., 2007] offer many specialized features with respect to the analysis of software languages. More recently, Aster [Kats et al., 2009] used strategic programming to abstract over common patterns in attribute equations. Specifications made with these systems are highly modular and extensible. Unfortunately, they are lacking in their support for transformations, particularly for context-sensitive transformations that depend directly on the type analysis, and vice versa. Ongoing work focuses on integrating the two paradigms: using the expressiveness of a strategic term rewriting system for transformations and the high-level specification capabilities of attribute grammar systems for analysis. Challenges in this area include the integration of analysis and transformation: as a tree changes through transformation, this should have a proportionate effect on the attribute evaluation system. Normal attribute equations are declarative definitions of immutable properties of nodes in the tree, and may be memoized and computed on demand. In the context of a rewriting system, where the tree is a mutable model that undergoes multiple transformations, these computation principles must be reconsidered. This also raises the question of how control should be determined in a system that combines attribute equations and rewrite rules. Different designs of explicit control, eager or on-demand evaluation, and other approaches can be taken, and have a large impact on the effectiveness of such a system.

4.11 CONCLUSION

In this chapter we described several techniques to improve separation of concerns in DSL generators. The core technique is *code generation by model transformation*. The key idea behind code generation by model transformation is to represent both the source model and target code as terms. Current practice is often to directly generate plain text code, using template engines. We demonstrated that generating code by term rewriting has a number of advantages, for instance the ability to ensure syntactical correctness of generated code and the ability to perform further transformations on generated code. This enables extension of the target language with features such as partial classes and methods which greatly improve the modularity and size of rewrite rules.

We have shown how high level abstractions can be built on top of a relatively small core DSL language. Abstractions are gradually transformed to core DSL elements in a process of *compilation by normalization*. We have argued that the advantage of implementing such abstractions as model transformations is that by keeping the core DSL small, the generator becomes more portable, making it feasible to develop multiple generator back-ends.

Many transformations rely on the availability of contextual information, such as type information. In previous work we discussed the difficulty keeping type annotations up-to-date as a model is transformed. In this chapter, we introduced a novel approach in which type analysis and rewriting are combined while still keeping the analysis and rewriting generator concerns separate. Repeatedly reanalyzing the entire model, the approach we previously took, is therefore no longer necessary.

PIL: A Platform Independent Language for Retargetable DSLs

5

ABSTRACT

Intermediate languages are used in compiler construction to simplify retargeting compilers to multiple machine architectures. In the implementation of *domain-specific languages* (DSLs), compilers typically generate high-level source code, rather than low-level machine instructions. DSL compilers target a software platform, i.e. a programming language with a set of libraries, deployable on one or more operating systems. DSLs enable targeting *multiple* software platforms if its abstractions are platform independent. While transformations from DSL to each targeted platform are often conceptually very similar, there is little reuse between transformations due to syntactic and API differences of the target platforms, making supporting multiple platforms expensive. In this chapter, we discuss the design and implementation of PIL, a Platform Independent Language, an intermediate language providing a layer of abstraction between DSL and target platform code, abstracting from syntactic and API differences between platforms, thereby removing the need for platform-specific transformations. We discuss the use of PIL in an implemementation of WebDSL, a DSL for building web applications.

5.1 INTRODUCTION

Intermediate languages have been used in compiler construction since the 1960s [Steel, 1961] to improve the retargetability of compilers. Rather than generating machine architecture specific instructions directly, compilers emit machine-independent instructions written in a low-level intermediate language, which is subsequently translated into machine-specific instructions by machine-specific compiler back-ends.

In the context of model-driven engineering, research has been focusing on the development of compilers for *domain-specific languages*. Domain-specific languages (DSLs) raise the level of abstraction in software development by providing constructs to express high-level concepts from which lower-level implementations are generated. Ideally, compilers that implement the DSL are reused to develop multiple applications for multiple customers. Rather than generating executable machine code, DSL compilers typically generate source code written in languages such as Java or Python. By generating source code rather than machine instructions, DSL compilers abstract from the low-level machine instructions that compilers typically produce. In addition, source code is much simpler to generate and DSL compilers can therefore be developed much more efficiently.

Generating source code instead of machine instructions poses a new retargetability challenge at the level of *software platforms*. A software platform consists of one or more programming languages with a set of libraries and frameworks, deployable on one or more operating systems. Dozens of software platforms compete and companies typically standardize on a single one (e.g. Sun's Java, Microsoft .NET or LAMP[1]). Consequently, DSL vendors have to choose whether to generate code for a single software platform, or multiple software platforms. Ideally, a DSL compiler targets many platforms, to maximize its potential customer base. Whereas encoding implementation knowledge of domain-specific concepts in a compiler enables the reuse of this knowledge between *applications*, there is little reuse between the different *back-ends* of such a compiler, due to language and library discrepancies between platforms. This lack of reuse causes significant maintenance problems. For instance, the ANTLR parser generator [Parr and Quong, 1994] has code generator back-ends for over a dozen platforms. However, many of them are not up-to-date with the latest ANTLR release. Similarly, WebDSL [Visser, 2008], a DSL for data-intensive web applications, has back-ends for Java and Python, but whenever a new feature is added to WebDSL, it needs to be implemented and maintained in each back-end individually, in practice leading to incompatible platform back-ends.

Back-end maintenance is an even more prominent issue when back-ends heavily rely on the target platform's syntactic sugar and platform-specific frameworks and libraries. Such platform features are designed to enable *developers* to be productive coding on that platform. When code is *generated*, however, such productivity features are of less value. Specifically, these features complicate the implementation of multiple back-ends with consistent behavior, due to incompatible semantics across platforms. Thus, to fully control the behavior of generated code, and consequently the behavior of the DSL, lower-level code is generated using only a subset of the target platform. Conversely, features that are beneficial to code generators are often lacking in programming languages. Therefore, generating monolithic code artifacts, such as complex classes, can result in large and complex code generation rules. Such large rules can be circumvented by extending the target language with code compositionality features such as partial classes and methods enabling small code generation rules that emit smaller artifacts. Similarly, code generation features such as identifier concatenation and expression blocks substantially reduce the size of generation rules.

The lack of reuse between compiler back-ends could be circumvented by performing automatic language translation, e.g. translating generated Java code to Python, but this translation is expensive because of the complexity of the Java language and its libraries. Efforts to port dynamic languages, specifically Ruby and Python, to the CLR (IronPython, IronRuby) and JVM (JRuby, Jython) so that software written in these languages is portable to these

[1]Linux, Apache, MySQL and Perl/Python/PHP

platforms, are also very complex, often incomplete and have performance issues.

In this chapter we introduce the intermediate language PIL, a Platform Independent Language providing a level of abstraction between DSL and software platforms, abstracting from discrepancies between platforms, thereby removing the need for platform-specific back-ends. In contrast to intermediate languages in traditional compiler construction, PIL operates on a higher level of abstraction and has a concrete syntax, based on a subset of Java, leveraging the productivity advantages of generating source code over generating machine instructions. PIL is designed as a small intermediate language, capturing only essential object-oriented constructs and is therefore easier to port to multiple platforms than Java, for instance. In addition, the design of a language specifically targeted at code generators enables the development of code-generation specific language features. PIL/G, a thin layer of abstraction on top of PIL, provides some of such code generation such as partial classes, partial methods, identifier concatenation and expression blocks. In the future we also see opportunities to integrate DSL debugging support as part of PIL/G. We realized an implementation of the Java and Python backends of the WebDSL compiler using PIL, reducing the maintenance effort of these back-ends.

5.1.1 *Contributions*

The contributions of this chapter are as follows:

1. The design of the PIL language, an intermediate language at the source code level aimed at DSL compilers.
2. PIL/G, a collection of code generation-specific abstractions built on PIL.
3. An evaluation of our approach by implementing a Java and Python back-end for WebDSL through the use of PIL.

5.1.2 *Outline*

In the next section we describe the typical architecture of a DSL generator with a single back-end. In Section 3 we discuss several approaches to extend this architecture to generate code for multiple platforms. Section 4 describes PIL and its design and features. In Section 5 we discuss how PIL interacts with platform-specific code. In Section 6 the applicability of PIL, future work and related work is discussed.

5.2 CODE GENERATOR ARCHITECTURE

In this section we describe the general architecture of a code generator generating code for a *single* platform. We examine how to cater for *multiple* platforms in the next section. The initial single back-end generator architecture is depicted in Figure 5.1. It is composed of two parts: the front-end, which parses, checks and desugars models described in the DSL, and the back-end, which generates code from the model. We first give a brief overview of the operation of the generator front-end, followed by a discussion of generator back-ends.

The generator front-end The front-end of the generator is responsible for parsing, checking and transforming a model to a simplified representation from which a back-end produces executable code. Based on the grammar of a DSL (which also defines the DSL's meta-model), the parser produces an abstract syntax tree (AST). The AST is subsequently checked for inconsistencies, such as type errors and other deficiencies.

A set of model-to-model transformations, also known as *desugarings*, transform the AST to a simplified, core DSL model. Normalizing transformations perform model simplification, such as adding default values for omitted optional information. More complex transformations transform higher-level constructs to a reduced set of lower-level constructs. For example, in WebDSL, access control [Groenewegen and Visser, 2008] and workflow (Chapter 3) are implemented as abstractions on top of the user interface, data model and action sub-languages, implemented through a number of model-to-model transformations. The result of the front-end transformations is a fully checked, normalized model represented in a reduced set of *core DSL constructs*.

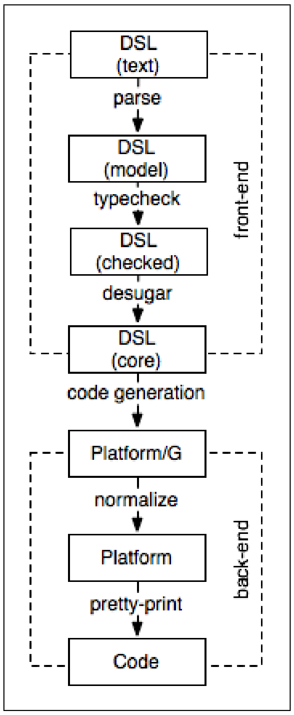


Figure 5.1 Code generator architecture

The generator back-end A generator back-end generates code from a core DSL model produced by the front-end. Intuitively, generating code that uses a high-level framework seems an attractive, productive option [Stahl et al., 2006, van Deursen et al., 2000a]. However, in the long term, frameworks often become too restrictive when more control is required over the exact execution of the generated application [Groenewegen et al., 2008]. Ini-

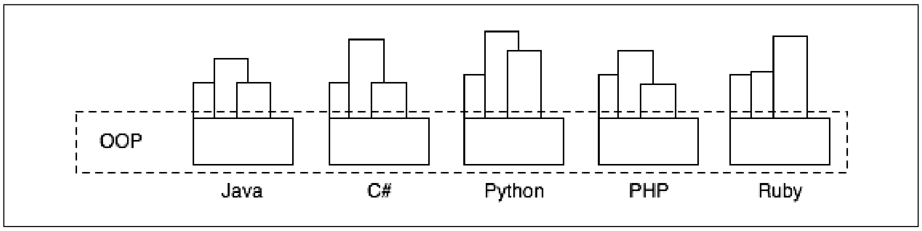


Figure 5.2 Platforms and their features

tially, the WebDSL compiler generated code for the JBoss Seam framework, a high-level Java framework utilizing Enterprise Java Beans (EJBs) for the business logic, Hibernate for models and JSF (Java Server Faces) for constructing views. As the WebDSL language evolved, JSF in particular, became too restrictive. The WebDSL view models no longer were a good match for JSF. Consequently, JSF and EJBs were replaced by plain Java servlets that contain `println` statements printing HTML code.

There is mismatch in platform requirements between *developers* and *code generators*. Many modern software platforms (e.g. Java, .NET, Ruby, Python and PHP) are object-oriented at their core. Platforms typically try to differentiate by adding features on top of that core (Figure 5.2), to improve the expressivity for *developers*, e.g. syntactic sugar and high-level frameworks. While improving developer productivity, these features limit flexibility, because they are only optimized for common use cases. By generating lower-level code, the execution of the generated application can be more effectively and flexibly controlled. In addition, because the object-oriented core of these platforms is similar, generating code at this level also significantly improves portability, which is discussed extensively in section 5.3. Although lower-level code is more verbose, the code is not intended to be read or modified, so this is not an issue.

Conversely, platform features that enable clean and concise code generation rules are often absent from platforms. Generation rules that generate large code artifacts, typically become very long and complex. Such "God rules" dispatch a large number of smaller generation rules to generate a monolithic target artifact (e.g., a Java class). "God rules", similar to "God classes" in object-oriented programming, are an anti-pattern and can be avoided by the use of code compositionality features, specifically partial classes and methods.

```
@Partial
public class SomeClass {
  private int a;

  @Partial
  public void init() {
    a = 10;
  }
}
// ...
@Partial
public class SomeClass {
  private int b;

  @Partial
  public void init() {
    b = 8;
  }
}
```

Figure 5.3 Partial classes and methods added to Java

Partial classes are class fragments that are combined at compile time by merging their contents. Similarly, partial methods enable fragments of a method to be distributed over multiple partial classes. Partial methods are also merged at compile-time. Some languages support partial classes, e.g. Smalltalk, Objective-C, C# 2.0, Common LISP (CLOS) and Ruby, but many other languages do not support this feature, e.g. Java and PHP. Partial methods are less common. C# 3.0's partial method support is different than the partial methods just described; partial methods in C# are an optimization feature for providing hooks into generated code. Partial classes

```
define page blogEntry(e : BlogEntry) {
  section {
    header { outputString(e.title) }
    outputText(e.content)
  }
}
```

Figure 5.4 A simple page definition in WebDSL

in C#, typically generated by a code generator, can declare the signature of a method as partial, meaning that if the method is implemented in another partial class with the same name, typically defined by a programmer, it operates as a regular method. However, if the method is not implemented in another partial class, all calls to the partial method are removed.

In Chapter 4 we presented the *code generation by model transformation* approach. The key idea of this approach is to represent code as a model, enabling further transformation of generated code. Compositionality features such as partial classes and methods can be implemented by extending the target language. Figure 5.3 shows an example of Java/G, Java extended with partial classes and methods, marked with `@Partial` annotations (in Figure 5.1 the generalized form of this language is referred to as Platform/G). The compiler's transformation rules emit fragments of Java/G code, which are subsequently merged and written to files.

As an example of a code generation rule, we illustrate how Java code is generated from WebDSL page definitions using the Stratego/XT transformation toolset. WebDSL is a domain-specific language for data-intensive web applications [Visser, 2008]. It has sub-languages for the definition of data models, user interfaces, access control, workflow and business logic. The WebDSL page definition in Figure 5.4 defines a page `blogEntry` with an argument of type `BlogEntry`. The view of the page defines a section, consisting of a header with the title of the blog entry and its content. Figure 5.5 defines a Stratego/XT rewrite rule `page-to-java`, which transforms a WebDSL page to a Java class. When the left-hand side of the rule (before `->`) is matched its meta variables *x_page*, *farg**, *elem** are bound to their corresponding values. The right-hand side of the rule defines the generated Java/G code. In the `where` condition, individual page elements are mapped to Java statements using the `elem-to-java` rule. The code pattern in the left and right-hand side of the rule use the *concrete syntax* [Bravenboer and Visser, 2004] of the source and target languages, respectively. Code enclosed in `|[` and `]|` quotations is internally parsed by Stratego and turned into its AST representation. Consequently, the `page-to-java` rule matches an *AST representation* of a page definition and produces a Java/G AST, rather than textual code.

```
page-to-java :
  |[ define page x_page(farg*) { elem* } ]| ->
  |[ package pages;
     import javax.servlet.http.*;
     import java.io.*;
     @Partial
     public class x_page extends Page {
       public void renderPage(Request req, Response res) {
         PrintWriter out = res.getWriter();
         out.print("<html><head><title>"+getPageTitle()+
                   "</title></head>");
         out.print("<body>");
         stat_elem*
         out.print("</body></html>");
       }
     }
  ]|
  where stat_elem* := <map(elem-to-java)> elem*
```

Figure 5.5 Rewrite rule that transforms pages to Java classes

5.3 RETARGETING A DSL GENERATOR

In this section, we evaluate three approaches to extend the single platform compiler architecture to support an additional platform. The first approach is *copying* the existing *back-end* and porting the transformations to the new target platform. A second approach is *translating code* generated by the already present back-end to a new platform. As a third aproach, we argue that high-level intermediate languages provide a better approach to supporting multiple platforms in a DSL generator.

5.3.1 *Adding a Backend to a Generator*

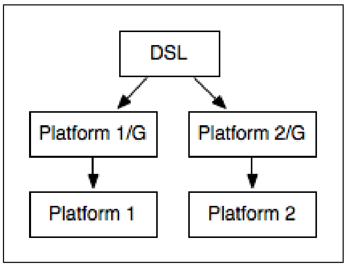


Figure 5.7 Adding a backend to a generator

The most intuitive approach to support an additional target platform in a DSL implementation is copying an existing back-end and porting it to generate code for the new platform (Figure 5.7). Generalizing this approach, supporting *N* platforms requires *N* back-end implementations. Figure 5.6 shows how the `page-to-java` rule (Figure 5.5) has been ported to generate Python code. A comparison of the Java and Python back-ends suggest an additional advantage of generating low-level code: the syntax and low-level APIs do not differ that much between platforms. The main changes that have to be made to port a back-end

```
page-to-python :
  |[ define page x_page(farg*) { elem* } ]| ->
  |[ @partial
     class x_page(Page):
       def renderPage(self, req, res):
         out = res.writer
         out.print("<html><head><title>"+pageTitle()+
                   "</title></head>")
         out.print("<body>")
         stat_elem*
         out.print("</body></html>") ]|
  where stat_elem := <map(elem-to-python)> elem*
```

Figure 5.6 A Python version of the `page-to-java` rules (Figure 5.5)

are syntactic and relate to minor API differences. Although there is conceptual reuse between back-end generation rules, there is no code reuse between them, resulting in large-scale code duplication. In addition, code duplication also occurs in the reimplementation of code compositionality features for each back-end. Code duplication gives rise to maintenance problems. For instance, when the DSL is changed, modifications have to be propagated to all back-ends.

5.3.2 *Language Translation*

Efforts to translate dynamic languages, specifically Ruby and Python, to the CLR (IronPython, IronRuby) and JVM (Jython, JRuby) bytecode and Microsoft's *Java Language Conversion Assistant* to translate Java to C# code appear an attractive option to build retargetable software. Since only one transformation from the DSL to one of these platforms needs to be defined, this approach would solve the code duplication issue in generator back-ends. Figure 5.8 depicts this scenario. Transformations that port code from one platform to another are reusable in multiple generators. In addition, code compositionality language extensions have to be implemented only once and need not be ported. However, language ports are problematic due to sheer language complexity, performance issues and the fact that these languages and their platform libraries are not designed to be portable across platforms. Consequently, this approach does not scale well.

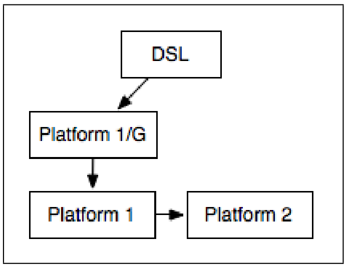


Figure 5.8 Language translation

5.3.3 *High-Level Intermediate Languages*

Although not feasible in the general case, porting a language (such as Java, Ruby or Python in the previous section) to multiple platforms is attractive, because multiple similar back-ends need no longer be maintained. When generating code, only a low-level subset of the platform is used. Software platforms, at this level, are very similar. Therefore, only a port of a *subset* of the platform is sufficient to retarget a DSL.

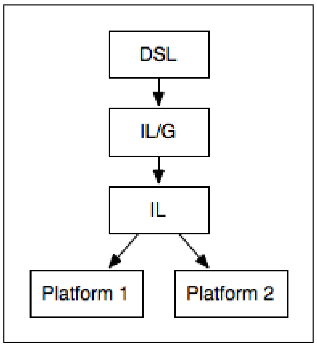


Figure 5.9 High-level intermediate language

One approach is to generate code for an existing platform, e.g. Java, and by *convention* only use a subset of that platform. Translations to other platforms are defined only for this subset. The problem with this approach is the difficulty to *enforce* it. In addition, as programming languages are typically not designed to be easily translatable to other languages, there may be hurdles to do so. An example of this is Java's `.` (dot) operator, whose meaning at the syntactic level is ambiguous and therefore requires type analysis to disambiguate, requiring language translators to perform such analysis. An alternative approach is to *formalize* a high-level intermediate language. Naturally, this intermediate language can be based on the subset of an existing language, but it can also be further simplified and extended with code generation features.

Typically, the most expensive transformation in a DSL compiler is the transformation from the DSL to target platform code. This transformation is expensive because of the large *semantic gap* between DSL and platform code. Thus, the number of times that this transformation needs to be implemented should be limited as much as possible. A well-known technique in compiler construction is the use of intermediate languages [Steel, 1961, UCSD, 1981]. By using an intermediate language, the maintenance of the compiler is much improved, since only one complex transformation from the DSL to the intermediate language needs to be implemented and maintained. Furthermore, the semantic gap between the intermediate language and the platform is very small, enabling implementations of the intermediate language for new platforms to be developed with little effort. Such intermediate language implementations are reusable in multiple DSL generators. Code compositionality features, as well as other features convenient for code generation can be implemented as an abstraction on top of the intermediate language, implemented as a transformation. The architecture of this scenario is depicted in Figure 5.9.

5.3.4 *Evaluation*

Figure 5.10 compares the costs of the three approaches to construct retargetable DSLs. As the transformation from the DSL to platform code is expensive, the first approach, where *N* supported platforms require *N* back-end implementations, is not a desirable solution. In the second approach, language translation, only one DSL to code transformation needs to be implemented. However, the language translation C, although reusable in multiple compilers, is very expensive to implement due to the high cost of implementing full language translations. Using an intermediate language requires only one transformation from the DSL to code written in the intermediate language. Implementing translations from the intermediate language to Platform 1 and 2 (A' and B') is cheap because of the small size of the intermediate language. In addition, these translations only need to be implemented once and are reusable in multiple compilers. Therefore, a future DSL compiler is only required to implement the transformation from the DSL to the intermediate language.

5.4 PIL: A PLATFORM INDEPENDENT LANGUAGE

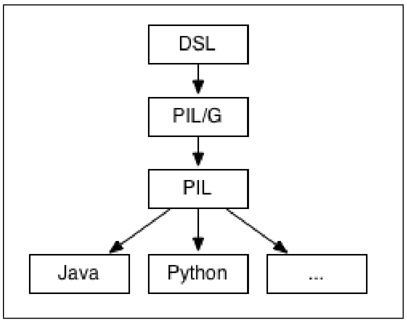


Figure 5.11 The PIL architecture

We have developed PIL, a Platform Independent Language designed for code generation, abstracting from syntactic differences between object-oriented languages, slight mismatches between common data types, and providing infrastructure to interact with underlying platforms. In contrast to traditional intermediate languages as used in compiler construction, PIL is used at a higher level of abstraction and has a convenient concrete syntax enabling source code generation through the use of code generation rules. Compared to typical programmer-oriented software platforms, PIL is slightly lower level and simpler, making the language easier to port. The concrete syntax of PIL is derived from Java and therefore familiar to Java developers. PIL/G adds a collection of code generation specific abstractions to the small PIL base language, such as code

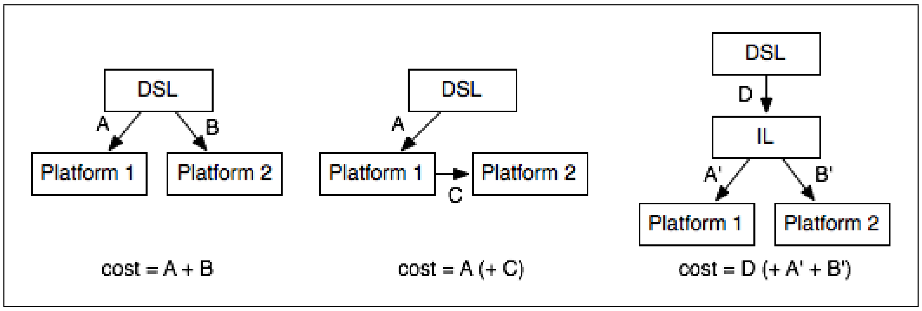


Figure 5.10 The scenarios and costs of transformation options

compositionality features. Due to space constraints this chapter will not discuss the full PIL language, for that we refer the reader to the PIL website[2].

By generating PIL code, rather than e.g. Java or Python code, the cost of targeting multiple platforms is greatly reduced. Any code generation toolset can be used to generate textual PIL code, which is subsequently translated to either Java or Python code by the PIL compiler (Figure 5.11). PIL can also be linked to the generator directly as a library. Currently the PIL compiler library can be linked into Stratego/XT programs, but we are working to enable usage of the PIL library from other tools. The advantage of using PIL as a library is that the overhead of parsing, pretty-printing and I/O can be eliminated by handing an AST to the PIL library rather than a textual representation of the PIL program. While the PIL compiler currently generates Java and Python code, more platforms can be added. Adding a new PIL target platform is cheap.

5.4.1 *PIL: Object-Oriented Programming Essentials*

Instead of providing a high-level platform targeted at *developers*, PIL provides a relatively low-level language with a limited set of easy to port built-in data types. At their core, the platforms many DSL generators target are based on the object-oriented paradigm. PIL captures essential object-oriented features and maps them to their specific incarnations on each platform. The concrete syntax and semantics of PIL are based on Java, because it is well known and statically typed. A dynamically typed intermediate language would complicate the mapping to statically typed languages, whereas mapping a statically typed language to a dynamically typed language is simple.

Since PIL is a language aimed at code generators rather than developers, Java features not useful from a code generation perspective are discarded, thereby reducing the size of the language and lowering the effort of porting the language to new platforms.

From the Java language the following features are omitted:

- Visibility modifiers for classes, fields and methods (e.g. public, private, protected): information hiding features serve no purpose in generated code.
- Interface and abstract classes: can be replaced with classes with dummy implementations of interface methods.
- Inner and anonymous classes: can be implemented as regular classes.
- Imports: are syntactic sugar for the use of fully qualified class names.
- Checked exceptions: are not supported by most other platforms
- Distinction between primitive and object types: in PIL everything is an object. Nevertheless, the Java *implementation* of PIL does use primitive types and boxes and unboxes as required.

[2] `http://www.pil-lang.org`

- Type coercion (e.g. from `int` to `long`): can be made explicit by a code generator.
- Enums: can be implemented using e.g. integers.
- Array syntax, e.g. `byte[] a` and `new byte[] { ... }`. In PIL an array is a regular generic type: `Array<Byte>` and can be instantiated with `new Array<Byte>(...)`.
- The one public class per file restriction. This feature is of no use in the context of code generation.

The Java syntax had to be slightly adjusted to make the language context free. Java's `.` (dot) operator, which is used in package names, static member access, as well as instance member access requires type analysis to disambiguate. In PIL the dot operator is only used for instance member access. In the context of package names, PIL uses `::`. PIL has no static member support. Each language requires at least a minimal set of built-in data types, such as integers, strings, characters, arrays and maps. PIL implementations map each of these types to platform-specific implementations. Platform-specific APIs not part of the built-in data types can be accessed through *external class declarations*, which are further discussed in Section 5.5.

5.4.2 *PIL/G: Compositionality of Code Generation*

PIL is a small, easy to port language, but it lacks some features that greatly simplify code generation. In section 5.2 we discussed that most general purpose programming languages lack compositionality features such as partial classes and methods, which enable concise and modular code generation rules. PIL/G adds such compositionality features to PIL. Through a number of model-to-model transformations the PIL/G model is normalized to regular PIL and then mapped to platform code. In addition to partial classes and methods, PIL/G also adds identifier concatenation and expression blocks.

Partial classes and methods Partial classes and methods enable small code generation rules to emit pieces of code that together define a larger artefact. Figure 5.12 shows two examples of PIL/G code that use partial classes and methods. Normalization of PIL/G to PIL results in a single `SomePage` class containing two fields (`inputCounter` and `pageTitle`) and one `init()` method in which `inputCounter` and `pageTitle` are set. The order in which partial methods' code bodies are concatenated is not defined.

Identifier concatenation A common pattern in transformations is composing two or more identifiers into one. For instance, generating getters and setters for a class property (Figure 5.13) requires repetitive invocation of helper rules to render proper names for the getter and setter method. PIL/G adds a special # identifier concatenation operator to achieve the same result in a more concise manner, as demonstrated in Figure 5.14. The operator adheres to the Java naming convention meaning that the concatenation of `get` and `name` results in `getName`.

```
@partial
class page::SomePage
    extends Page {
  Int inputCounter;
  @partial void init() {
    inputCounter = 0;
  }
}
```

```
@partial
class page::SomePage
    extends Page {
  String pageTitle;
  @partial void init() {
    pageTitle = "welcome";
  }
}
```

Figure 5.12 Partial classes and methods

```
page-farg-to-pil :
  |[ x_prop : srt ]| ->
  |[ t x_prop;
     t x_get() {
       return x_prop;
     }
     void x_set(t value) {
       this.x_prop = value;
     }]|
where x_get := <gen-getter>
                 x_prop
    ; x_set := <gen-setter>
                 x_prop
    ...
```

Figure 5.13 Transformation without identifier concatenation

```
page-farg-to-pil :
  |[ x_prop : srt ]| ->
  |[ t x_prop;
     t get#x_prop() {
       return x_prop;
     }
     void set#x_prop(t value) {
       this.x_prop = value;
     }]|
where ...
```

Figure 5.14 Transformation with identifier concatenation

```
var be : BlogEntry :=
  BlogEntry  blog := b
```
⇓
```
BlogEntry be =
 {| BlogEntry e0 = new BlogEntry();
    e0.setBlog(b);
  | e0 |}
```
⇓
```
BlogEntry be = exprBlock0(b);
...
BlogEntry exprBlock0(Blog b) {
  BlogEntry e0 = new BlogEntry();
  e0.setBlog(b);
  return e0;
}
```

Figure 5.15 Transformation from WebDSL entity constructor expressions to PIL implementation

Expression blocks DSL constructs typically have a higher expressivity than the target platform language that implements them. Therefore, it is common that an expression in the DSL requires multiple statements of implementation code. In WebDSL this problem occurs while transforming entity constructor expressions to PIL expressions. Figure 5.15 shows the transformation steps required to implement a simple example, in which an instance of the `BlogEntry` entity is created, initializing its `blog` property with blog `b`, assigning the result to a new variable `be`. The implementation of this example in PIL requires a variable declaration statement with an initialization expression derived from the entity constructor expression. In order to realize the entity construction, two PIL statements are required: one statement to create an instance of the `BlogEntry` class, and a second to set the `blog` property. Implementing such a transformation is complex, requiring statement lifting. To simplify this type of transformation, PIL/G provides expression block syntax [Bravenboer et al., 2006a]: `{| stat* | returnvalue |}`, as demonstrated in the second transformation step in Figure 5.15. During the PIL/G to PIL normalization, expression blocks are lifted to separate methods, receiving closure variables as arguments.

5.4.3 *Developing PIL Back-Ends*

PIL back-ends can be developed for any language in which it is possible to implement basic OOP features such as objects, classes with single inheritance, virtual method dispatch and garbage collection. Consequently many advanced object-oriented features of the targeted languages remain unused, but this is not an issue as long as the features that PIL requires of a language are a subset of the features offered by the targeted language. Although PIL assumes its target platform to provide garbage collection, it has no assumptions on how this is implemented. Therefore, it is possible to implement a simple garbage collector as part of the back-end transformation. For instance, a language such as Objective-C already provides reference counting using a `retain` and `release` mechanism. The sequence of `retain` and `release`s can be derived from the PIL code based on scopes and data flow analysis. Depending on the targeted platform, implementing new PIL back-ends is relatively cheap. A back-end implementation requires a grammar of the target language specified in SDF and around 1200 lines of Stratego/XT code, much of which can be based on existing back-ends.

5.5 PIL/PLATFORM INTERACTION

PIL has a number of built-in data types such as integers, strings, lists and maps. Any interaction with the platform beyond those is performed through external class interfaces. For example, code generated from WebDSL models accesses web request information provided by the web request API. Similarly, code generated by a parser generator uses IO libraries to read a file to be parsed. For data persistence, generated code often interacts with an object-relational mapper framework such as Hibernate or SQLAlchemy.

Generated *platform-specific* code typically interacts directly with platform-specific APIs. In contrast, when using PIL to target multiple platforms, direct interaction with platform-specific APIs is not an option. The interfaces of APIs of each supported platform need to be wrapped behind a single consistent PIL interface with consistent behavior across platforms. This section discusses three scenarios that demonstrate how platform interaction can be achieved in a platform-independent manner. The section ends with an example of glue

```
external class webdsl::Request {
  webdsl::Session getSession();
  String getParameter(String name);
}

external class webdsl::Response {
  webdsl::util::StringWriter getWriter();
  void redirect(String url);
  void setContentType(String ct);
}
```

Figure 5.16 Web request and response interface in PIL

```
package webdsl;

import javax.servlet.http.*;

public class Response {
  private HttpServletResponse r;
  public Response(HttpServletResponse r) {
    this.r = r;
  }
  public String getParameter(String name) {
    return r.getParameter(name);
  }
  // ...
}
```

Figure 5.17 Part of Java wrapper of Request interface

code that is often required to combine pieces of platform-specific code with generated code in order to build a runnable application.

5.5.1 *API Wrapping*

APIs such as I/O, threading and networking libraries are typically available and similar across platforms. For instance, the API to handle HTTP requests looks slightly different on each platform, but behaves the same. On each platform there is a method to retrieve a GET or POST parameter, get or set a cookie and get access to sessions. Thus, these APIs can be wrapped behind an interface, such as depicted in Figure 5.16 which shows PIL `external class` declarations for a simple Web API. On the Java platform, this interface is implemented using the Java Servlet APIs (Figure 5.17) and on the Python platform it is implemented wrapping its CGI module. The `external class` declaration as seen in Figure 5.16 exposes these classes to PIL code. After the code is generated, the defined wrapper APIs and generated code are combined and compiled by the platform compiler, or interpreted by a platform interpreter.

5.5.2 *Missing API on Some Platforms*

It can occur that a particular API is not available on one or more platforms. In this scenario there are two options. The first is to simply not support the part of the DSL that relies on the particular API on every platform. The second option is to port an implementation of the API to the platforms where it is not already available. The latter can be achieved in two ways, either by porting the API to other platforms directly, or porting the framework to PIL and generating platform implementations from that. Although PIL is intended to be used as a code generation language, it can be used as a language to port an API to as well. The advantage of using PIL over building custom ports for each language, again, is that PIL implementations are portable.

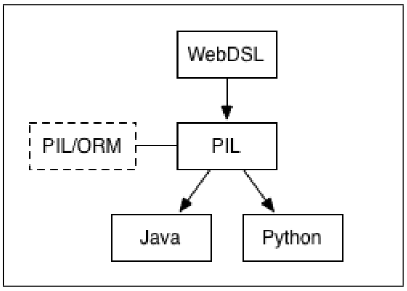


Figure 5.18 ORM implemented in PIL.

Because we could not find a suitable object-relational mapper for Python that is compatible with Hibernate, and because Hibernate did not suit our needs entirely anyway, we implemented a simple ORM framework in PIL (Figure 5.18). Although Hibernate's implementation is substantial, WebDSL only requires a fraction of its features. A significant part of Hibernate's implementation is dedicated to framework *usability*, such as its extensive configuration and annotation support. Such features are of little value when code is generated. Therefore, the ORM library we implemented in PIL, provides only the features and behavior that WebDSL requires. PIL makes the ORM framework very portable, because each platform only requires the wrapping of a low-level database API, enabling the execution of SQL queries (Figure 5.19).

5.5.3 *Semantic Mismatches*

Behavior of platform APIs sometimes differs slightly. In the case of an object-relational mapping framework, for instance, the `persist` operation may have slightly different behavior in one framework than it has in another. Framework *A* may persist the object and all of the objects it references, while framework *B* only persists the object itself. It is sometimes possible to hide these differences in behavior in the API wrapper. In this particular case the wrapper of the framework *B* can traverse the object graph to explicitly persist each object, emulating the behavior of framework *A*.

If changing semantics in the wrapper is not feasible, adapting the framework could be an option. However, this requires a fork of the framework and the maintenance effort that comes with it. Another solution in this case is to reimplement the incompatible frameworks, either in PIL or in a platform-specific manner, as described in Section 5.5.2.

```
external class pil::db::Database {
  new(String hostName, String username, String password,
      String database);
  pil::db::Connection getConnection();
  ...
}

external class pil::db::Connection {
  List<Result> query(String query, Array<Object> args);
  void updateQuery(String query, Array<Object> args);
  ...
}

external class pil::db::Result {
  Int getInt(Int index);
  String getString(Int index);
  Object getObject(Int index);
  ...
}
```

Figure 5.19 Low-level interface to database

```
page-to-register-pil :
  |[ define page x_page(farg*) { elem* } ]| ->
  <emit-pil> |[
    @partial class WebApp {
      @partial void initPages() {
        allPages.put("x_page", page::x_page.class);
      }
    } ]|
```

Figure 5.20 Transformation that emits a partial function registering the page class

5.5.4 *Platform-Specific Glue*

Code generated from a DSL often does not implement the entire application. The canonical example of this are parser generators which only generate parsers that are subsequently invoked from code written specifically for the platform, or code generated from another DSL. Similarly, WebDSL generated code is not invoked directly either, but compiled in conjunction with web application glue code. WebDSL pages are translated to `Page` classes as illustrated in Figure 5.5. Additional code is emitted that registers the page class in a global map during application initialization (Figure 5.20). For each web application a singleton `WebApp` class is generated, whose `initPages` method is extended for each page. Glue code, specific for each platform, instantiates the generated `WebApp` class and retrieves the classes to instantiate based on the request. An example of such glue code for Java is shown in Figure 5.21.

```
public class DispatchServlet extends HttpServlet {
  WebApp webApp = new WebApp();
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response) {
    webApp.initPages();
    Class pc = webApp.allPages.get(Utils.getPageName(request));
    Page page = (Page)pc.newInstance();
    page.renderPage(new webdsl.Request(request),
                    new webdsl.Response(response));
  }
}
```

Figure 5.21 Instantiating PIL-generated Page objects from Java

5.6 DISCUSSION

Applicability PIL is based on the assumption that the target platforms of a DSL are based on an object-oriented language with little dependency on unique platform-specific features. While not the case for every DSL, there are many DSLs, other than WebDSL, for which this is true. Parser generators such as ANTLR and SDF and model transformation languages such as Stratego/XT, ATL or QVT, are examples of these.

Costs of an intermediate language The use of an intermediate language always comes at a price. In the compiler, more transformation steps are required to produce platform code, although this overhead is limited because of the simplicity of the transformation. Flexibility in target platforms is limited by PIL's assumption that target platforms are based on the object-oriented paradigm. Targeting C would therefore be difficult. Platform-specific performance tuning can be implemented by tuning translations from PIL to platform code or by moving performance critical code, code for which efficient implementations depend highly on the platform, to an external API that is called from PIL code. In our implementation of back-ends for WebDSL using PIL, such platform-specific optimizations were not required, however. Another type of overhead occurs when experimenting with new language features. When adding language features that require additions to the compiler back-end, the solution domain is first explored by manually writing platform code. Once the code works, it is generalized and ported to the compiler. However, when PIL is used, platform code cannot be moved to the compiler as-is, it first needs to be translated to PIL. An alternative option is to explore the solution domain by writing PIL code, rather than platform code. Once the PIL code works, it can be ported to the compiler.

5.6.1 *Future Work*

PIL has currently three platform back-ends: Java, Python and PHP5. In the future we intend to add more, such as C#/.NET and Objective-C back-ends. We used PIL to implement back-ends for WebDSL, but in the future we intend to use it to implement back-ends for other DSLs as well. PIL has been developed using Stratego/XT, a DSL for program transformation, which we also want to port to other platforms. Currently there is a C and a partial Java back-end, PIL could greatly simplify maintaining such back-ends. We intend to also investigate if it is feasible to port the SGLR [Visser, 1997a] parser implementation to PIL.

A problem with source code generation as implemented by many DSL compilers is that debugging is very difficult, because the structure and line numbers of the resulting source code are typically very different than the original DSL source. Techniques such as origin tracking [van Deursen et al., 1993] can be used to keep track of position information during transformations. Wu et al. [2008] describe a technique in which position information mappings between source and target code are used by a wrapper around an already existing debugger for the target language. Another approach is by instrumenting generated code that communicates with an external generic debugger. TIDE [van den Brand et al., 2005] takes this approach to simplify the process of defining debuggers for languages built using ASF+SDF. This approach also seems well-suited when implementing debugging support for multiple platforms, since a consistent debugging interface can be implemented for all platforms and no platform-supplied debugging support is required. Using the TIDE approach, code needs to be instrumented with `step` calls that send events to an external debugger with position information and the current environment. We see an opportunity for PIL/G to simplify adding debugging support in this manner, for instance by adding a `step` abstraction to the language that can easily be enabled or disabled.

5.6.2 *Related Work*

Intermediate languages Reusable intermediate languages for the purpose of retargetability are not a new idea. In compiler construction they appeared as early as 1960. UNCOL [Steel, 1961], the Universal Computer Oriented Language, was developed in response to the increasing number of programming languages required to target an increasing number of machine architectures. M languages and N machines require $M * N$ compilers, whereas with an UNCOL only $M + N$ generators and translators are required. Unfortunately, no truly universal UNCOL emerged to handle all languages and machines, likely due to the large size of the instruction set required to generate *efficient* machine code. Other proposed intermediate languages include BCPL's O-code [Richards, 1971], P-code [UCSD, 1981] and `C--` [Peyton Jones et al., 1999]. `C--` is a more recent attempt to simplify machine code generation for multiple machine architectures. Rather than a byte-code representation of the intermediate language, `C--` has a concrete syntax similar to C. Similar to PIL, `C--` is designed as a code generation language. Its focus is much more low-level, however. It addresses a number of problematic areas of C, such as the lack of garbage collection and the difficulty of implementing tail calls. Retargetability is achieved by plugging in one or more machine code generation back-ends, e.g. gcc, VPO [Benitez and Davidson, 1988] or MLRISC [George, 1997]. The mentioned intermediate languages all operate at the abstraction level of machine architecture instructions. PIL by contrast is a much higher level language. It unifies object oriented programming languages and their platforms rather than machine architectures. What PIL adds on top of the mentioned approaches is *code generation specific features* such as partial classes, partial methods, identifier concatenation and expression blocks.

Union and intersection machines [Davidson and Fraser, 1984] represent fictional machines with features roughly equivalent to the *union* or the *intersection* of features offered by typical target machines. They are used as a basis from which intermediate language are derived. Union intermediate languages are good for generating efficient machine code, whereas the small size of intersection intermediate languages, such as PIL, are easier to port.

Basil is a high-level intermediate language with two use cases: (1) a target language for compilers for high-level languages and (2) a language to develop run-time libraries to be used by generated code [Semenzato, 1993]. PIL, too, is targeted at code generation and can be used to develop run-time libraries as well. Basil can translated to C. A subset of Basil, pure Basil, can be translated to LISP. In contrast to PIL, Basil is not an object oriented language and its purpose is not to simplify retargeting compilers. Instead, it is designed to make the semantic gap between source language and machine language smaller. In addition, Basil allows its code to be annotated with position information, which can be used for debugging.

ANTLR Code generation back-ends for the ANTLR [Parr and Quong, 1994] parser generator are defined using StringTemplate, a template engine designed for code generation. For each supported platform (currently over a dozen) a number of code templates define the code to generate to implement ANTLR's features. As of version 3 of ANTLR, each back-end requires around 2800 lines of StringTemplate code. Whenever a new ANTLR version is released, the templates for each templates need to be adapted, resulting in a number of platform back-ends being out of sync with the current ANTLR version.

PIL could reduce ANTLR's maintenance issues. A back-end for the PIL compiler for one platform encompasses around 1100-1300 lines of Stratego/XT code. Note that these back-ends are reusable in multiple DSL compilers as well. A single ANTLR to PIL transformation needs to be defined, presumably also requiring about 2800 lines of code. In addition, custom code needs to be written for each platform wrapping IO APIs. Once PIL back-ends for each of ANTLR's supported platforms are implemented, PIL could reduce the 33,000 lines of code required to implement all ANTLR back-ends significantly.

The Model-Driven Architecture OMG has defined the Model-Driven Architecture [Miller and Mukerji, 2003] in which platform-independent models (PIMs), through an *MDA mapping* are transformed to platform-specific models (PSMs). Whether this mapping should be performed manually or automatically is not specified. In our approach the WebDSL model is a PIM and the platform-specific code that is generated from that are PSMs. PIL acts as a thin layer in-between PIM and PSMs. The separation of platform-independent models from platform-specific models is essential when targeting multiple platforms. For instance, DSLs that contain escapes to the underlying platform are not platform-independent and can therefore not realistically be ported to other platforms. The fact that we implemented WebDSL back-ends for two different platforms (Java and Python), proves that WebDSL models are indeed platform-independent.

Bezivin et al. [2004] demonstrate how MDA can be used to automatically derive multiple PSMs from one PIM. They generate Java, web services and JWSDP from UML and EDOC PIM models. Muller et al. [2003] apply the MDA approach to generate web applications from visual UML-based models. Their code generator is written in Java and can generate either Java or PHP code that communicates with either an Oracle, MySQL or PostgreSQL database. The development of WebDSL so far has not focussed on supporting multiple database systems, but this is future research. We intend to investigate supporting not only SQL databases, but also alternative types of databases such as Google's BigTable [Chang et al., 2008]. Although PIL itself does not solve the problem of targeting different types of database systems, it can make the code to use these database systems portable across software platforms.

5.6.3 *Conclusion*

In this chapter we explored a number of approaches to construct DSL compilers targeting multiple software platforms. The ability to retarget DSLs is enabled by the strict separation between platform-*independent* models and platform-*specific* models. It is common practice to maintain separate compiler back-ends for each targeted platform. However, the maintenance of these back-ends is costly because DSL to target platform transformations are expensive to build and maintain. We argued that high-level intermediate languages can improve the retargetability of DSLs by only having to define one transformation from the DSL to the intermediate language. Subsequent mappings from the intermediate language to target platforms are cheap to develop and reusable in multiple DSL compilers.

We presented PIL, a Platform Independent Language, as an implementation of such a high-level intermediate language. PIL is based on a subset of Java, and is therefore a more familiar and easy to target language than traditional low-level intermediate languages as commonly used in compiler construction. PIL/G, a collection of abstractions built on PIL, adds features simplifying the use of PIL as a code generation language, such as partial classes and methods, identifier concatenation and expression blocks.

We validated our approach by implementing a PIL back-end generating Java and Python code for the WebDSL compiler. Previously, these platforms were supported through separate back-ends which led to large-scale code duplication. By using PIL, only one DSL to PIL transformation needs to be maintained, as well as small platform-specific API wrappers for database and HTTP request access.

Postscript: PIL

It is common practice to develop a DSL as a thin syntactic layer on top of an existing framework [Stahl et al., 2006, van Deursen et al., 2000b]. This is how WebDSL was developed initially, by building it on top of the JBoss Seam framework. However, as development of the DSL and underlying framework start to diverge, mapping the DSL onto the framework becomes increasingly difficult [Groenewegen et al., 2008]. Therefore, as discussed in Section 1.8, we redeveloped the WebDSL back-end to generate much lower-level code. At this level of abstraction, the difference between platforms is largely cosmetic, enabling the use of an intermediate language such as PIL.

As described, in the implementation of the Java and Python WebDSL back-ends, one library was not replaced: the object-relational mapper (Hibernate, in the case of the Java back-end). Hibernate is a complex piece of software that was offered enough flexibility for WebDSL. However, when adapting WebDSL to generate code through PIL, a solution had to be found for the ORM part, which eventually we solved by implementing an ORM in PIL (as described in Section 5.5.2). However, *reimplementing* a substantial library just to enable retargetability of the DSL is definitely a trade off to be evaluated closely.

PIL AND WEBDSL

In the case of WebDSL, the implementation of the PIL back-end for WebDSL was never developed beyond a prototype. There were two reasons:

The first was the manual porting of libraries. We built a simple ORM library, but it did not yet support all the features required, such a complex queries. The effort to get it to that level was considerable. We also considered adding other new features to WebDSL, such as search. For Java there is the excellent Lucene[3] library, but there is no such library available for e.g. Python or PHP. Would the search feature simply not work on platforms other than Java, or would we have to port Lucene to PIL as well?

The second reason was that when development of PIL started, we forked the WebDSL Java back-end and slowly ported it to PIL. Subsequently, the PIL and the WebDSL PIL back-end were developed in tandem. While we developed PIL, colleagues kept developing the Java back-end, resulting in the PIL back-end being completely out of sync with Java back-end by the time we got basic WebDSL applications to compile using PIL. The Java back-end had advanced a lot since the time the PIL back-end was branched (almost half a year earlier).

We had a group meeting to discuss the future of the PIL back-end, trying to decide if there was sufficient reason to move our development effort from the

[3]http://lucene.apache.org

Java back-end and focus all of our effort on a new, up-to-date PIL back-end, that would likely take a couple of months of full-time work to complete.

We decided to approach the problem from a commercial point of view. Would the considerable effort pay off? Would WebDSL find more use if it could also generate Python and PHP code? We concluded this was not really the case. If a potential user was willing to invest in WebDSL, he would accept having to run it in a Java environment. It is important to note that from the end-user perspective, WebDSL applications are very portable. They are accessible from any browser on any platform, it is only the *server-side* where portability is an issue. We decided that adding a PHP and Python back-end would not significantly increase our potential user base in the case of WebDSL. Therefore, the effort it would take to port the current Java back-end to PIL would not be worth it. We decided to discontinue the development of the PIL back-end for WebDSL.

PIL AND MOBILE

In the mobile domain the portability problem was very apparent. In order to develop a mobile application for multiple platforms one has to effectively implement it completely separately for every platform — there was very little overlap in the technology stacks used on the different software platforms. Ostensibly, this seemed an ideal application of PIL: a mobile DSL that would generate PIL, which in turn would be mapped to Objective-C for iPhone and Java for Android.

PIL works well for many application aspects including application logic and even data persistence (we had already built a PIL ORM library for WebDSL that could be reused). However, there were two issues that let us decide not to use PIL for a mobile DSL.

The first was user interface APIs. Apple's iOS platform has a very different API to create user interfaces than Android. Wrapping the existing UI libraries for each platforms as a consistent PIL library would be a significant undertaking. The goal of PIL is to reduce the amount of maintenance work to be done for each platform back-end. However, the majority of maintenance was likely to take place in user interface aspect of the language, still requiring those changes to be made in the user interface library for each platform individually, thereby reducing the advantage of using PIL significantly.

The second issue was Apple's stringent rules about *how* iPhone applications were to be built. One of the newly instated rules was that applications were to be "originally written in Objective-C, C++, C or Javascript", thereby effectively forbidding applications to be *generated* from a DSL. In addition, Apple refused to sell certain kinds of applications in its AppStore. In response, Google started to develop web-based versions of some of its popular applications, including Google Voice (an application that had been rejected by Apple because it "duplicated phone functionality") and Gmail.

As a result, we decided to follow the path that Google chose and revised our original plan: our mobile DSL, called mobl (Chapter 6), would generate mobile *web* applications rather than native applications.

APPLICATION AREAS

While we did not end up using PIL for our mobile DSL, we are convinced PIL still has applications. The reoccuring problem that led us not to use PIL for either WebDSL or mobl was reusing existing libraries that were too large to port to PIL or too hard to wrap consistently. However, there is a class of DSLs that does not heavily rely on existing libraries, for instance parser generators, which only need to read in a string and produce a parse tree or AST data structure; or state machines, which receive simple signals and produce simple events to communicate with the outside world.

Declaratively Programming the Mobile Web with Mobl

6

ABSTRACT

A new generation of mobile touch devices, such as the iPhone, iPad and Android devices, are equipped with powerful, modern browsers. However, regular websites are not optimized for the specific features and constraints of these devices, such as limited screen estate, unreliable Internet access, touch-based interaction patterns, and features such as GPS. While recent advances in web technology enable web developers to build web applications that take advantage of the unique properties of mobile devices, developing such applications exposes a number of problems, specifically: developers are required to use many loosely coupled languages with limited tool support and application code is often verbose and imperative. We introduce *mobl*, a new language designed to declaratively construct mobile web applications. Mobl integrates languages for user interface design, styling, data modeling, querying and application logic into a single, unified language that is flexible, expressive, enables early detection of errors, and has good IDE support.

6.1 INTRODUCTION

With the rapid growth in sales of modern smart phones and tablets, such as iPhone, iPad, Android and BlackBerries, the web becomes available on an increasing number of powerful mobile devices equipped with modern web browsers. However, today's websites are optimized for personal computer browsers and environments, whereas mobile devices are used in different contexts, and have different features and constraints than personal computers, for instance:

- Internet access is not always available, reliable or fast;
- Screen estate is limited;
- Expected user interaction patterns are different, such as touch controls and gestures such as tapping, swiping and pinching;
- Applications are expected to respond to changes in context, such as holding the device in portrait or landscape mode, or changes in location.

Consequently, hundreds of thousands of custom *native* mobile applications are being developed. Examples include communication applications (e-mail, messaging), content viewers (books, articles, papers, RSS feeds, video, photos, audio) and location-based services (wikihood, foursquare, loopt). While these applications run locally on the device itself, a large class of these applications are *data-driven applications* that communicate with one or more web services to exchange data.

While iOS, Android, BlackBerry, WebOS, Windows Phone 7 and other platforms are similar in terms of interaction, features and restrictions, their development environments are quite different. iPhone and iPad applications are developed using the Objective-C language; Android and BlackBerry applications are built using Java, but using very different APIs; WebOS applications use HTML, CSS and JavaScript; Windows Phone 7 development is done using .NET. Developing software that is *portable* to multiple platforms is difficult. In addition, *deployment* is non-trivial; most platforms come with an application marketplace, some of which require manual testing of submitted applications by the marketplace provider before being published — a process that can take many weeks — and applications can be rejected for seemingly arbitrary reasons.

At the end of the 1990s, mobile phones started to gain access to the Internet through WAP (Wireless Application Protocol). The development model for WAP applications was very similar to the development of regular web applications. Rather than sending HTML, a server would send WML (Wireless Markup Language) to the mobile phone. With the release of the original iPhone in 2007, a new generation of smart phones and tablets started to be released with more powerful browsers that support all modern web technologies. At the same time, advancements in HTML (HTML 5) and CSS (CSS 3) started to enable the creation of web applications that offer a comparable experience to native applications, especially for *data-driven applications*, by supporting application and data caching, detection of touch gestures and access to geographical position information (GPS). The portability and deployment advantages of web applications make the use of web technologies for building mobile applications very attractive.

Similar to *native* applications, mobile web applications can now be developed that run completely disconnected from the server, requiring a different development model than regular web applications. When a mobile web application is first launched through the web browser, its application code is cached on the device. The application can use local SQL databases to cache data obtained from a server for offline use. When no Internet connection is available, the mobile browser retrieves the application from its cache and continues to operate. All application logic, written in JavaScript, resides on the device rather than on the server as is the case in regular web applications. Communication with the server, similar to native applications, happens by performing web service calls using AJAX (Asynchronous JavaScript and XML). At the time of writing, HTML5 is well supported by the iPhone, iPad, Android, WebOS and BlackBerry (6+) platforms.

While HTML5 makes it *possible* to develop offline-capable mobile web applications that are portable and easy to deploy, development of such applications exposes a number of problems.

First of all, web development does not enforce a particular application architecture; application concerns (such as data modeling, user interface and application logic) can be mixed arbitrarily – an approach that does not scale well. Therefore, a *structured architecture* is required to develop mobile web applications. A common architectural style in organizing user-facing software is the Model-View-Controller (MVC) pattern [Gamma et al., 1995]. The MVC pattern separates the Model (data, e.g. in a database) strictly from the View (the user interface) by making the Controller responsible for communication between the two. While the separation of View and Model is good, the MVC pattern results in *boiler plate code* that needs to be written to glue the application together.

Second, mobile web applications are built by mixing a number of *loosely coupled languages* including HTML, CSS, JavaScript, SQL and a cache manifest. While the use of domain-specific languages in web development support a declarative programming model, they are not very well *integrated*. In Chapter 2 we studied the current state of server-side web frameworks which, similar to mobile web development, take advantage of multiple loosely-coupled languages. The consequence of this design is the same both in mobile and regular web development: lack of static analyses detecting inconsistencies results in *late detection* of failures. In addition, developers have come to expect excellent *IDE support* for their languages, including in-line error highlighting, reference resolving, outlines, code completion and refactoring support. The dynamic nature and loose coupling of the web languages complicates the construction of IDE support.

Third, web languages such as HTML and CSS do not support basic *abstraction* mechanisms, complicating the reuse of user interface elements. As a result, HTML and CSS artifacts contain a lot of code duplication.

Fourth, JavaScript in the browser is a single-threaded environment, forcing developers to use asynchronous APIs for performing expensive computations, including database queries and obtaining GPS coordinates. These asynchronous APIs require the developer to write code in the unnatural *continuation-passing style*, one example of *accidental complexity* in mobile web development.

In this chapter, we introduce *mobl*[1], a high-level, declarative language for programming mobile web applications, which addresses these problems. Mobl is our second case study in the design and implementation of *syntactically integrated DSLs*, DSLs that integrate sub-languages for multiple application aspects, enabling static verification of the entire application. Previously, we developed WebDSL, a DSL to develop data-driven web applications. While covering a different domain, many ideas from WebDSL are reused in the design of mobl. Mobl integrates languages for user interface design, styling, data modeling, query and application logic into a single, unified language. The language is *high-level* since it avoids accidental complexity such as continuation passing style and supports the definition of reusable user interface elements. The language is *declarative* since it ensures automatic updates of

[1]`http://www.mobl-lang.org`

the user interface through reactive programming and automatic persistence of data in the client-side database.

Mobl implements the *Model-View* (MV) pattern, a variant of Model-View-Controller where the role of Controller has been *automated*, data model-related logic has been moved to the Model and user interface logic has been moved to the View. The MV pattern reduces the amount of *boilerplate code* that needs to be written compared to MVC.

The integration of the various concerns of mobile web programming into a single language, enables consistency checking across concern boundaries, ensuring early detection of many common errors by the mobl IDE (integrated into Eclipse), which provides in-line error reporting, code completion and reference resolving. The mobl compiler compiles mobl code into a pure client-side web application, implemented using a combination of HTML, CSS, JavaScript and application caching manifests. Mobl applications can be deployed to any web server and are server-technology agnostic.

6.1.1 *Contributions*

The contributions of this chapter are as follows:

- An analysis of the mobile web application domain and how it it differs from the regular web.
- The mobl language, a new language composing language features such as transparent data persistence, declarative programming, reactive programming and the continuation-passing style transformation into an integrated, verifiable language geared towards the mobile domain.
- A demonstration how mobl's core abstractions can be used to build high-level abstractions, such as tab sets and master-detail controls as mobl libraries.

6.1.2 *Outline*

The rest of this chapter is organized as follows: Section 6.2 analyzes the mobile domain and its problems. Section 6.3 describes the general architecture and design principles of mobl. Subsequent sections discuss the various aspects of mobile applications and how mobl supports them: data modeling (Section 6.4), user interfaces (Section 6.5), navigation (Section 6.6), higher-order controls (Section 6.7) and styling (Section 6.8). Section 6.9 discusses related work and Section 6.10 concludes.

6.2 MOBILE WEB APPLICATIONS

The design of a new language for mobile web application development requires a thorough understanding of the mobile domain. This section discusses the architecture of traditional web applications and compares it to the architecture of *mobile* web applications. Subsequently, we identify a number of problems in the *development* of mobile web applications.

6.2.1 *Technical Architecture*

The traditional style of web applications, sometimes referred to as RESTful web applications [Richardson and Ruby, 2007], are request-oriented. Objects on the server have the life span of a single request, and are recreated as needed on every request. Since making HTTP requests is relatively expensive, they are used sparingly, when navigating to a new page, submitting a form or performing an AJAX (Asynchronous Javascript and XML) call. The web application server responds to requests from the client (browser). When a request comes in, it is handled by a server written using, for instance, Java, .NET, PHP or Ruby. The server communicates with a database to retrieve or manipulate data, and eventually sends back HTML to the browser which renders it on the user's screen. A server handles multiple users and typically stores data for all its users in a shared database. HTTP requests can also be sent by JavaScript code on the web page, using AJAX calls. Based on the result of such a request, the JavaScript may manipulate the HTML DOM (Document Object Model) to make changes to the user interface without requiring an entire page reload. In addition to performing AJAX calls, JavaScript is used for client-side validation of user input in forms.

There are multiple approaches to developing *mobile* web applications. For older, non-smart phones, processing power is the main bottleneck. Therefore, several thin-client approaches exist [Lai et al., 2004, Kim et al., 2006] where all processing happens on the server and phones are served with pre-rendered pages. However, today's modern smart phones have more powerful processors, thus client-side processing is no longer a bottleneck. Therefore, for these devices applications can be developed in a range of styles. On one end of the scale are web applications that are built similarly to regular web applications, except reducing the amount of data presented on a single page, to fit the screen size of the mobile device. It is relatively easy to adjust a regular web application to produce pages that are more friendly to the smaller screen size of a mobile device. A drawback of this approach is that such applications are not available without an Internet connection. In addition, Internet speeds on mobile devices are on average a lot slower than on PCs, resulting in a bad user experience.

At the other end of the spectrum are *offline-capable* mobile web applications that, once accessed by the mobile browser, are cached locally. They may fetch data from the server and cache it in a local database on the device as well. The development model of this type of application is very similar to desktop applications and native mobile applications and merely use web technologies as an implementation means. All the application logic, written in JavaScript, executes at the client, in the device's browser. This enables much more responsive user interfaces, because a "click" no longer requires a HTTP request be sent to the server. Events can therefore be processed much more granu-

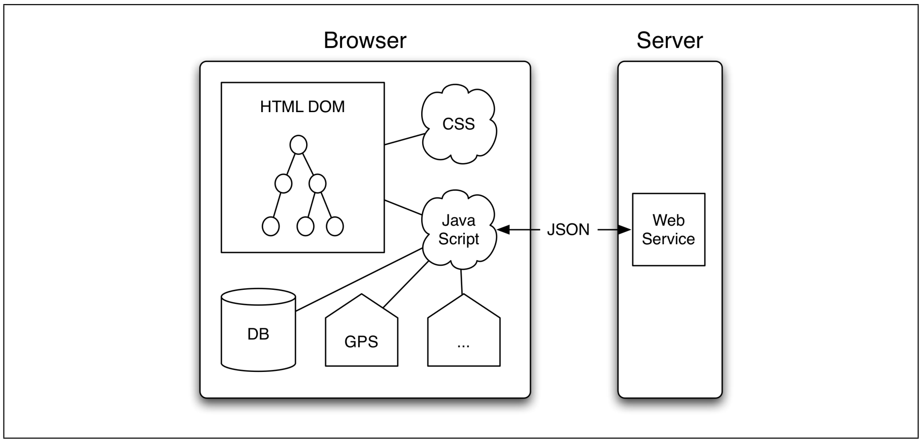


Figure 6.1 Mobile web application technical architecture

larly than in RESTful-style web applications, and can respond immediately to gestures and key presses. Like desktop applications, mobile web applications are single-user applications that do not require user authentication and access control. This type of application can be used without an Internet connection, after the application and its data is loaded and cached locally. Internet latency on mobile networks is also less problematic because fewer requests have to be sent to the server.

Figure 6.1 shows the technical architecture of offline-capable mobile web application. The user interface is defined using HTML (HyperText Markup Language) and styled using CSS (Cascading StyleSheets). The runtime representation of the user interface is the Document Object Model (DOM), which can be manipulated at runtime using JavaScript. JavaScript acts as a glue language, manipulating the DOM, calling web services and executing database queries. The application's data is stored in a SQLite database running locally on the device. The database is accessed through an asynchronous JavaScript API that supports the execution of SQL queries. All application resources (such as HTML, CSS, JavaScript and images) are cached locally on the device using the HTML5 Application Cache. When an Internet connection is available, the application can request data from, and push data to the server.

6.2.2 *Architectural Patterns*

There is no particular application architecture enforced in web development. HTML, JavaScript and CSS can be mixed arbitrarily. While lowering the barrier to entry, this unstructured web application development does not scale well for larger applications. Therefore, a number of architectural patterns have been developed for user facing applications. The most commonly used is the Model-View-Controller (MVC) [Gamma et al., 1995] pattern, but alternatives include Model-View-Presenter [Potel, 1996] and Model-View-ViewModel [Smith, 2009].

The Model-View-Controller pattern creates a strict separation between three layers of the application:

1. The **Model** represents the data to be manipulated by the application, e.g. persistent data objects.
2. The **View** defines a user interface, presenting (elements of) the Model.
3. The **Controller** responds to user events and adapts the View and Model accordingly.

While developing WebDSL [Visser, 2007b], we studied MVC web frameworks that are commonly used in web development. We observed that the Controller is required to perform a mostly infrastructural role. It is responsible for reading user input, applying requested changed to the Model, and manipulating the View. It impedes the rapid development of applications: minor changes, such as a new property in the Model that has to be editable from the View, requires not only the adaptation of the Model and View, but the Controller as well. Consequently, the use of the Model-View-Controller, as well as similar patterns, result in a lot of *boiler plate code* that needs to be written.

6.2.3 *No Integration*

In Chapter 2 work we surveyed the state of practice in web development. We observed that web frameworks typically rely on a number of *loosely-coupled languages*, e.g. Java, XML configuration files, SQL, HTML, CSS and Javascript. Due to their loose coupling, these framework typically lack tools that can statically verify applications to detect inconsistencies between components of the applications defined using different languages, such as HTML pages that link to non-existing Java controllers, or HTML elements that reference non-existing CSS styles. As a result, errors materialize as runtime faults with obscure error messages that are hard to trace back to their origin.

Mobile web development suffers from the same problem. It too relies on the use of multiple languages, such as HTML for creating user interfaces, CSS for styling, JavaScript for application logic, SQL for database querying and caching manifests for application caching. In addition, since all web languages are dynamically typed, accurate implementation of typical IDE features such as code completion and reference resolving has become challenging. Consequently, tool support for mobile web development is sub-optimal.

6.2.4 *No Abstraction*

HTML was architected to define the structure of an entire web page. It does not support the definition of reusable HTML templates, or means to invoke a template. Similarly, CSS's support for abstraction is also limited. Using CSS classes, styles can reused by attaching them to multiple HTML tags, but no parameterization of these styles is supported to vary colors slightly, for instance. SQL does not support abstraction either. A SQL query can only be expressed as a whole, not in reusable parts. Although it is possible to iteratively construct a query by concatenating strings, this is very error prone.

6.2.5 *Accidental Complexity*

JavaScript in the browser runs on a single thread that is shared with the page renderer. Therefore, JavaScript calls that take a long time to complete can freeze the browser. As Javascript does not allow developers to create threads, many JavaScript APIs are defined as *asynchronous* APIs. Asynchronous computations are computed on a separate thread (managed by the browser), and call back to the Javascript thread when the computation completes. While synchronous calls return the result of their computation as a return value, asynchronous methods are passed a *callback function* (or *continuation*), which is called with the result when the computation has finished. This style of programming is called *continuation passing style*.

Asynchronous APIs have favorable performance characteristics, because they do not block the user-interface thread. Nevertheless continuation-passing style leads to verbose, difficult to read and maintain code. Effectively, developers have to adapt their programming style as a result of a low-level performance-related issue.

6.3 MOBL ARCHITECTURE

We have developed mobl. Mobl is a new statically typed, domain-specific language designed specifically for the rapid development of data-driven mobile web applications.

Mobl *linguistically integrates* all aspects of mobile application development into a single, statically verifiable language. It enables *separation of concerns* by supporting the separation of user interface and data model. It applies *domain abstraction* to abstract from accidental complexity and irrelevant details of the platform/domain. It supports *user-defined abstractions* by enabling users to define reusable screens, controls and styles.

This section discusses the high-level aspects of the language and application architecture. Subsequent sections give detailed descriptions of the sublanguages that mobl comprises.

6.3.1 *Integration and Tooling*

HTML, JavaScript and CSS contain numerous cross-references. For instance, CSS selectors rely on the structure of the HTML page and JavaScript is used to manipulate the HTML DOM at run-time, e.g. by attaching or removing CSS classes. Verifying that these cross references are correct, e.g. that a CSS selector matches the right HTML tag and JavaScript attempts to manipulate an existing DOM node, is typically done by *running* the program, resulting in *late failure*. In addition, loose coupling of web languages makes implementing

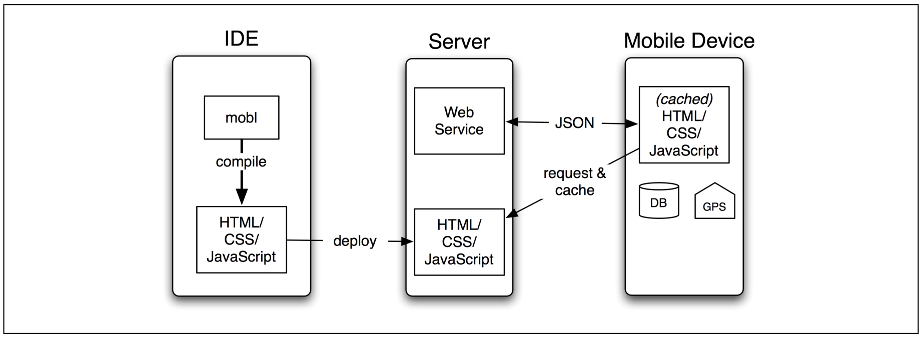


Figure 6.2 Mobl application compilation and deployment

accurate IDE support difficult. Therefore, web development IDEs are not at the level of languages such as Java and C#.

By contrast, mobl integrates the aspects of mobile applications into a single, *integrated language*, rather than using several loosely-coupled languages. Mobl consists of a number of integrated sub-languages for the definition of data models, queries, user interfaces, styles and application logic. Language elements are shared across the sub-languages. For instance, the expression language used in application logic is reused in user interfaces, resulting in a consistent language. This *linguistic integration*, previously also applied in the implementation of WebDSL (Chapter 2), enables accurate end-to-end static verification of applications, e.g. verifying that controls are invoked correctly, invoked screens exist, the properties of data objects presented in the user interface exist and are of the correct type, and queries filter based on existing properties.

The mobl *compiler* compiles a mobl module to a combination of HTML, JavaScript and CSS. As Figure 6.2 shows, the resulting compiled files can be deployed to a web server along with any web services that the application may use. A mobile device requests the HTML file, automatically fetching the CSS and JavaScript resources. All application resources are cached in the browser's HTML5 application cache, allowing the application to be launched even when no Internet connection is available. The application runs on the device and has access to a local database, as well as other APIs including Geo Location. The application may call a web service on the server pulling or pushing data, presenting that data and optionally caching it locally in the database.

The mobl IDE is implemented as an Eclipse plug-in using Spoofax [Kats and Visser, 2010b]. As Figure 6.4 illustrates, it offers an outline view, in-line highlighting of verification errors, reference resolving and code completion. The mobl compiler is integrated into the IDE, and triggered on every save of a mobl module. There is also a stand-alone compiler available.

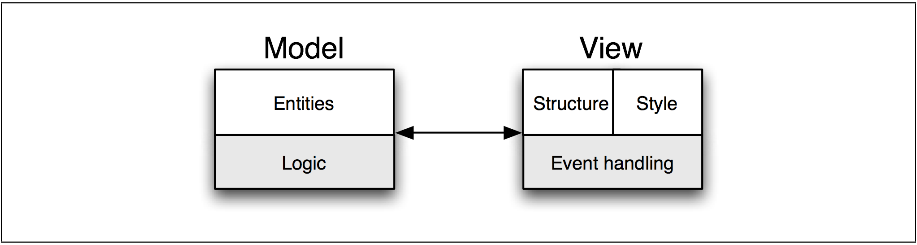
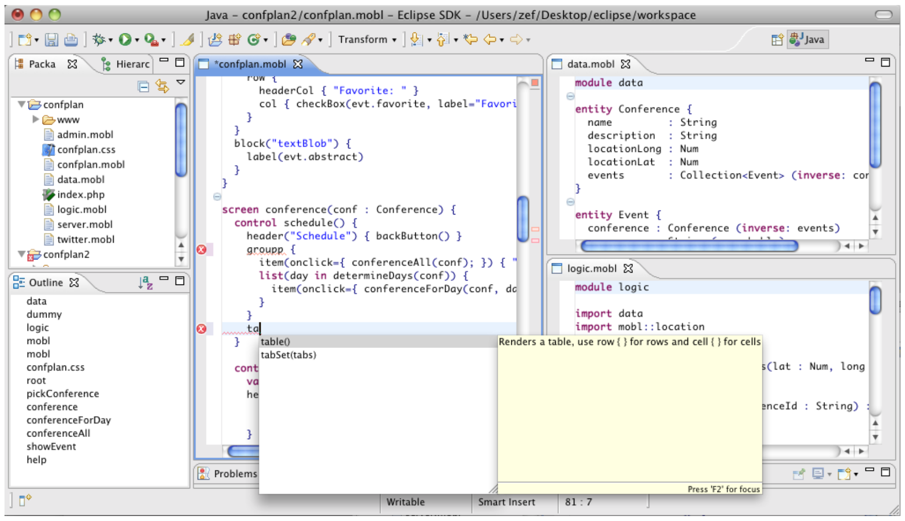


Figure 6.3 Model-View pattern



```
      row {
        headerCol { "Favorite: " }
        col { checkBox(evt.favorite, label="Favori
      }
    }
    block("textBlob") {
      label(evt.abstract)
    }
  }

  screen conference(conf : Conference) {
    control schedule() {
      header("Schedule") { backButton() }
      groupp {
        item(onclick={ conferenceAll(conf); }) { "
        list(day in determineDays(conf)) {
          item(onclick={ conferenceForDay(conf, da
        }
      }
      ta
    }

    cont
      va
      he

      }
```

```
table()
tabSet(tabs)
```

Renders a table, use row { } for rows and cell { } for cells

Press 'F2' for focus

```
module data

entity Conference {
  name         : String
  description  : String
  locationLong : Num
  locationLat  : Num
  events       : Collection<Event> (inverse: cor
}

entity Event {
  conference : Conference (inverse: events)
```

```
module logic

import data
import mobl::location

s(lat : Num, long

enceId : String) :
```

Figure 6.4 The mobl Eclipse IDE

6.3.2 *Model-View Pattern*

While the Model-View-Controller pattern is a good organizational tool, it also requires a considerable amount of *boiler plate* code to set up and to achieve simple tasks. This boiler plate is largely caused by the Controller. In a typical application, the Controller has the following responsibilities:

- Read data from the Model and send it to the View;
- Manipulate the Model based on forms defined in the View (user input);
- Persist changes in the Model to database;
- Activate and deactivate (parts of) Views;
- Communicate with external data sources, e.g. web services.

While the core of the application is encoded in the Model and the View, a lots of plumbing code is required in the Controller, while most of the Controller's tasks are very common and infrastructural in nature. Therefore, mobl implements the Model-View (Figure 6.3) architecture. The MV architecture is an adaptation of the MVC pattern, *automating* the tasks of the Controller rather than letting the developer encode them manually.

In the MVC architecture the Controller is responsible for instantiating Views and populating them with data. In contrast, in the MV architecture Views are the initiators. *Views* can be parameterized with one or more Model objects to present, or they can send a request to the Model themselves to retrieve data. Views are also responsible for handling user input events, such as button clicks and responding to them, e.g. navigating to another View or calling a method on the Model. The *Model* is automatically persisted to a database, no explicit save operations are required. In addition, the Model communicates with web services to synchronize and cache data. *Data binding* establishes a direct connection between the Model and View, eliminating the need to manually copying data from the Model to the View and vice versa.

6.4 DATA MODEL

The implementation of an application's data model, as well as the manipulation of data at run-time, is cumbersome in mobile web development because of the lack of *domain abstraction*. This section details the underlying issues and shows how mobl raises the level of abstraction by using declarative data models, its imperative language and integrated query language. It concludes with an example detailing the implementation of a data model for a simple task manager application.

HTML5 DATA PERSISTENCE Part of HTML5 is the Web SQL API[2], enabling the creation of local (SQLite) databases in a mobile device's browser. The amount of space available to a database varies from the device to device, but is typically around 5 megabytes. Therefore, the local database is perfectly suited to store small amounts of data and cache data from remote resources.

Since the HTML5 database APIs are new, libraries and frameworks built around them are still limited. Therefore, communicating with an HTML5 database is still done at the level of *low-level SQL statements*, which is not only inconvenient for developers, but also more prone to security problems such as SQL injection attacks. In addition, encoding queries in strings is error prone because developers do not get the support from the IDE that they do get for the rest of the language, including syntax checking, semantic checking and code completion.

SQL queries do not compose well. It is difficult to pass a partial query to a different part of the application where it can be extended, e.g by adding an additional filter condition. Therefore, reuse of queries is limited to what can easily be achieved using string concatenation.

SEARCH Most mobile web applications require simple full-text search functionality, allowing users to quickly search through local data. HTML5 does not offer direct support for this. Therefore, custom solutions need to be built.

[2] `http://www.w3.org/TR/webdatabase/`

Logic Database and web service related Javascript APIs are exposed as asynchronous APIs. This requires the developer to write code in a *continuation-passing style*. For instance, consider the following code written using (hypothetical) *synchronous* JavaScript APIs:

```
var tasksJSON = httpRequest("/export");
tx.executeQuery("INSERT INTO Task ...");
alert("Done!");
```

Javascript's asynchronous APIs, rather than returning the result as the result of a function, are passed a *callback function* and return immediately. The actual execution occurs on a separate thread, managed by the browser. When the computation finishes, the callback function is invoked with the result. Therefore, the above code using asynchronous APIs has to be rewritten as follows:

```
function completed() {
  alert("Done!");
}
function receiveTasks(tasksJSON) {
  tx.executeQuery("INSERT INTO Task ...",
                  completed);
}
var tasksJSON = httpRequest("/export",
                  receiveTasks);
```

As can be observed, the code in continuation-passing style is written in an inverted order. While this asynchronous code leads to more responsive applications in the browser, it impedes developer productivity. It is a typical example of *accidental complexity*.

6.4.1 *Data*

Mobl contains *domain abstractions* for declaratively defining persistent data structures (`entity` definitions), abstracting from the underlying SQL database that implements them. Persistence of data is handled by the mobl runtime transparently.

The syntax of data models is detailed in Figure 6.5. Data model declarations consist of zero or more `entity` definitions. Every entity has a name, zero or more *properties* and associated *functions* expressing application logic related to the entity. Each property has a name, type and optionally one or more annotations. Its type can be of a scalar type (e.g. `String`, `Num`, `DateTime` or `Bool`) as well as `Collection`s of other entities.

A `Collection` represents a (virtual) collection of entity instances that can be filtered, sorted, paged and manipulated. `Collection`s are used to represent one-to-many and many-to-many relationships in models, but also to query persistent data. In addition, the `Collection` abstraction is used for full-text search. The `(searchable)` annotations on textual properties indicate that the property should be included when performing full-text searches on instances of this entity. These searches are performed through a `EntityName.`

```
Def ::= "entity" ID "{" EBD* "}"

EBD ::= ID ":" Type ("(" {Anno ","}* ")")?
     | "static"? "function" ID
        "(" {FArg ","}* ")" ":" Type
        "{" Stat* "}"

Type ::= ID
       | "Collection" "<" Type ">"
       | "[" Type "]"
       | "(" {Type ","}* ")"

Anno ::= "inverse:" ID
       | "searchable"

FArg ::= ID ":" Type
       | ID ":" Type "=" Exp
```

Figure 6.5 Data model syntax

`search(phrase)` call, which returns a `Collection` object representing the search results. As with any `Collection`, the results can subsequently be filtered and paged.

An `(inverse: property)` annotation on a property defines `property` as the inverse property of this one. Properties declared as each other's inverse keep each other in sync and are used to declare one-to-one, one-to-many and many-to-many relationships.

Outside data models, mobl also supports variables of other collection types, including arrays and tuples. Arrays are declared using the `[Type]` notation and tuple types using `(T1, T2, T3)` syntax.

Implementation We have developed a JavaScript object-relational mapper (ORM) [Barry and Stanienda, 1998] library called persistence.js[3] to handle data persistence mobl. The library implements transparent data persistence, querying and search. Data models defined in mobl are translated to calls to persistence.js by the mobl compiler. A full-text search index (implementing a simple stemming algorithm [Lovins, 1968]) is automatically maintained by the ORM library.

6.4.2 *Logic*

Mobl's imperative object-oriented sub-language enables programming in the natural, *synchronous* style, abstracting from the accidental complexity of the asynchronous programming style enforced by HTML5 JavaScript APIs.

Imperative code is written using a JavaScript-like [ECMA, 2009] syntax. The language supports variable declarations, assignments, if-statements, for-each and while loops, function and method calls, and various arithmetic expres-

[3]http://persistencejs.org

```
Stat ::= "var" ID "=" Exp ";"
       | LVal "=" Exp ";"
       | Exp ";"
       | "if" "(" Exp ")" Stat ("else" Stat)?
       | "foreach" "(" LVal "in" Exp ")"
           "{" Stat* "}"
       | "while" "(" Exp ")" "{" Stat* "}"
           "{" Stat* "}"
       | "return" Exp? ";"
       | "screen" "return" Exp? ";"

LVal ::= ID
       | Exp "." ID
       | "(" LVal "," {LVal ","}* ")"

NamedExp ::= Exp
           | ID "=" Exp

Exp ::= STRING | NUMBER | ID | "true"
      | "false" | "null" | "this" | "!" Exp
      | "(" Exp ")" | "[" {Exp ","}* "]"
      | "(" Exp "," {Exp ","}* ")"
      | ID "(" {NamedExp ","}* ")"
      | Exp "." ID "(" {NamedExp ","}* ")"
      | Exp "." ID | Exp Op Exp
      | Exp "?" Exp ":" Exp | "{" Stat* "}"

Op ::= "||" | "&&" | "==" | "!=" | "<"
     | "<=" | ">"  | ">=" | "*"  | "/"
     | "%"  | "+"  | "-"  | "++" | "--"
```

Figure 6.6 Imperative language syntax

```
Exp    ::= Exp Filter+

Filter ::= "where" SetExp
         | "order" "by" OrderExp
         | "limit" Exp
         | "offset" Exp

OrderExp ::= ID | ID "asc" | ID "desc"

SetExp ::= ID "==" Exp | ID "!=" Exp
         | ID "<"  Exp | ID "<=" Exp
         | ID ">"  Exp | ID ">=" Exp
         | ID "in" Exp | ID "not" "in" Exp
         | SetExp "&&" SetExp
```

Figure 6.7 Query syntax

sions. Its full syntax is defined in Figure 6.6. Mobl comes with an extensive set of libraries[4] containing reusable user interface elements, as well as APIs to call web services, perform web searches and get contextual information such as GPS location and device orientation.

At compile-time, the mobl compiler analyzes mobl imperative code to determine whether it relies on asynchronous methods and functions. If so, it automatically performs the continuation-passing style transform [Plotkin, 1975], turning code written in a synchronous style to the asynchronous style with callback functions as illustrated in the beginning of this section.

6.4.3 *Query*

Mobl's query language is *linguistically integrated* into the expression language part of the imperative language defined in the previous sub-section. The query abstraction is built on the `Collection` abstraction. Collections can be instantiated by the user, but for each entity there is also an `Entity.all()` collection defined, and for each one-to-many and many-to-many property there is a collection object as well. The `Collection` type has methods for filtering, sorting, paginating, aggregating and manipulating the collection. For instance:

```
Task.all().filter("done", "=", true)
          .order("due", false)
          .limit(10)
```

This expression represents the top ten results of tasks that are not done, sorted by due date in descending order. The disadvantage of encoding queries as method calls is the lack of static checking of property names as well as its verbose syntax. Therefore, mobl defines a thin syntactic layer, similar to LINQ [Meijer et al., 2006] on these methods as defined in Figure 6.7. This (optional) syntactic layer has the added advantage of enabling code completion support in the IDE. The same expression using the query syntax look as follows:

```
Task.all() where done == true
           order due desc
           limit 10
```

Full-text search queries are formulated using an entity's `search(phrase)` method, returning a `Collection` representing search results, ordered by relevance. Like any other `Collection`, results can be filtered and paginated.

These (virtual) query collections can be *reused* and *extended* by storing them in variables and passing them to functions. The result of a query is only calculated when required (e.g. when iterating over the result). Therefore, it is possible to define a method on an entity that returns a filtered collection (using `where` clauses), which is subsequently called and paginated in the user interface by adding `limit` and `offset` clauses to the method's resulting `Collection` object.

[4]`http://docs.mobl-lang.org`

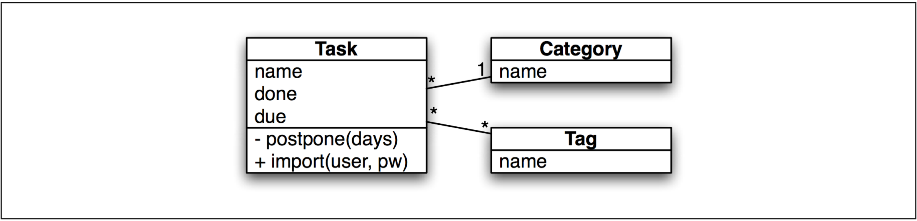


Figure 6.8 Todo list data model

6.4.4 *A Task Manager Data Model*

To demonstrate the data modeling language, as well as model-related logic, we describe the data model of a simple task manager (Figure 6.8). The mobl implementation of this data model is listed in Figure 6.9. The data model defines three entities: `Task`, `Category` and `Tag`. A task has a name, a done property to keep track of whether the task has been completed or not, a due date, a category it belongs to and a collection of tasks. Implicitly there is a one-to-many relationship between `Task` and `Category`: a task belongs (points to) a category, and a category has many tasks. The `inverse` annotations define the inverse relationship so that a task is automatically added to a category's collection of tasks when the `category` property is set and vice versa.

The function `postpone`, defined on `Task`, postpones the task a number of days, i.e. it moves the `due` date back. The static function (a function that is called on the entity itself, not an instance) `import` takes two arguments: a username and a password, and invokes a web service (located on the URI `/export`) to import all tasks defined on the server for the given user and cache them locally in the device's database. Web service results, by default, are returned as JSON[5] objects, a lightweight notation to represent structured data. The service returns an array of JSON objects, each representing a task. The `Task.fromSelectJSON` method is used to convert a JSON object into a `Task` object and cache it locally.

6.5 REACTIVE USER INTERFACES

In the current state of practice, web-based user interfaces are implemented at a low-level of abstraction. A lot of UI-related code is the result of *accidental complexity*. This section identifies the underlying problems and shows how mobl solves them by introducing *domain abstractions* such as data binding and reactive programming and by supporting *user-defined abstractions* such as controls.

Coupling View and Model The interaction between (persistent) application data and the user interface requires a lot of Controller *boilerplate code*. Data values have to be copied into the user interface when it is first loaded and stored back into data objects when certain events occur (e.g. when a "Save"

[5] `http://json.org`

```
entity Task {
  name     : String (searchable)
  done     : Bool
  due      : DateTime
  category : Category (inverse: tasks)
  tags     : Collection<Tag>
               (inverse: tasks)

  function postpone(days : Num) {
    this.due = DateTime.create(
      this.due.getFullYear(),
      this.due.getMonth(),
      this.due.getDate() + days);
  }
  static function import(user : String,
                          pw : String) {
    var tasksJSON =
      httpRequest("/export?user="
                  + user + "&pw=" + pw);
    foreach(t in tasksJSON) {
      add(Task.fromSelectJSON(t);
    }
  }
}
entity Category {
  name  : String
  tasks : Collection<Task>
            (inverse: category)
}
entity Tag {
  name  : String
  tasks : Collection<Task> (inverse: tags)
}
```

Figure 6.9 Mobl implementation of data model

button is pushed). Similarly, changes to data often give rise to changes in the user interface. For instance, when the user of a task manager application creates a new task in the database, the screen that displays all tasks has to be updated. Current frameworks require developers to encode this behavior manually.

Adapting the user interface is done by traversing the DOM and manipulating it in-place. These manipulations are imperative, e.g. "replace this node with this new node" and "remove this node".

ABSTRACTION A HTML page defines the content and structure of a page. CSS styles are used to apply styling to a HTML page (e.g. defining fonts, colors, borders and positioning), based on the knowledge it has about the page structure (using CSS selectors). A feature that both HTML and CSS do not

```
Def ::= Anno* "control" ID "(" {FArg ","}*
         ")" "{" SE* "}"
      | Anno* "screen" ID "(" {FArg ","}*
         ")" ":" Type "{" SE* "}"

SE ::= "<" HTMLID HtmlArg* ">"
         SE*
       "</" HTMLID ">"
     | Exp "(" {NamedExp ","}* ")"
        "{" SE* "}"
     | "var" ID "=" Exp
     | "list" "(" ID "in" Exp ")"
        "{" SE* "}"
     | "when" "(" Exp ")" "{" SE* "}"

HtmlArg ::= ID "=" Exp
          | "body" "=" Exp

NamedExp ::= Exp
           | ID "=" Exp

Anno ::= "@when" Exp
```

Figure 6.10 User interface syntax

support are *user-defined abstractions*. Reuse of page and style fragments, e.g. to reuse a calendar widget or a grid view control, is not supported by these languages. It also lacks support to *define* such reusable components. Consequently, JavaScript frameworks, such as jQTouch[6] and jQuery Mobile[7] attempt to fix this reuse issue by inventing an encoding. For instance, a framework like jQuery mobile may reinterpret a HTML tag `<div class="calendar"/>` as a calendar control, dynamically adapting the DOM to implement it. Nevertheless, such mechanisms only allow *use* of controls built into the framework, while *definition* of new controls has to be done using imperative JavaScript. Other frameworks, such as GWT[8] and Sencha Touch[9] abstract from HTML altogether with a Java (GWT) or JavaScript (Sencha) API to imperatively construct user interfaces.

Neither of these approaches is perfect. Annotating HTML is declarative, but uses an arcane encoding and attaching new meaning to HTML elements; using JavaScript to build the UI is imperative and low-level.

6.5.1 *Declarative User Interfaces*

Mobl supports user-defined abstractions for user interfaces through two core syntactic constructs: `screen`s and `control`s. Screens take up the entire size of the physical screen (hence the name) and are composed of controls, state variables, conditionals and loops. Both screens and controls have a name, a set of formal arguments and a body. Screens, in addition, have an optional return type. The full syntax of user interfaces in mobl is detailed in Figure 6.10.

The body of screens and controls consist of local variable declarations, HTML tags, control calls, conditionals (`when`, for conditionally rendering parts of the user interface) and loops (`list`, for rendering UI fragments for every item in a collection).

Local variables are used to store state relevant to the user interface. At the lowest level, mobl embeds HTML tags to construct a DOM. As can be seen from the syntax, HTML attributes in mobl cannot only contain strings, but arbitrary mobl expressions (numbers, variables, calculations, function calls).

Controls are *domain abstractions*, abstracting from low-level HTML. Controls are called by name with zero or more (optionally named) arguments and, optionally, a control body (in-between `{` and `}`). Screen and control arguments are passed by reference, enabling controls to write values back to the variables and properties passed to them, which is an essential element to enable *user-defined abstractions*, as will be demonstrated in the next sub-section.

6.5.2 *Data Binding and Reactive Programming*

Mobl user interfaces declare a View of the Model. *Data binding* establish a direct connection between View and Model. The View is automatically updated when the Model is changed, and the Model is updated when Model properties are changed in the View.

The following fragment of user interface code demonstrates how this data binding works using HTML tags:

```
var name = "John"
<input type="text" value=name/>
<span body="Hello, "+name/>
```

A local variable `name` is used to keep track of the user's name. The `<input>` HTML tag implements an input field and binds its value to the variable `name`. Consequently, *as the user types* in the text field (on every key stroke), the changed text box value is propagated back to the `name` variable.

The `<span>` tag implements a label in the user interface, whose *body* (the text that appears inside the label) is bound to the *expression* `"Hello, " + name`. Consequently, when the user types in the input field, the `name` is adapted,

[6]`http://jqtouch.com`
[7]`http://www.jquerymobile.com`
[8]`http://code.google.com/webtoolkit/`
[9]`http://www.sencha.com/products/touch/`

which, in turn, propagates to the `<span>` whose body is updated to reflect the new value of `name`.

Beside local variables and inline HTML, user interfaces use conditionals and loops which expose similar *reactive bahavior*. The `when` construct conditionally shows a part of the user interface as long as a certain condition holds, i.e. when the condition's value changes the `when` construct adapts the user interface accordingly.

As an example, in the following example the validation error remains hidden while the length of `name` exceeds 3 characters in length and appears as soon as the name is shorter than 3 characters:

```
var name = "John"
<input type="text" value=name/>
when(name.length < 3) {
  <span body="Name should be at least three
              characters"/>
}
```

The `list` construct iterates over a collection and renders its body for every item in the collection. Similar to `when`, `list` automatically adapts to changes in application state; it reacts to changes in the collection it iterates over, i.e. if items are added or removed from the collection, it adapts the user interface accordingly.

In summary, rather than imperatively manipulating the DOM to make changes to the user interface, mobl's user interfaces are *reactive* [Harel and Pnueli, 1985] – their structure and content depend on application state and adapts to changes automatically. As a result *boiler plate* code to implement this behavior manually is eliminated.

6.5.3 *Implementation*

Mobl's *data binding* establishes a direct connection between the value or an attribute of an HTML node in the DOM (representing an HTML tag) and a variable, property or expression in the Model. For variables and properties, a two-way binding is established: when the DOM is modified (for instance, when a user edits a text input field), this new value is propagated back into the variable or property. When the value of a variable or property changes this change is propagated back to the DOM. When a DOM node is bound to a more complex mobl expression, a one-way connection is established: whenever the value of the expression changes, it is propagated to the DOM.

Changes are propagated by using the Observer Pattern [Gamma et al., 1995]. Any piece of data in a mobl application is observable (including local variables, control arguments and entity properties) and UI constructs subscribe to change events of the observable values that they rely on. For instance, a label that shows a user's full name by concatenating its `firstName` with its `lastName` property, will subscribe to both of these properties and rerender itself whenever any of these two properties trigger a 'change' event.

Similarly, a `list` loop that iterates over a search collection, as is the case in Figure 6.13, subscribes to changes in the search collection. The search collection, in turn, keeps track of all `Task` objects and their `name` properties (which has been marked as `searchable`) and on every change, reevaluate if they match the search phrase or not.

6.5.4 *Reusable Controls*

Rather than requiring the duplication of the same HTML code in multiple places, *controls* can be used to implement *user-defined abstractions* for user interfaces. Control arguments are passed by reference, enabling constructing controls that abstract from low-level HTML while maintaining data binding semantics. Figure 6.11 demonstrates the implementation of the `textField` and `label` controls. Using these definitions the previous code, using HTML tags, can be reduced to the following, more clean and concise code, maintaining the same behavior. Section 6.7 gives more complex examples of control implementations.

```
var name = "John"
textField(name)
label("Hello, " + name)
```

Control arguments, as well as function and screen arguments, are passed in order, or can be explicitly named. For instance, `label("Hello")` is equivalent to `label(s="Hello")`. This proves particularly useful for optional arguments.

By annotating controls with a `@when <exp>` annotation it is possible to implement multiple version of a control, deciding at run-time which implemention to use, based on a run-time condition. An application of this will be demonstrated in Section 6.7.2.

6.6 NAVIGATION

Section 6.5 only considered the definition of single-screen interfaces. However, a typical application requires multiple screens and navigation between them.

The 'regular' web is navigated by clicking hyperlinks, sending the user from one web page to another. Browsing patterns can be random, and websites are not always organized in a strictly hierarchical manner. We observe that in mobile applications, navigation patterns are more stringent. Data-driven mo-

```
control textField(s : String) {
  <input type="text" value=s/>
}
control label(s : String) {
  <span body=s/>
}
```

Figure 6.11 Text field and label control implementation

```
screen prompt(question : String) : String {
  var answer = ""
  header(question) {
    button("Done", onclick={
      screen return answer;
    })
  }
  textField(answer)
}

screen root() {
  button("Ask", onclick={
    alert("Hello " + prompt("First name")
          + " " + prompt("Last name"));
  })
}
```

Figure 6.12 A screen with return type

bile applications typically organize information as trees. Some applications present the top-level of the tree as *tabs*, enabling the user to quickly switch between them. Deeper levels of information are presented in *list views*. When the user selects a list item, the current screen slides to the left, and a new one slides in from the right. Navigation between screens usually happens by navigating deeper into the hierarchy or moving back to a higher level (using the *back* button).

On iPhones and iPads, navigation is implemented using a stack of screens where only the top of the stack is visible. When an item is selected, a new screen, representing the item is pushed onto the stack and when the user pushes the back button, the screen at the top is popped off the stack and the previous screen appears. This screen stack has to be managed *manually* by the developer, by pushing and popping screens.

6.6.1 *Multiple screens*

This stack-based navigation very closely matches the call stack of function invocations in programming languages, a concept familiar to any programmer. Therefore, in mobl, screens are called as if they were functions and can optionally return a value using `screen return`.

As an example, Figure 6.12 defines a `prompt` screen, which takes a question as argument and returns the answer as result. The `textField` is bound to a local `answer` variable, which is returned by the screen when the "Done" button is clicked. The `root` screen contains a button, which, when clicked, invokes the `prompt` screen twice: first asking for the first name, then for the last name, then showing an alert pop-up box producing a greeting concatenating the results from the two `prompt` screen calls.

```
screen root() {
  var phrase = ""
  header("Tasks") {
    button("Add", onclick={ addTask(); })
  }
  searchBox(phrase)
  group {
    list(t in Task.search(phrase) limit 20){
      item {
        checkBox(t.done, label=t.name)
      }
    }
  }
}
screen addTask() {
  var t = Task()
  header("Add") {
    button("Done", onclick={
      add(t);
      screen return;
    })
  }
  textField(t.name)
  datePicker(t.due)
}
```

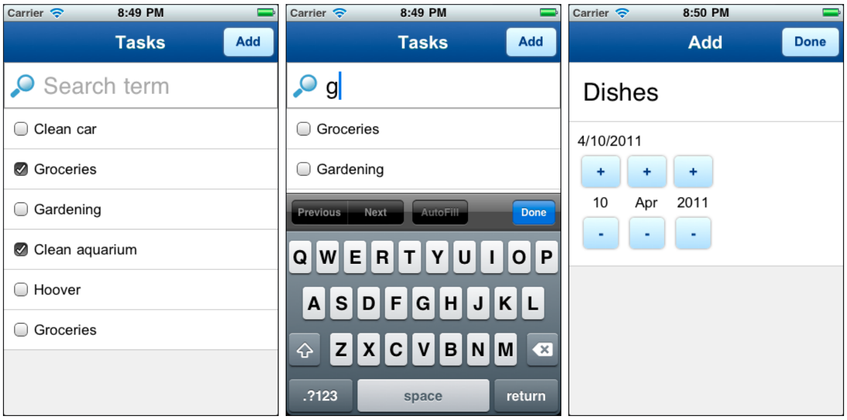


Figure 6.13 Tasks root screen with search

6.6.2 *A Task Manager User Interface*

Section 6.4.4 demonstrated how to define a data model for a task management application. Figure 6.13 shows how to build a simple user interface for this data model. When a mobl application launches, the `root` screen is loaded. Figure 6.13 defines the main screen of the task manager application. The screen shows a header, a search box and a group of at most 20 tasks that match the search phrase. Each task has a check box that can be used to mark the task as done.

The user interface is realized using a local user interface variable `phrase` to keep track of the search phrase. The search box is *bound* to this variable. The body of the `group` control contains a `list` construct that iterates over the search collection representing all tasks that match `phrase` with a maximum of 20 results. The body of the `list` construct instantiates an item control for every task, containing a `checkBox` which is bound to the `done` property of the task, as well as using the `name` property of the task for the checkbox label.

As the user types a search phrase in the search box, the changed search phrase is written back to the `phrase` variable. The `list` construct iterates over a collection that relies on the `phrase` variable. Therefore, it is recalculated as well. As a result, the list of tasks updates as the user is typing in the search phrase. Whenever new tasks are added to the database that match the search phrase, the task list will automatically be updated as well.

When the "Add" button is pushed, the `addTask` screen activates. The `addTask` screen uses a local variable `t` to keep a new task object whose `name` property is bound to a `textField` control and whose `due` property is bound to a `datePicker` control. The `button` control takes two arguments: a label to put on the button, as well as a named argument `onclick` of type `Callback`. Callbacks are snippets of imperative code, written using the same language as described in Section 6.4.2, to be executed when a certain event occurs (in this case an on click event). These snippets can be defined in-line in between `{` and `}`. When the user is done editting the name, he pushes the "Done" button, which adds the `t` object to the database and returns the user to the previous screen using a *screen return*.

6.7 HIGHER-ORDER CONTROLS

Mobl comes with a extensive library of reusable controls. These controls have been implemented in mobl itself, concisely defined using its abstraction, data binding and reactive programming features. Section 6.5 demonstrated how simple controls such as `textField` and `label` can be implemented top of HTML with data binding. This section will describe how higher-level controls are implemented. Specifically, the `tabSet` and `masterDetail` controls are described. The `tabSet` control is a higher-order control, taking other controls as arguments. The `masterDetail` takes control arguments as well, but in addition has two separate implementations: the 'right' implementation is chosen at run-time based on the screen width.

To support higher-order controls, mobl has a set of types to represent controls as values: `Control`, represents a control without arguments. Similarly, `Control1<Num>` represents a control with one argument, of type `Num`. Control arguments are passed as arguments are instantiated as any other control.

6.7.1 *Tab Set*

Figure 6.14 demonstrates how the `tabSet` control is used. It defines two controls: one for each tab. The `root` screen invokes the `tabSet` control with a list of tuples where each tuple represents a single tab. The first element of the tuple is the tab title (of type `String`), the second a reference to the control to use for the body of the tab (of type `Control`, a control without arguments). The `defaultTab` argument specifies the title of the tab to activate first. The screenshots in Figure 6.14 show the result: a tab bar along the top and when a tab is selected, the tab view changes to the selected tab's body.

Implementation Figure 6.15 details the entire implementation of the `tabSet` control. It takes two arguments: an array of tuples and the currently active tab. The tab set implementation relies on a few styles (styling in mobl will be discussed in Section 6.8) that are used with `block` controls. The `block` control is a simple stylable container control. An `activeTabButton` block appears as a selected tab, with rounded borders at the top. An `inactiveTabButton` block is similar, but looks like an inactive tab. The `visibleTab` and `invisibleTab` block respectively are visible and invisible. Thus, when a tab is not selected, its control is still rendered, it is just hidden using styling.

The `activeTab` argument keeps track of the currently selected tab. When a tab is selected, the `activeTab` variable is changed. Consequently, due to the reactive semantics, the styles on the tabs are toggled (tab content gets visible style, tab button gets selected style) and the new tab appears.

The `list((tabName, tabControl) in tabs) { ... }` notation uses tuple syntax on the left-hand side. It binds the first value of each tuple in `tabs` to `tabName` and the second to `tabControl`.

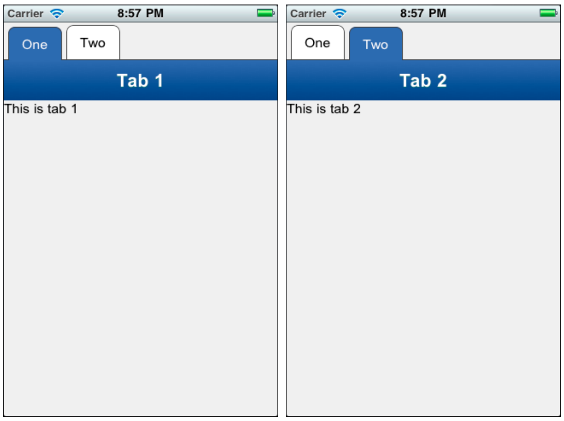
6.7.2 *Master-detail*

A common pattern in mobile user interfaces is the master-detail user interface pattern. There are two common implementations of this pattern, based on the available screen estate: On mobile devices with narrow screens, such as phones, initially a list of items appears and after selecting one, its details appear on a separate screen containing a back button to navigate back to the list. On devices with wider displays, such as tablet devices, the list of items appears along the left side of the screen and details of the selected items appear along the right.

Figure 6.16 shows how the `masterDetail` is used and how it looks on a narrow-screen device (first two screenshots) and on a wide screen (third screenshot). Two controls are defined: `taskItem` is used in the list view and `taskDetail` in the detail view. The `root` screen calls the master detail control with a collection representing all tasks ordered by due date in descending order, the `taskItem` and `taskDetail` controls.

Implementation Figure 6.17 shows the default implementation of the `master Detail` control (used for devices with a narrow screen). It takes three arguments: a collection of any type (`?` is syntactic sugar for the `Dynamic` type,

```
control tab1() {
  header("Tab 1")
  label("This is tab 1")
}
control tab2() {
  header("Tab 2")
  label("This is tab 2")
}
screen root() {
  tabSet([("One", tab1), ("Two", tab2)],
         defaultTab="One")
}
```



This is tab 1

This is tab 2

Figure 6.14 Using `tabSet`

```
control tabSet(tabs : [(String,Control)],
               activeTab : String) {
  list((tabName, tabControl) in tabs) {
    block(onclick={ activeTab = tabName; },
          style=activeTab==tabName ?
                activeTabButton
                : inactiveTabButton) {
      label(tabName)
    }
  }
  list((tabName, tabControl) in tabs) {
    block(activeTab==tabName ?
            visibleTab : invisibleTab) {
      tabControl()
    }
  }
}
```

Figure 6.15 `tabSet` implementation

```
control taskItem(t : Task) {
  checkBox(t.done, label=t.name)
}
control taskDetail(t : Task) {
  textField(t.name)
  datePicker(t.due)
}
screen root() {
  header("Tasks")
  masterDetail(Task.all() order by due desc,
               taskItem, taskDetail)
}
```

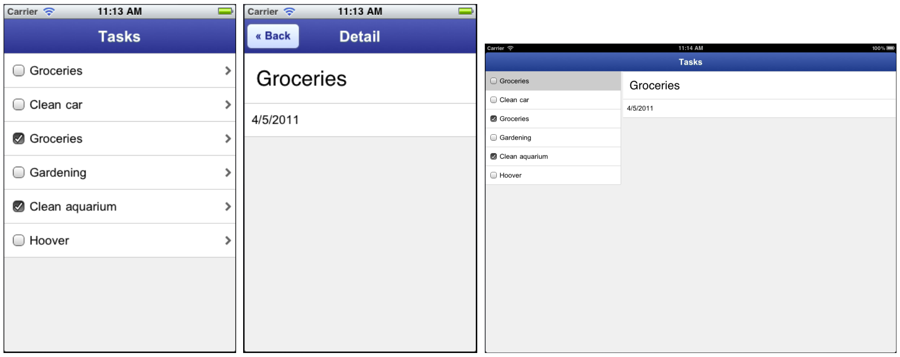


Figure 6.16 Using the `masterDetail` control

representing dynamically typed values), a `masterItem` control that is used for the list view and a `detail` control that is used to show the details of the item. Both the `masterItem` and `detail` are controls that take an item from the `items` collection as argument.

The control iterates over each item and renders an `item` control for it, using the `masterItem` control to render the content of the item. When the item is clicked, the `detailScreen` is called with both the item and the detail control as arguments. The implementation of the `detailScreen` renders a header control with a backButton, that, when click returns the user to the previous screen. It calls the `detail` control, passed as an argument with the `it` argument to render the detail view.

Figure 6.18 shows an alternative implementation of the `masterDetail` control that is only used when the browser window's inner width is larger than 500 pixels (expressed using the `@when` annotation), i.e. on wider screens. The arguments match exactly with the previous implementation, but the control body differs. A local variable `current` is used to keep track of the currently selected item in the collection. It is initialized to the first item in the collection (the `.one()` method limits the collection to a single item, returning the first one). The `sideBarStyle` is used to show a block to the left of the screen containing the list of `items`. The style (color) used for the item depends on

```
control masterDetail(items : Collection<?>,
    masterItem : Control1<?>,
    detail : Control1<?>) {
  group {
    list(it in items) {
      item(onclick={
        detailScreen(it,detail);
      }) {
        masterItem(it)
      }
    }
  }
}
screen detailScreen(it : ?,
                    detail : Control1<?>) {
  header("Detail") {
    backButton()
  }
  detail(it)
}
```

Figure 6.17 `masterDetail` implementation

whether it is selected or not. When the item is clicked, it is assigned as the `current` item. The block styled with the `mainContentStyle` appears right of the list and uses the `detail` control to render the currently selected item's details. The item rendered by the `detail` control automatically updates as new values are assigned to `current`.

6.8 STYLING

Cascading Stylesheets (CSS) are used to define the look and feel of a mobile web application. Styles are attached to HTML either automatically (using CSS selectors) or explicitly by attaching `class` attributes to HTML tags. Nevertheless, stylesheets are source of *code duplication* due to its lack of support for parameterization.

For instance, the following style can be attached to an HTML element to implement rounded corners. Due to the current state of browser support for the `border-radius` (a CSS3 feature), it uses browser-specific properties for Webkit and Gecko-based browsers (two common rendering engines) to make it work on all browsers:

```
.rounded-corners {
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
}
```

```
@when window.innerWidth > 500
control masterDetail(items : Collection<?>,
          masterItem : Control1<?>,
          detail : Control1<?>) {
  var current = items.one()
  block(sideBarStyle) {
    group {
      list(it in items) {
        item(style=current == it ? selectedItemStyle
                                 : notSelectedItemStyle,
             onclick={ current = it; }) {
          masterItem(it)
        }
      }
    }
  }
  block(mainContentStyle){
    detail(current)
  }
}
```

Figure 6.18 A wide-screen `masterDetail`

```
Def ::= "style" ID "{" StyleProp* "}"
      | "style" "mixin" ID
        "(" {StyleFarg ","}* ")"
        "{" StyleProp* "}"
      | "style" "$" ID "=" StyleVal

StyleProp ::= ID "=" StyleVal* ";"
            | ID "(" {StyleVal ","}* ")" ";"

StyleVal ::= CSSSTYLEVALUE
           | "$" ID
           | "$" ID "." "r"
           | "$" ID "." "g"
           | "$" ID "." "b"
           | StyleVal "+" StyleVal
           | StyleVal "*" StyleVal
           | StyleVal "-" StyleVal
```

Figure 6.19 Styling language syntax

However, whenever rounded corners are required with a radius other than 5 pixels, these three lines have to be duplicated and adapted.

6.8.1 *Styling in Mobl*

In order not to reinvent the wheel, mobl's styling language reuses all of CSS3's styling properties [W3C, 2011]. In addition, it adds styling constants, calculations based on these constants and style mixins. These additions were inspired by Sass[10], an extension of CSS that adds similar features.

Figure 6.19 defines the syntax for styles in mobl. At the HTML level, style values are attached to the `class` attribute of tags. Typically, controls have a `style` argument (of type `Style`) that is used to pass styles around. For instance, the `block` control:

```
style largeStyle {
  font-size: 100pt;
}
screen root() {
  block(largeStyle) { label("Large text") }
}
```

6.8.2 *Theming*

Applications can easily be themed with custom colors by overriding style constants used by the standard mobl library of controls. For this purpose, mobl supports global style constants that can be referenced in styles. When using RGB (Red-Green-Blue) colors, the individual color components can be accessed to build new colors:

```
style $baseColor = rgb(72, 100, 180)
style $textButtonColor = rgb($baseColor.r-50,
                             $baseColor.g-50,
         $baseColor.b-50)

style buttonStyle {
  color: $buttonTextColor;
  ...
}
```

The controls that come with mobl all derive their colors from the `$baseColor` constant. Therefore, simply overriding this constant and changing it to a different color, creates a new color theme based on the given base color. Figure 6.20 shows how buttons change with different `$baseColor` settings.

Parameterized styles are implemented using *style mixins*. Style mixins can be used and parameterized in other styles. Figure 6.21 implements a parameterized version of a border radius style taking the border *radius* as argument. The `buttonStyle` uses the mixin to realize a border radius of 5 pixels. Mobl comes with a library of reusable style mixins, including border radius and gradient mixins.

[10]`http://sass-lang.com`

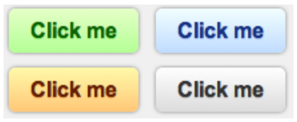


Figure 6.20 Four sample theme derivations

```
style mixin borderRadiusMixin($radius) {
  -moz-border-radius:    $radius;
  -webkit-border-radius: $radius;
  border-radius:         $radius;
}
style buttonStyle {
  color: $buttonTextColor;
  borderRadiusMixin(5px);
  ...
}
```

Figure 6.21 Style mixin example

6.9 DISCUSSION

To evaluate the coverage of mobl we have built a number of applications using mobl, ranging from simple toy applications such as a todo list manager and a tip calculator to more complex applications such as a twitter client, a conference planner application and even simple graphical games and a collaborative drawing application. Mobl receives a lot of interest from industry. Several companies are working on mobile applications built using mobl. Together with our user community we grew a library of reusable controls, ranging from basic, such as labels and buttons, to more complex, such as the tab set, a master-detail, accordion, date picker and context menu controls. The definitions of these controls are all declarative and concise. A member of the community has also developed a framework (using mobl) to enable unit testing of the data model and logic.

This section discusses the limitations of our approach and compares it to related work.

LANGUAGE LIMITATIONS While mobl's type checker checks many program properties, it does not yet check everything. For instance, `item` controls controls have to be nested within `group`s to be rendered properly. Mobl does not yet support declaring such nesting requirements.

Data synchronization with web services currently has to be implemented manually. In the future we intend to support transparent data synchronization web services as part of the mobl language, thereby eliminating the custom synchronization code that need to be written on an application-by-application basis.

Mobile web applications generated by mobl are portable to any mobile platform that supports HTML 5. However, the user interface does not adapt to the look-and-feel of the platform, while mobl *supports* this variability using `@when` annotations, we have not yet developed many platform-specific control implementations.

Performance The performance of mobile *web* applications will always be worse than *native* applications, just as web applications in general are slower than native desktop applications. Nevertheless, by caching both the application and its data locally and the recent performance improvements of (mobile) browsers, performance of mobile web applications is very reasonable. While performance has not been the primary focus of the mobl compiler thus far, it is possible to produce an optimized build which eliminates all unused definitions from the generated JavaScript and CSS files. In addition, unnecessary whitespace is removed and variables are renamed with shorter names to considerably reduce the application's download size.

Good Web Citizenship While mobl uses the web as a medium to deliver applications, and uses web technologies to run applications, a mobl application is not built like a regular web application: a mobl application does not consist of pages with unique URLs; breaks the browser's back button; and is not indexable by search engines. We intend to solve some of these issues. A working back button is relatively easy to implement. Full history support is much more complex, requiring some type of encoding of the application state in the URL of the application. Indexing mobile applications can be useful for some data-driven applications. A tool such as CrawlJax [Mesbah et al., 2008] could be used to generate a static, indexable version of the application.

Web Application Limitations While web applications have the advantage of being portable, they have limitations too. HTML5 offers many JavaScript APIs that give access to various device services, but their implementation in mobile devices is not always complete. Access to audio and video services is limited — it is possible to play an audio or video file, but only by launching the dedicated audio or video player. Access to other device-specific features such as bluetooth, the built-in compass, camera and local file storage are not supported yet.

A way around these restrictions is a native/web hybrid approach. PhoneGap[11] allows a developer to build applications using web technologies, and expose additional native APIs including a file storage API and a camera API through JavaScript, an approach that works nicely with mobl. Applications built with PhoneGap can be deployed as native applications through e.g. the Apple AppStore or Android Marketplace.

Web applications have limitations in user experience as well. It is very difficult to reproduce certain native application behaviors in web applications. Inertia scrolling is one such behavior, where, after a finger flick on the screen, the screen keeps scrolling for a while longer after the finger no longer touches

[11] `http://www.phonegap.com`

the screen. There are a number of projects that attempt to emulate this behavior in the browser, but it has proven very difficult to do perfectly. Fixed positioning is another behavior that is difficult to achieve in mobile browsers. A control that has a fixed position, does not move when the rest of the screen scrolls. A typical example is a screen header. A header is positioned at the top of the screen and while the rest of the content scrolls, the header remains fixed at the top.

6.9.1 *WebDSL*

In previous work we developed WebDSL [Visser, 2007b], a domain-specific language for the development of RESTful web applications. From a WebDSL program, the WebDSL compiler generates a Java web application, deployable in any Java servlet container.

Mobl borrows many concepts from WebDSL. For instance, like WebDSL, mobl is statically verifiable (see Chapter 2) and has similar constructs for the definition of data models.

State and Event Handling Syntactically, the definition of WebDSL and mobl user interfaces are similar, but their semantics differ when it comes to the time of data binding. The unit of interaction within a WebDSL application is a HTTP request, either executed using an AJAX call, a form submit or page request. Pages are reconstructed on every request, instead of incrementally updated as is the case in mobl. Incremental user interface updating is cheap when maintaining state locally, while implementing incremental updates efficiently in a client-server application requires *application state* to be maintained on the server as well as client, which would require the storage of application state for potentially thousands or millions of users.

Handling of events in mobl is more fine-grained than in WebDSL: when editing a data object in WebDSL, changes are persisted only when the edit form is submitted to the server, rather than instantaneously as is the case with mobl applications. Since all interaction and persistence happens locally, such continuous persistence is much cheaper to implement. Sending every keystroke to the server would be very expensive.

Extension Mobl has a different philosophy than WebDSL when it comes to language extension. WebDSL developed many abstractions as built-ins, including built-in types, controls and functions. As a result, any modifications or improvements to these constructs requires extension or adaptation of the compiler. Mobl takes the approach of library extension. Rather than hard-coding types and controls into the compiler, they are defined in libraries either encoded in mobl itself, or through the native Javascript interface. The advantage of this approach is that users can easily add new functionality to mobl, without the need to know how its compiler works. This approach is currently in process of being adopted in WebDSL as well.

6.9.2 *Related work*

DSLs for mobile development Behrens [2010] describes a domain-specific language for creating *native* mobile applications, using a single language from which both iPhone and Android applications can be generated. Similar to mobl, the language comes with an IDE plug-in for Eclipse that supports error high-lighting, code completion and reference resolving. Berhens' language has a number of high-level controls built into the language, including sections, detail views and cells. It can fetch its data from data providers. However, the DSL currently only supports data *viewing* and is not as flexible as mobl; defining custom controls is not supported, for instance.

Kejriwal and Bedekar [2009] developed MobiDSL, an XML-based language for developing mobile web applications. Unlike mobl, the application is executed on the server and plain HTML is sent to the mobile device. MobiDSL comes with a number of built-in controls, such as query views, page headers and search requests that can be used to build pages. It is not possible to define custom controls, nor is there specific IDE support available.

Google Web Toolkit is a tool that enables client-side web applications using Java. The use of Java has the advantage of having excellent IDE support. A GWT plug-in[12] enables access to HTML 5 APIs such as geolocation and local databases. Like mobl, GWT applications are compiled to a combination of HTML, Javascript and CSS. However, user interfaces using GWT have to be defined using verbose Java code. In addition, GWT does not provide data binding or reactive programming support, therefore requiring a lot of boiler plate code to bind the Model to the View.

Reactive User Interfaces Courtney and Elliott [2001] developed Fruit, a Haskell framework that applies functional reactive programming [Nilsson et al., 2002, Elliott and Hudak, 1997, Wan and Hudak, 2000] to user interfaces. It is based on signals (streams of events) and signal transformers (functions that transform streams of events). On top of these concepts, Fruit builds a purely functional user interface library. Mobl's user interfaces are also reactive, but not based on pure functions. Concepts such as signals and signal transformers are not exposed to the developer in mobl. Instead, events triggered by changes in data or control events, result in updates to the user interface.

Meyerovich et al. [2009] describe FlapJax, a language for building AJAX applications. Flapjax is also built on the concept of event streams: streams of events that model, for instance, mouse movements, clicks and web service responses. These streams can be filtered and merged to build responsive user interfaces. Mobl takes a more traditional approach to event handling. Events in mobl trigger event handler logic, which can modify application state potentially resulting in user interface changes.

[12] `http://code.google.com/p/gwt-mobile-webkit/`

6.10 CONCLUSION

In this chapter we introduced mobl, a new language for developing mobile web applications. Mobl *linguistically integrates* languages for data model definition, user interface, styling and application logic. It introduces *domain abstractions* to abstract from accidental complexity and irrelevant details of the platform and domain. Mobl's support for *user-defined abstractions*, data binding and reactive program enable the *reusable* implementation of both simple controls (labels and button) and higher-level controls (tab sets and master-detail). Mobl automates the tasks typically manually encoded in Controller logic, thereby reducing the amount of boiler plate code that needs to be written. Mobl has received a lot of interest from industry. A number of companies have already committed to implement their mobile applications using mobl.

Postscript: Mobl

Mobl allowed us to develop a new language from scratch, applying lessons learned in previous chapters to a new domain. In addition, it enabled us to do a few things differently than in WebDSL based on lessons learned. Once a language has been released and there is a body of programs written using it, it becomes difficult to change fundamental design decisions. A new language is a fresh start. Therefore, in this postscript we describe the design decisions we made differently in mobl than in WebDSL. These are not specific to any domain in particular, but general language design lessons learned.

NAMESPACING

WebDSL has no language support for namespaces, which is an issue as applications grow larger. In mobl, every module has its own namespace. Every reference to screens, controls, styles, functions and types is fully qualified, either explicitly or based on import statements.

SEPARATE COMPILATION

WebDSL applications have to be compiled as a whole, rather than module by module. Largely, this is due to the aspect weaving features (such as access control and WebWorkFlow) that make separate compilation difficult to implement. At the time of this writing there is a student working on this problem, but the time between saving a WebDSL module and being able to test it in the browser is still long. As other languages that are based on aspect weaving (such as AspectJ[13]) show, there is performance overhead in compiling languages that require aspect weaving. In the end it is a trade off between compilation time and language features.

In mobl we have avoided aspect weaving and lengthy compilation. This is partly supported by the domain: there is less need for features that require aspect weaving. In addition, mobl's target platform does not require additional compilation and deployment steps (like WebDSL produced code, which has to be compiled by a Java compiler, archived into a `.jar` file and deployed to a server). Mobl compiles applications on a module-by-module basis. Whenever the user saves a module, it is immediately recompiled. A refresh in the mobile browser enables the immediate testing of changes made.

[13]`http://www.eclipse.org/aspectj/`

SYNTACTIC EXTENSION VERSUS LIBRARY EXTENSION

Traditionally, the way to add features to WebDSL has been to extend its syntax and compiler. As a result, the WebDSL has grown to around 30,000 lines of Stratego code. The addition of new types, or a new built-in templates requires knowledge of compiler construction in general and WebDSL's compiler architecture in particular.

The approach in mobl has been different. The language itself is kept small, and all controls, types etc. are defined in mobl libraries. As a result, at the time of this writing, the mobl compiler is around 7,500 lines and the mobl standard library consists of around 4,000 lines of mobl code. Naturally, mobl is only about 1.5 years old and WebDSL about 4.5 years, so undoubtedly the mobl compiler will increase in size over time. At the time of this writing, WebDSL has started to adopt the use of libraries and many built-in templates are now reimplemented as templates implemented in WebDSL.

While WebDSL supports modules, it does not yet have a practical infrastructure for easily reusing code. In practice, reuse is implemented by copying WebDSL modules from one project to the next. Lack of namespacing support makes reuse harder, because verbose name need to be chosen for templates and function in order not to clash with existing code.

By contrast, every mobl release comes with a library of reusable controls, screens, styles, types and functions in mobl source form. The directory containing this standard library is added to the module search path of the mobl compiler by default. When a one of the modules from the standard library is imported, it is compiled and the generated files are placed in the project's output directory along with any resource files that the module depends on (such as images and Javascript files).

PORTABILITY

WebDSL applications can be accessed using any platform as long as it has a browser and is therefore platform-independent from the end-user perspective. However, applications need to be installed in a server environment that typically *does* impose a particular software platform. This is not the case for mobl. Mobl applications are pure client-side web applications that run in any HTML5 enabled web browser, only requiring a server to serve the application files — something that any web server can do.

With this in mind, we created explicit dependencies on the underlying platform, by supporting escapes to HTML and JavaScript. These escapes increase the coverage of mobl, as any web feature not directly supported by mobl itself can be used through embedding HTML or by invoking native Javascript.

Recently, a master student has been developing a back-end for mobl that targets *native* iPad applications, rather than the web. Consequently, many of the escapes that mobl offers to the underlying platform cannot be used in this context. As a result, libraries that use web-specific features cannot be reused with the iPad back-end. These libraries have to be largely redeveloped by implementing them in Objective-C. However, potentially, libraries that build on top of a set of controls that are consistently implemented in both the web and iPad version, can be used to develop mobile applications that can both be run in the browser as well as natively on the iPad.

Since mobl now has two back-ends we will likely see similar maintainability problems we have seen in WebDSL (Chapter 5). Could PIL be applied to mobl after all? While the iPad back-end is still in development and in a prototype state, there are a few potential problems in applying PIL. First of all, there is a considerable amount of custom written, platform-specific code that wraps iPad user interface controls as mobl controls. In addition, there is a lot of work put into emulating a lot of standard web browser behaviour using the iPad APIs, such as positioning of controls, implementing certain elements of the styling language and data persistence.

In addition, PIL works best when there are two or more back-ends whose generated code has a very similar structure. However, this is not the case for the current implementation of the back-ends. For instance, Javascript natively supports closures, which are heavily relied on in the Javascript back-end implementation; Objective-C does not support (mutable) closures and therefore the back-end generates code to emulate them. The Javascript back-end relies on asynchronous APIs and performs a continuation-passing style transformation. Most APIs on the iPad are not asynchronous and therefore do not require such a complex transformation. Therefore, whether PIL will solve maintenance problems remains to be seen.

Conclusion

7

The core research question of this thesis is "How to design and implement statically verifiable and syntactically integrated domain-specific languages?" While we do not yet have an all-encompassing answer this question, we did study a number of aspects of the design and implementation of syntactically integrated DSLs in this thesis. The lessons learned are valuable contributions toward a systematic approach to DSL design and implementation.

We demonstrated that syntactic integration enables static verification of applications, supporting early error detection. Good *coverage* of a DSL can be ensured by developing a core DSL at a relatively low-level as well as supporting a *native interface* to be able to invoke underlying platform-specific features. On top of the core language, layers of abstraction can be built that enable programs to be written at a higher-level of abstraction, while still having the option to escape to the lower-level of abstraction if required. We developed and implemented techniques to manage the complexity of DSL compilers, supporting both modularity and separation of concerns. In order to more efficiently implement DSLs that target multiple software platforms, we introduced PIL, a Platform Independent Language operating at the level of software platforms.

To evaluate our techniques, we applied them in the design and implementation of WebDSL. To explore the replicability of the approach, we conducted a second case study in DSL design: mobl, a language to rapidly develop mobile web applications. Mobl applies many of the described techniques to the domain of mobile web applications.

7.1 SUMMARY OF CONTRIBUTIONS

Each of the chapters in this thesis have individual contributions. We summarize the core contributions as follows:

- A declarative, rule-based approach to linguistic integration and consistency checking (Chapter 2).
- The WebWorkFlow language, a flexible language enabling the construction of complete web applications involving workflows (Chapter 3).
- The *Code Generation by Model Transformation* approach to building DSL compilers and its implementation in the WebDSL compiler (Chapter 4).
- The PIL language, a source code level intermediate language aimed at DSL compilers that reduces the amount of work required to build and maintain multiple DSL compiler back-ends (Chapter 5).

- The mobl language, a language to rapidly develop mobile web applications (Chapter 6).

7.2 WEBDSL AND MOBL IN PRACTICE

WebDSL[1] is used in the Model-Driven Engineering class taught by the software engineering department. Students have to build a non-trivial web application to learn about the advantages of developing software with DSLs. The feedback from students has lead to many improvements in the WebDSL language and tooling. In addition, WebDSL has been used to develop a number of web applications used in production, most notably researchr.org[2], an online bibliography tool and YellowGrass[3], a tag-based issue tracker that we use to track bugs for all our projects.

Mobl[4] has a growing community (90+ members on the mailing list at the time of this writing) of industrial users that build applications, submit issues, contribute to the mobl library and, occasionally, contribute fixes to the compiler. The feedback gained from the user community is steering the development of mobl. A number of companies (unrelated to our research group) are developing commercial products using mobl. Mobl has been featured on a number of web sites, including a feature article on InfoQ[5].

7.3 RESEARCH QUESTIONS

RESEARCH QUESTION 1

Is the lack of static verification a common problem in today's web frameworks? If so, how can that problem be remedied?

In Chapter 2 we surveyed a number of state-of-the-art web frameworks. We introduced errors in programs developed using these frameworks and observed how the errors become manifest. Indeed, the problem of late discovery of faults is not specific to the Seam framework, but occurs in other frameworks as well. Errors appeared only at run-time, often with obscure error messages that are hard to trace back to their origin. We proposed to remedy these problems by designing a language in such a way that it integrates all application aspects into a single, syntactically integrated language. Static analyses can be implemented that verify various application properties involving different application aspects, because the language shares a single type system.

We demonstrated how the static verification component of such a compiler can be implemented by detailing the implementation of verification rules for a subset of WebDSL using Stratego.

[1]`http://webdsl.org`
[2]`http://researchr.org`
[3]`http://yellowgrass.org`
[4]`http://mobl-lang.org`
[5]`http://www.infoq.com/articles/Mobl`

RESEARCH QUESTION 2

> *How can the level of abstraction in a DSL be raised without reducing its coverage?*

By design, domain-specific languages cover only a limited class of applications — otherwise they would be considered general purpose languages. However, within a domain there is a trade-off to be made between high coverage using a lower-level language, and lower coverage using a higher-level language. In Chapter 3 we demonstrated we can have the best of both worlds by taking a lower-level language as core language, and layering higher-level abstractions on top. We demonstrate this approach through the design of *WebWorkFlow*, an extension of WebDSL that adds language-level workflow constructs. Unlike other workflow languages it is not constructed as a separate DSL, but builds on top of an existing, lower-level DSL (WebDSL). In WebWorkFlow, workflows can be defined at three levels: process-level, procedure-level and pure WebDSL. Process-level descriptions are very high-level, supporting the common workflow patterns such as sequential task execution and loops. If this level is not flexible enough, parts of the workflow can be expressed using WebWorkFlow's more flexible, but more verbose *procedure* abstraction. If procedures do not support sufficient flexibility, parts of workflows can be expressed in plain WebDSL.

RESEARCH QUESTION 3

> *How to maintain separation of concerns and efficiently combine analysis with transformation in a DSL compiler?*

Chapter 4 described techniques for improving separation of concerns in the implementation of code generators for domain-specific languages, based on our experience with the implementation of WebDSL. The core technique employed is *code generation by model transformation*, that is, the generation of a structured representation (model) of the target program instead of plain text. This approach enables the transformation of code after generation, which in turn enables the extension of the target language with features that allow better modularity in code generation rules (such as partial classes, partial methods and expression blocks). The technique can also be applied to implement *syntactic extension* of a DSL, through internal code generation, gradually transforming high-level extensions of a DSL to lower-level constructs.

The mobl compiler still generates plain text source code. At the time of initial development, it was deemed the most pragmatic and productive solution to quickly develop a compiler for the mobl language. However, over time the compiler grew larger and started to suffer from similar problems as WebDSL. Presently, there are plans to refactor the mobl compiler to use code generation by model transformation as well.

RESEARCH QUESTION 4

How to reduce the effort to maintain multiple platform back-ends in the compiler?

Chapter 5 introduced PIL, a Platform Independent Language to be used as a high-level intermediate language. Instead of generating code for each software platform individually, a DSL compiler can generate PIL code, which is subsequently mapped to target platform code. As a result, only a single compiler back-end has to be maintained. Unlike traditional intermediate languages, PIL has a readable concrete syntax similar to Java. PIL is aimed as a language to be used purely for code generation — it is a small, easy to port language. PIL platform back-ends can be reused in multiple DSL compilers. An extension of PIL, called PIL/G adds features *specific to the domain of code generation* – similar features that are added in a more ad-hoc manner to Java in Chapter 4, including partial classes and methods, identifier concatenation and expression blocks. PIL is evaluated by the creation of a prototype back-end for WebDSL, enabling the WebDSL compiler to generate code for Python as well as the Java platform.

RESEARCH QUESTION 5

How can the language design and implementation techniques developed for WebDSL be applied to the mobile web domain?

Chapter 6 described mobl, a new language to rapidly develop mobile web applications. Mobl applies a number of design ideas previously developed for WebDSL, including design for static verification (Chapter 2) and syntactic language extensions (Chapter 3) to the mobile domain and adapts them appropriately. While syntactic extensions are part of mobl, the primary way to extend mobl is using *libraries*. Much of the functionality in mobl is implemented in mobl itself, rather than being encoded into the compiler. As a result, mobl can easily be extended by users to support new abstractions. While the mobl compiler does not yet generate code using code generation by model transformation techniques (Chapter 4), this is considered future work. In the postscript of Chapter 5 we described our reasons not to use PIL for mobl and instead use web technologies to achieve portability.

7.4 FUTURE WORK

7.4.1 *Systematic approach*

The goal of the MoDSE project is to develop a systematic approach to DSL design and implementation. The case studies conducted as part of the work for this thesis were steps toward that goal. Language design is difficult, the design space is typically very large. While there is likely no algorithm to be discovered that turns the description of a domain into an appropriate language, we do believe more lessons can be extracted by performing more case studies, also in domains completely separate from the web domain.

7.4.2 *Evaluation*

The evaluation of programming languages is difficult. Our approach to evaluation has been to identify specific problems and demonstrate how our languages solve those problems. However, we did not evaluate another important feature of our DSLs: the significant reduction of the amount of code that needs to be written to build an application. We have anecdotal evidence that is the case, but we have not set up an experiment to investigate this.

Such an experiment could involve an expert familiar with some competing web framework, e.g. Ruby on Rails. The developer would be asked to develop a particular non-trivial application. Subsequently, the same application would be developed using WebDSL and the number of lines required could be compared. Ideally, this experiment would have to be repeated a number of times for different applications, to avoid a bias toward a particular type of web application.

Bibliography

A. Kraus, A. K. and Koch, N. (2007). Model-driven generation of web applications in UWE. In *Model-Driven Web Engineering (MDWE'07)*, Como, Italy. (Cited on page 110.)

AndroMDA.org (2007). AndroMDA documentation. `http://galaxy.andromda.org`. (Cited on page 106.)

Arnoldus, J., Bijpost, J., and van den Brand, M. (2007). Repleo: a syntax-safe template engine. In Consel, C. and Lawall, J. L., editors, *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007*, pages 25–32, Salzburg, Austria. ACM. (Cited on page 110.)

Atkins, D. L., Ball, T., Bruns, G., and Cox, K. (1999). Mawl: A domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346. (Cited on page 42.)

Balland, E. and Brauner, P. (2008). Term-graph rewriting in Tom using relative positions. *Electronic Notes in Theoretical Computer Science*, 203(1):3 – 17. Proceedings of the Fourth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2007). (Cited on page 107.)

Balland, E., Brauner, P., Kopetz, R., Moreau, P., and Reilles, A. (2007). Tom: Piggybacking rewriting on java. *Lecture Notes in Computer Science*, 4533:36–47. (Cited on page 105.)

Barry, D. K. and Stanienda, T. (1998). Solving the java object storage problem. *computer*, 31(11):33–40. (Cited on page 151.)

Bast, W., Belaunde, M., Blanc, X., Duddy, K., Griffin, C., Helsen, S., Lawley, M., Murphree, M., Reddy, S., Sendall, S., Steel, J., Tratt, L., Venkatesh, R., and Vojtisek, D. (2005). MOF QVT final adopted specification. OMG document `ptc/05-11-01`. (Cited on page 106.)

Behrens, H. (2010). MDSD for the iPhone. In *SPLASH '10: Proceedings of Object oriented programming systems languages and applications companion*. (Cited on page 172.)

Benitez, M. E. and Davidson, J. W. (1988). A portable global optimizer and linker. *SIGPLAN Not.*, 23(7):329–338. (Cited on pages 9 and 132.)

Bentley, J. (1986). Programming pearls: little languages. *Commun. ACM*, 29. (Cited on page 1.)

Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2):171–188. (Cited on page 74.)

Bézivin, J. (2006). Model driven engineering: An emerging technical space. In Lämmel, R., Saraiva, J., and Visser, J., editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer. (Cited on page 106.)

Bezivin, J., Hammoudi, S., Lopes, D., and Jouault, J. (2004). Applying MDA approach for web service platform. In *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International*, pages 58–70, Washington, DC, USA. IEEE Computer Society. (Cited on page 133.)

Brabrand, C., Moeller, A., and Schwartzbach, M. I. (2002). The <bigwig> project. *ACM Trans. Internet Techn.*, 2(2):79–114. (Cited on page 41.)

Brabrand, C., Mller, A., and Schwartzbach, M. I. (2001). Static validation of dynamically generated html. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 01, Snowbird, Utah, USA, June 18-19, 2001*, pages 38–45. ACM. (Cited on page 41.)

Brambilla, M., Cabot, J., and Comai, S. (2007a). Automatic generation of workflow-extended domain models. In Engels, G. et al., editors, *Model Driven Engineering Languages and Systems (MoDELS 2007)*, volume 4735 of *LNCS*, pages 375–389. Springer. (Cited on page 63.)

Brambilla, M., Ceri, S., Fraternali, P., and Manolescu, I. (2006). Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 15(4):360–409. (Cited on page 63.)

Brambilla, M., Comai, S., Fraternali, P., and Matera, M. (2007b). Designing web applications with WebML and WebRatio. *Web Engineering: Modelling and Implementing Web Applications*, pages 221–260. (Cited on page 42.)

Bravenboer, M., de Groot, R., and Visser, E. (2006a). MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In Lämmel, R. and Saraiva, J., editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, volume 4143 of *Lecture Notes in Computer Science*, pages 297–311, Braga, Portugal. Springer Verlag. (Cited on page 126.)

Bravenboer, M., Dolstra, E., and Visser, E. (2007). Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In Lawall, J., editor, *Generative Programming and Component Engineering (GPCE 2007)*, pages 3–12, New York, NY, USA. ACM. (Cited on page 42.)

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70. Special issue on experimental software and toolkits. (Cited on pages 1, 30, 45, 71, 76, 78, 81, and 105.)

Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. (2006b). Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178. (Cited on pages 34, 92, and 105.)

Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Schmidt, D. C., editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 365–383, Vancouver, Canada. ACM Press. (Cited on pages 42, 50, and 118.)

Bringert, B., Höckersten, A., Andersson, C., Andersson, M., Bergman, M., Blomqvist, V., and Martin, T. (2004). Student paper: Haskelldb improved. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115, New York, NY, USA. ACM. (Cited on page 41.)

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. J. (2003). *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional. (Cited on page 106.)

Burns, E. and Kitain, R., editors (2006). *JavaServer Faces Specification. Version 1.2*. Sun Microsystems. (Cited on page 74.)

Ceri, S., Fraternali, P., and Bongio, A. (2000). Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157. (Cited on page 42.)

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2). (Cited on page 133.)

Comai, S., Matera, M., and Maurino, A. (2002). A model and an xsl framework for analyzing the quality of webml conceptual schemas. In Spaccapietra, S., March, S. T., and Kambayashi, Y., editors, *Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Tampere, Finland, October 7-11, 2002, Proceedings*, volume 2503 of *Lecture Notes in Computer Science*, pages 339–350. Springer. (Cited on page 42.)

Cook, S., Jones, G., Kent, S., and Wills, A. C. (2007). *Domain-Specific Development with Visual Studio DSL Tools*. Addison Wesley. (Cited on page 106.)

Cooper, E., Lindley, S., Wadler, P., and Yallop, J. (2006). Links: Web programming without tiers. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W. P., editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer. (Cited on page 40.)

Cordy, J. (2006). The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210. (Cited on page 105.)

Courtney, A. and Elliott, C. (2001). Genuinely functional user interfaces. In *PLI*. (Cited on page 172.)

Coward, D. and Yoshida, Y. (2003). *Java Servlet Specification. Version 2.4*. Sun Microsystems. (Cited on page 74.)

Curbera, F., Goland, Y., Klein, J., Leymann, F., Thatte, and Weerawarana, S. (2003). Business process execution language for web services, version 1.1. Technical report, IBM. (Cited on pages 6 and 49.)

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646. (Cited on page 106.)

Davidson, J. W. and Fraser, C. W. (1984). Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526. (Cited on page 132.)

Deligiannis, I. S., Stamelos, I., Angelis, L., Roumeliotis, M., and Shepperd, M. J. (2004). A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143. (Cited on page 8.)

DeMichiel, L. and Keith, M., editors (2006a). *JSR 220: Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements*. Sun Microsystems. (Cited on page 75.)

DeMichiel, L. and Keith, M., editors (2006b). *JSR 220: Enterprise JavaBeans, Version 3.0. Java Persistence API*. Sun Microsystems. (Cited on page 74.)

Dumas, M. and ter Hofstede, A. H. M. (2001). Uml activity diagrams as a workflow specification language. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML 2001)*, pages 76–90, London, UK. Springer-Verlag. (Cited on pages 6 and 49.)

ECMA (2009). ECMA-262 ECMAScript language specification. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`. (Cited on page 151.)

Eder, J., Groiss, H., and Liebhart, W. (1997). The Workflow Management System Panta Rhei. *Advances in Workflow Management Systems and Interoperability. Springer, Istanbul, Turkey, August*, pages 129–144. (Cited on page 63.)

Efftinge, S. and Friese, P. (2007). openArchitectureWare. `http://www.eclipse.org/gmt/oaw`. (Cited on pages 94 and 106.)

Efftinge, S., Friese, P., Haase, A., Kadura, C., Kolb, B., Moroff, D., Thoms, K., and Völter, M. (2007). *openArchitectureWare User Guide. Version 4.2*. www.openarchitectureware.org. (Cited on page 106.)

Efftinge, S. and Völter, M. (2006). oAW xText - a framework for textual DSLs. In *Modeling Symposium, Eclipse Summit*. (Cited on page 102.)

Ekman, T. and Hedin, G. (2004). Rewritable reference attributed grammars. In Odersky, M., editor, *18th European Conference Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169, Oslo, Norway. Springer. (Cited on page 111.)

Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *ICFP*, pages 263–273. (Cited on page 172.)

Fowler, M. (2005). Fluent interfaces. `http://www.martinfowler.com/bliki/FluentInterface.html`. (Cited on page 2.)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional. (Cited on pages 141, 144, and 158.)

George, L. (1997). MLRISC: Customizable and reusable code generators. Technical report, AT&T Bell Laboratories, Murray Hill, NJ. (Cited on pages 9 and 132.)

Gray, R. W., Levi, S. P., Heuring, V. P., Sloane, A. M., and Waite, W. M. (1992). Eli: a complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130. (Cited on page 111.)

Groenewegen, D. and Visser, E. (2008). Declarative access control for WebDSL: Combining language integration and separation of concerns. In Schwabe, D. and Curbera, F., editors, *International Conference on Web Engineering (ICWE 2008)*. IEEE CS Press. (Cited on pages 45, 51, 53, 71, 88, and 116.)

Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. (2008). When frameworks let you down. platform-imposed constraints on the design and evolution of domain-specific languages. In Gray, J. et al., editors, *Domain Specific Modelling (DSM'08)*, pages 64–66. (Cited on pages 9, 12, 116, and 135.)

Groenewegen, D. M., Hemel, Z., and Visser, E. (2010). Separation of concerns and linguistic integration in WebDSL. *IEEE Software*, 27(5). (Cited on page 12.)

Groenewegen, D. M. and Visser, E. (2009). Integration of data validation and user interface concerns in a dsl for web applications. In van den Brand, M. and Gray, J., editors, *Software Language Engineering, Second International Conference, SLE 2009, Denver, USA, October, 2009. Revised Selected Papers*, Lecture Notes in Computer Science. Springer. (Cited on pages 26 and 45.)

Harel, D. and Pnueli, A. (1985). On the development of reactive systems. *Logics and models of concurrent systems*. (Cited on page 158.)

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75. (Cited on pages 1 and 4.)

Hemel, Z., Groenewegen, D. M., Kats, L. C. L., and Visser, E. (2011). Static consistency checking of web applications with WebDSL. *Journal of Symbolic Computation*, 46(2):150–182. (Cited on page 12.)

Hemel, Z., Kats, L. C. L., Groenewegen, D. M., and Visser, E. (2010). Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling*, 9(3):375–402. (Cited on pages 12 and 47.)

Hemel, Z., Kats, L. C. L., and Visser, E. (2008a). Code generation by model transformation. A case study in transformation modularity. In Gray, J., Pierantonio, A., and Vallecillo, A., editors, *Theory and Practice of Model Transformations. First International Conference on Model Transformation (ICMT 2008)*, volume 5063 of *Lecture Notes in Computer Science*, pages 183–198, Heidelberg. Springer. (Cited on pages 12, 54, 69, 70, and 96.)

Hemel, Z., Verhaaf, R., and Visser, E. (2008b). WebWorkFlow: An object-oriented workflow modeling language for web applications. In Czarnecki, K., editor, *Model Driven Engineering Languages and Systems (MODELS 2008)*, Lecture Notes in Computer Science. Springer. (to appear). (Cited on page 12.)

Hemel, Z. and Visser, E. (2009). PIL: A platform independent language for retargetable DSLs. In van den Brand, M., Gasevic, D., and Gray, J., editors, *Software Language Engineering, Second International Conference, SLE 2009*, volume 5969 of *Lecture Notes in Computer Science*, pages 224–243. Springer. (Cited on page 12.)

Hemel, Z. and Visser, E. (2011). Declaratively programming the mobile web with mobl. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, Portland, Oregon, USA. ACM. (Cited on page 12.)

Hollingsworth, D. (1995). *The Workflow Reference Model*. Workflow Management Coalition, Document Number TC00-1003 - Issue 1.1 edition. (Cited on page 49.)

Huang, S. S. and Smaragdakis, Y. (2006). Easy language extension with Meta-AspectJ. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 865–868. ACM. (Cited on page 109.)

Janssen, N. (2005). Transformation tool composition. Master's thesis, Institute of Information and Computing Sciences Utrecht University, Utrecht, The Netherlands. (Cited on page 108.)

JetBrains (2009a). Intellij idea. `http://www.jetbrains.com/idea/`. (Cited on page 43.)

JetBrains (2009b). Web ide. http://www.jetbrains.com/webide. (Cited on page 43.)

Jouault, F. and Bézivin, J. (2006). KM3: a DSL for metamodel specification. In *Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *LNCS*, pages 171–185, Bologna, Italy. Springer. (Cited on page 106.)

Jouault, F., Bézivin, J., and Kurtev, I. (2006). TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Generative programming and component engineering (GPCE'06)*, pages 249–254. ACM. (Cited on page 106.)

Jouault, F. and Kurtev, I. (2006). Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer. (Cited on page 106.)

Kats, L. C. L., Bravenboer, M., and Visser, E. (2008). Mixing source and bytecode. A case for compilation by normalization. In Kiczales, G., editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, Nashville, Tenessee, USA. ACM Press. (Cited on pages 51, 93, and 105.)

Kats, L. C. L., Kalleberg, K. T., and Visser, E. (2009). Domain-specific languages for composable editor plugins. In Ekman, T. and Vinju, J., editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers. (to appear). (Cited on pages 45, 109, and 111.)

Kats, L. C. L. and Visser, E. (2010a). The spoofax language workbench. rules for declarative specification of languages and IDEs. In Rinard, M., editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*. (to appear). (Cited on pages 1 and 45.)

Kats, L. C. L. and Visser, E. (2010b). The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463. (Cited on pages 47 and 147.)

Kejriwal, A. A. and Bedekar, M. (2009). MobiDSL - a domain specific langauge for mobile web applications: developing applications for mobile platform without web programming. In *Proceedings of the 9th OOPSLA Workshop on Domain Specific Modelling (DSM'09)*. (Cited on page 172.)

Kelly, S., Lyytinen, K., and Rossi, M. (1996). MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In *CAiSE*, pages 1–21. (Cited on page 106.)

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling. Enabling Full Code Generation*. John Wiley & Sons, Inc. (Cited on page 69.)

Kim, J., Baratto, R. A., and Nieh, J. (2006). pthinc: a thin-client architecture for mobile wireless web. In *WWW*, pages 143–152. (Cited on page 143.)

Kittoli, S., editor (2008). *Seam - Contextual Components. A Framework for Enterprise Java.* Red Hat Middleware, LLC. (Cited on page 74.)

Klint, P. (1993). A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201. (Cited on pages 105 and 106.)

Koch, N., Kraus, A., and Hennicker, R. (2001). The authoring process of the uml-based web engineering approach. In *First International Workshop on Web-Oriented Software Technology*. (Cited on page 42.)

Kroiss, C., Koch, N., and Knapp, A. (2009). Uwe4jsf: A model-driven generation approach for web applications. In Gaedke, M., Grossniklaus, M., and Daz, O., editors, *Web Engineering, 9th International Conference, ICWE 2009, San Sebastin, Spain, June 24-26, 2009, Proceedings*, volume 5648 of *Lecture Notes in Computer Science*, pages 493–496. Springer. (Cited on page 42.)

Kulkarni, V. and Reddy, S. (2008). An abstraction for reusable mdd components: model-based generation of model-based code generators. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 181–184, New York, NY, USA. ACM. (Cited on page 110.)

Kurtev, I., Bézivin, J., Jouault, F., and Valduriez, P. (2006). Model-based DSL frameworks. In *Companion to OOPSLA'06*, pages 602–616. ACM. (Cited on page 106.)

Lai, A. M., Nieh, J., Bohra, B., Nandikonda, V., Surana, A. P., and Varshneya, S. (2004). Improving web browsing performance on wireless pdas using thin-client computing. In *WWW*, pages 143–154. (Cited on page 143.)

Lima, F. and Schwabe, D. (2003). Modeling applications for the semantic web. In Lovelle, J. M. C., Rodrguez, B. M. G., Aguilar, L. J., Gayo, J. E. L., and del Puerto Paule Ruz, M., editors, *Web Engineering, International Conference, ICWE 2003, Oviedo, Spain, July 14-18, 2003, Proceedings*, volume 2722 of *Lecture Notes in Computer Science*, pages 417–426. Springer. (Cited on page 42.)

Lovins, J. B. (1968). Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31. (Cited on page 151.)

Lussenburg, V., van der Storm, T., Vinju, J., and Warmer, J. (2010). Mod4j: A qualitative case study of model-driven software development. In *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin / Heidelberg. (Cited on page 4.)

M. Brambilla, S. Comai, P. F. and Matera, M. (2007). Designing web applications with WebML and WebRatio. In Rossi, G. et al., editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series. Springer. (Cited on page 110.)

Meijer, E., Beckman, B., and Bierman, G. M. (2006). Linq: reconciling object, relations and xml in the .net framework. In Chaudhuri, S., Hristidis, V., and Polyzotis, N., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706. ACM. (Cited on pages 41 and 153.)

Mens, T. and van Gorp, P. (2006). A taxonomy of model transformation. In *Graph and Model Transformation (GraMoT 2005)*, volume 152, pages 125–142. (Cited on page 107.)

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344. (Cited on page 1.)

Mesbah, A., Bozdag, E., and van Deursen, A. (2008). Crawling ajax by inferring user interface state changes. In *ICWE*, pages 122–134. (Cited on page 170.)

Meyerovich, L. A., Guha, A., Baskin, J. P., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). Flapjax: a programming language for ajax applications. In *OOPSLA*, pages 1–20. (Cited on page 172.)

Miller, J. and Mukerji, J. (2003). MDA guide version 1.0.1. (Cited on page 132.)

Miller, J. A., Palaniswami, D., Sheth, A. P., Kochut, K. J., and Singh, H. (1998). Webwork: Meteor2's web-based workflow management system. *J. Intell. Inf. Syst.*, 10(2):185–215. (Cited on page 64.)

Möller, A. and Schwarz, M. (2009). Jwig: Yet another framework for maintainable and secure web applications. In Filipe, J. and Cordeiro, J., editors, *WEBIST 2009 - Proceedings of the Fifth International Conference on Web Information Systems and Technologies, Lisbon, Portugal, March 23-26, 2009*, pages 47–53. INSTICC Press. (Cited on page 41.)

Muller, P.-A., Studer, P., and Bézivin, J. (2003). Platform independent web application modeling. In Stevens, P., Whittle, J., and Booch, G., editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 220–233. Springer. (Cited on page 133.)

Nilsson, H., Courtney, A., and Peterson, J. (2002). Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. (Cited on page 172.)

Nunes, D. A. and Schwabe, D. (2006). Rapid prototyping of web applications combining domain specific languages and model driven design. In Carr, L., Roure, D. D., Iyengar, A., Goble, C. A., and Dahlin, M., editors, *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 889–890. ACM. (Cited on page 42.)

O. Pastor, J. Fons, V. P. (2003). OOWS: A method to develop web applications from web-oriented conceptual models. In *Web Oriented Software Technology (IWWOST'03)*, pages 65–70. (Cited on page 110.)

Object Management Group (OMG) (2003). OMG/RFP/QVT MOF 2.0 query/views/transformations RFP. (Cited on page 108.)

Object Management Group (OMG) (2006). Meta object facility (MOF) core specification. OMG available specification. Version 2.0. `http://www.omg.org`. (Cited on page 106.)

P. Cáceres, E. Marcos, B. V. (2003). A MDA-Based approach for web information system development. In *Proceedings of Workshop in Software Model Engineering*. (Cited on page 110.)

Paakki, J. (1995). Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27(2):196–255. (Cited on page 111.)

Parr, T. J. (2004). Enforcing strict model-view separation in template engines. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 224–233, New York, NY, USA. ACM. (Cited on page 78.)

Parr, T. J. and Quong, R. W. (1994). ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810. (Cited on pages 114 and 132.)

Pastor, O., Fons, J., and Pelechano, V. (2003). OOWS: A method to develop web applications from web-oriented conceptual models. In *Web Oriented Software Technology (IWWOST'03)*, pages 65–70. (Cited on page 42.)

Peyton Jones, S., editor (2003). *Haskell98 Language and Libraries. The Revised Report*. Cambridge University Press. (Cited on page 104.)

Peyton Jones, S., Ramsey, N., and Reig, F. (1999). C–: A portable assembly language that supports garbage collection. In Nadathur, G., editor, *PPDP*, volume 1702 of *LNCS*, pages 1–28. Springer. (Cited on pages 9 and 131.)

Peyton Jones, S. L. and Santos, A. L. M. (1998). A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47. (Cited on page 104.)

Pierre-Alain Muller, Philippe Studer, F. F. and Bézivin, J. (2005). Platform independent web application modeling and development with Netsilon. *Software and Systems Modeling*, 4(4):424–442. (Cited on page 110.)

Plasmeijer, R., Achten, P., and Koopman, P. W. M. (2007). itasks: executable specifications of interactive work flow systems for the web. In Hinze, R. and Ramsey, N., editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 141–152. ACM. (Cited on page 64.)

Plotkin, G. D. (1975). Call-by-name, call-by-value and the lambda-calculus. *TCS*, 1(2):125–159. (Cited on page 153.)

Potel, M. (1996). MVP: Model-View-Presenter the taligent programming model for c++ and java. *Taligent Inc.* (Cited on page 144.)

Recker, J. and Strategy, M. (2006). Process Modeling in the 21 stCentury. *BPTrends, May*, pages 1–8. (Cited on page 64.)

Richards, M. (1971). The portability of the BCPL compiler. *Software - Practice and Experience*. (Cited on pages 9 and 131.)

Richardson, L. and Ruby, S. (2007). *RESTful Web Services*. O'Reilly. (Cited on page 143.)

Russell, N., Arthur, van der Aalst, W. M. P., and Mulyar, N. (2006). Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org. (Cited on pages 61 and 63.)

Russell, N., ter Hofstede, A., Edmond, D., and van der Aalst, W. (2005). Workflow Data Patterns. In *Proc. of 24th Int. Conf. on Conceptual Modeling (ER05)*, pages 353–368. (Cited on pages 61, 62, and 63.)

Sandholm, A. and Schwartzbach, M. I. (2000). A type system for dynamic web documents. In *POPL*, pages 290–301. (Cited on page 41.)

Sarkar, D., Waddell, O., and Dybvig, R. K. (2004). A nanopass infrastructure for compiler education. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 201–212, New York, NY, USA. ACM. (Cited on page 105.)

Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, 39(2):25–31. (Cited on page 69.)

Schwabe, D., Rossi, G., and Barbosa, S. D. J. (1996). Systematic hypermedia application design with oohdm. In *Hypertext 96, The Seventh ACM Conference on Hypertext, Washington DC, March 16-20, 1996*, pages 116–128. ACM. (Cited on page 42.)

Semenzato, L. (1993). The high-level intermediate language l. Technical Report UCB/CSD-93-760, EECS Department, University of California, Berkeley. (Cited on page 132.)

Smith, J. (2009). WPF Apps With The Model-View-ViewModel Design Pattern. `http://msdn.microsoft.com/en-us/magazine/dd419663.aspx`. (Cited on page 144.)

Spiewak, D. and Zhao, T. (2009). Scalaql: Language-integrated database queries for scala. In van den Brand, M., Gasevic, D., and Gray, J., editors, *Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, Lecture Notes in Computer Science. Springer. (Cited on page 41.)

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99. (Cited on page 1.)

Spinellis, D. and Guruprasad, V. (1997). Lightweight languages as software engineering tools. In *Proceedings of the Conference on Domain-Specific Languages, October 15-17, 1997, Santa Barbara, California, USA*, pages 67–76. USENIX. (Cited on page 1.)

Stahl, T., Voelter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons. (Cited on pages 116 and 135.)

Stahl, T. and Völter, M. (2005). *Model-Driven Software Development*. Wiley. (Cited on page 69.)

Steel, Jr., T. B. (1961). A first version of UNCOL. In *IRE-AIEE-ACM '61 (Western): Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 371–378, New York, NY, USA. ACM. (Cited on pages 9, 113, 121, and 131.)

Suzuki, J. and Yamamoto, Y. (1999). Extending UML with aspects: Aspect support in the design phase. *Lecture Notes in Computer Science*, pages 299–299. (Cited on page 110.)

Tatlock, Z., Tucker, C., Shuffelton, D., Jhala, R., and Lerner, S. (2008). Deep typechecking and refactoring. In Harris, G. E., editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 37–52. ACM. (Cited on page 43.)

ter Hofstede, A., Edmond, D., and van der Aalst, W. (2004). Workflow resource patterns. *BETA Working Paper Series*, pages 216–232. (Cited on pages 61 and 63.)

The Apache Foundation (2007). Velocity User Guide. `http://velocity.apache.org/engine/devel/user-guide.html`. (Cited on pages 78, 80, and 106.)

Thiemann, P. (2002). Wash/cgi: Server-side web scripting with sessions and typed, compositional forms. In Krishnamurthi, S. and Ramakrishnan, C. R., editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, volume 2257 of *Lecture Notes in Computer Science*, pages 192–208. Springer. (Cited on page 41.)

UCSD (1981). *UCSD p-System and UCSD PASCAL Users Manual*. SofTech Microsystems. (Cited on pages 121 and 131.)

Valderas, P., Pelechano, V., and Pastor, O. (2007). A transformational approach to produce web application prototypes from a web requirements model. *Int. J. Web Eng. Technol.*, 3(1):4–42. (Cited on page 42.)

van den Brand, M., Cornelissen, B., Olivier, P. A., and Vinju, J. J. (2005). Tide: A generic debugging framework - tool demonstration. *Electr. Notes Theor. Comput. Sci.*, 141(4):161–165. (Cited on page 131.)

van den Brand, M. G. J., de Jong, H., Klint, P., and Olivier, P. (2000). Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291. (Cited on page 106.)

van der Aalst, W. M. P., Hofstede, A. H. M. T., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51. (Cited on pages 51, 61, and 64.)

van der Aalst, W. M. P. and ter Hofstede, A. H. M. (2005). YAWL: yet another workflow language. *Information Systems*, 30(4):245–275. (Cited on pages 6, 49, and 63.)

van der Sluijs, K., Houben, G.-J., Broekstra, J., and Casteleyn, S. (2006). Hera-s: web design using sesame. In Wolber, D., Calder, N., Brooks, C. H., and Ginige, A., editors, *Proceedings of the 6th International Conference on Web Engineering, ICWE 2006, Palo Alto, California, USA, July 11-14, 2006*, pages 337–344. ACM. (Cited on page 42.)

van Deursen, A. and Klint, P. (1998). Little languages: little maintenance? *Journal of Software Maintenance*, 10(2):75–92. (Cited on page 1.)

van Deursen, A., Klint, P., and Tip, F. (1993). Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545. (Cited on pages 108 and 131.)

van Deursen, A., Klint, P., and Visser, J. (2000a). Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36. (Cited on pages 1 and 116.)

van Deursen, A., Klint, P., and Visser, J. (2000b). Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36. (Cited on page 135.)

van Wijngaarden, J. and Visser, E. (2003). Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University. (Cited on page 90.)

Vdovjak, R., Frasincar, F., Houben, G.-J., and Barna, P. (2003). Engineering semantic web information systems in hera. *J. Web Eng.*, 2(1-2):3–26. (Cited on page 42.)

Visser, E. (1997a). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam. (Cited on pages 1 and 131.)

Visser, E. (1997b). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam. (Cited on pages 30, 42, 45, 74, and 105.)

Visser, E. (2002). Meta-programming with concrete object syntax. In Batory, D., Consel, C., and Taha, W., editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer-Verlag. (Cited on pages 71, 79, and 80.)

Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C. et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag. (Cited on pages 1, 4, 71, and 77.)

Visser, E. (2005). A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873. Special issue on Reduction Strategies in Rewriting and Programming. (Cited on page 105.)

Visser, E. (2007a). Domain-specific language engineering. In Lämmel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, pages 265–318, Braga, Portugal. Universidade do Minho. International Summer School GTTSE 2007, Pre-Proceedings. (Cited on pages 15, 42, and 45.)

Visser, E. (2007b). WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373. (Cited on pages 145 and 171.)

Visser, E. (2008). WebDSL: A case study in domain-specific language engineering. In Lammel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer. (Cited on pages 4, 5, 26, 50, 52, 70, 74, 86, 114, and 118.)

Visser, E., Benaissa, Z.-e.-A., and Tolmach, A. (1998). Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP 1998)*, pages 13–26. ACM Press. (Cited on pages 33 and 81.)

Voelter, M. and Groher, I. (2007). Handling variability in model transformations and generators. In *Domain-Specific Modeling (DSM'07)*. (Cited on pages 78, 80, 106, and 110.)

W3C (2011). CSS 3 working draft. `http://www.w3.org/TR/css3-roadmap`. (Cited on page 168.)

Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. In *PLDI*, pages 242–252. (Cited on page 172.)

Warmer, J. and Kleppe, A. (2006). Building a flexible software factory using partial domain specific models. In *OOPSLA Int. Workshop on Domain-specific modeling*. (Cited on page 109.)

WfMC (1999). Terminology and glossary, 3rd edition. Document Number WFMC-TC-1011, Workflow Management Coalition. (Cited on page 49.)

White, S. (2004). Introduction to BPMN. *IBM Cooperation*. (Cited on page 64.)

Wu, H., Gray, J., and Mernik, M. (2008). Grammar-driven generation of domain-specific language debuggers. *Softw., Pract. Exper.*, 38(10):1073–1103. (Cited on page 131.)

Wyk, E. V., Krishnan, L., Bodin, D., and Schwerdfeger, A. (2007). Attribute grammar-based language extensions for Java. In Ernst, E., editor, *21st European Conference on Object-Oriented Programming (ECOOP 2007)*, volume 4609 of *Lecture Notes in Computer Science*, pages 575–599, Berlin, Germany. Springer. (Cited on page 111.)

Zook, D., Huang, S. S., and Smaragdakis, Y. (2004). Generating AspectJ programs with Meta-AspectJ. In Karsai, G. and Visser, E., editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 of *Lecture Notes in Computer Science*, pages 1–18. Springer. (Cited on page 109.)

Appendix: Consistency Checking in Web Application Frameworks

A

Appendix to Chapter 2.

A.1 DATA MODEL CONSISTENCY CHECKING

Web applications typically store data in a database. To simplify data persistence, the three frameworks abstract over database architectures, allowing developers to define a data model consisting of *entities* with *properties* and *relationships* between these entities. These can be one-to-one, one-to-many, or many-to-many relationships. In this section we study consistency checks of entity types, relations, and data validation constraints that may be specified for the data model.

A.2 CONSISTENCY OF PROPERTY TYPES

	M	**R**	**C**
Rails	Runtime	–	–
Seam	Compile	+	+
Lift	Compile	+	+

All three frameworks map their data models to relational databases by default. In relational databases, for each column in a table (i.e., each entity property) an existing type (i.e., a primitive type or the type of another entity) has to be specified.

In Ruby on Rails, entities, their properties, and types are defined in *database migration* scripts. Database migrations create the initial database and apply data migrations as the application evolves. In the following example, we define an migration that creates a `posts` table with three properties: `name` of type string, `title` of type string, and `content` of type text:

```
create_table :posts do |t|
  t.string :name
  t.string :title
  t.text :content
end
```

When a migration creates a property with an undefined type, (e.g. `t.strin`), no column is generated in the table for the property, nor is an error reported during the migration. The error is only detected when the property in question is *used* somewhere else in the application. Depending on the use of the property this may result in a range of errors, e.g. a `NoMethodError` is thrown when rendering an input control for the `:name` property. The error in does not lead back to the migration script in which this mistake was originally made, the error is therefore not only unclear, it is also not easily retraceable to the source of the problem.

In Seam and Lift, data models are defined as annotated Java/Scala classes, where entity properties are defined as fields. Consequently, when undefined property types are referenced, a compile-time error is reported by the Java or Scala compiler. The exact location of the error is clearly marked, and the error message – while not using the terms "entity" or "property" – is clear.

A.3 CONSISTENCY OF ENTITY RELATIONSHIPS

	M	**R**	**C**
Rails	Runtime	−	+/−
Seam	Deploy	+	+
Lift	Compile	+	+

To define relationships between two entities *A* and *B*, the data model must specify a property of type *B* in *A* and an *inverse* property that links entity *B* back to *A*. For instance, in the context of an online discussion board, a topic has many messages. A `Topic` entity would therefore define a `messages` property, and a `Message` entity a `topic` property, modeling the inverse of the relationship. This inverse property must explicitly specify the nature of this relationship (one-to-one, etc.).

In Rails, inverse properties are declared using `belongs_to`, `has_one`, `has_many` and `has_and_belongs_to_many` calls:

```
class Topic < ActiveRecord::Base
  has_many :messages
end
```

This example defines a one-to-many relationship from topics to messages. It implies there must be a `Message` entity, which has a field named `topic_id`, referring back to this topic.

Rails enforces the convention of naming inverse properties by pluralizing the entity they refer to: i.e., `Message` becomes `:messages`. When this convention is not followed, or when an entity is referred to that does not exist, no error is reported when the database is initialized or migrated. However, when the property is used, a `NoMethodError` is reported, tracing back the error to wherever the property was used rather than the entity declaration that was inconsistent.

In Seam, inverse columns are defined using the `@OneToMany` annotation (in case of a one-to-many relationship) specifying the inverse property with the `mappedBy` attribute:

```
@OneToMany(mappedBy="auction")
public Set<Bid> getAllBids() {
  return allBids;
}
```

If the `mappedBy` property does not exist or is misspelled (e.g. as `aauction` instead of `auction`), an exception occurs when the application is deployed.

While the actual error message tends to "drown" in an enormous stack trace, the actual message reported is accurate and specific:

```
mappedBy reference an unknown target entity property:
org.jboss.seam.example.seambay.Bid.aauction in
org.jboss.seam.example.seambay.Auction.allBids
```

While no line number or filename is supplied, a class name and property is supplied, which makes it relatively easy to find.

Lift has no support for persistent inverse properties. Instead, it allows inverse properties to be defined using a query:

```
def entries = Expense.findAll(By(Expense.account, this.id))
```

Any errors in the inverse property name (`Expense.account` in this case) are found at compile time are easy to trace back to the origin of the problem. The error message is generic, but clear.

A.3.1 *Consistency of Data Validation*

	M	**R**	**C**
Rails	Runtime	−	−
Seam	Compile/Deploy	+/−	+
Lift	Compile/Deploy	+/−	+

Most web frameworks allow developers to specify data validation constraints to validate user input. Examples of such constraints are constraints on the length of the input, or requiring a particular property to be set.

In Rails, validation constraints can be defined in entity classes. An example is the `validates_presence_of` construct, which defines that a field must be set:

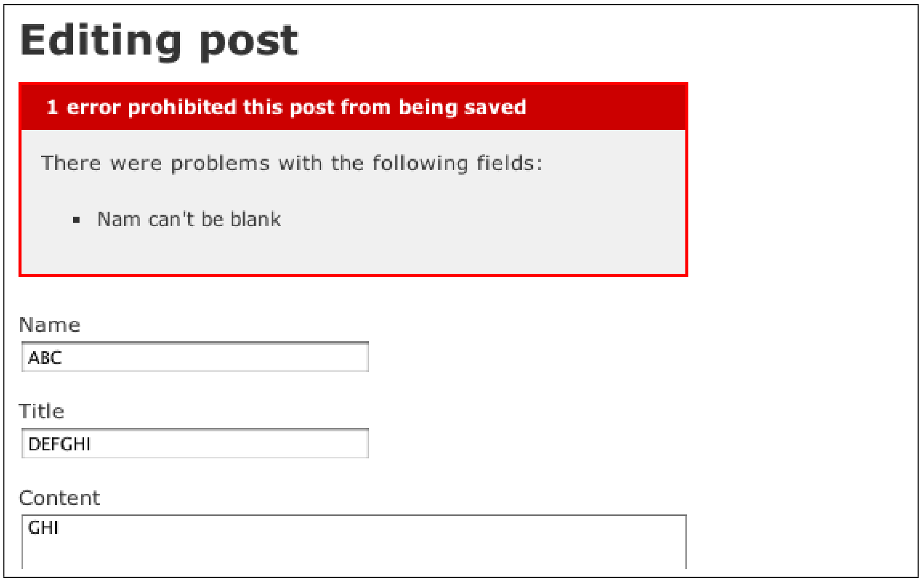
```
class Post < ActiveRecord::Base
  validates_presence_of :name
end
```

Constraint rules are not checked for validity but are still used in the user interface of an edit form in the application and when the application attempts to save an input. For example, if a presence constraint is specified for a non-existent property `nam`, Rails simply reports that property has not been set (see Figure A.1). As Rails fails to report an error directed to the developer about this problem, the message does not provide location information of the source of the problem and does not clearly state the underlying problem.

In Lift and Seam, property validation is defined using validator annotations that are mostly checked at compile time. However, certain types of validators, such as regular-expression validators require a regular expression to be encoded as a string. The following Seam example demonstrates this:

```
@Pattern(regex="^\\w*$", message="not a valid username")
public String getUsername() {
   return username;
}
```

A syntactically incorrect regular expression such as `^[\\w*$` is not detected


Editing post

1 error prohibited this post from being saved

There were problems with the following fields:

- Nam can't be blank

Name
ABC

Title
DEFGHI

Content
GHI

Figure A.1 Ruby on Rails validation error

at compile time. Instead, it is detected when the application is deployed, printing a long stack trace in which a `PatternSyntaxException` is reported. While the regular expression in question is printed, no indication is given about the location of the error.

A.4 USER INTERFACE CONSISTENCY CHECKING

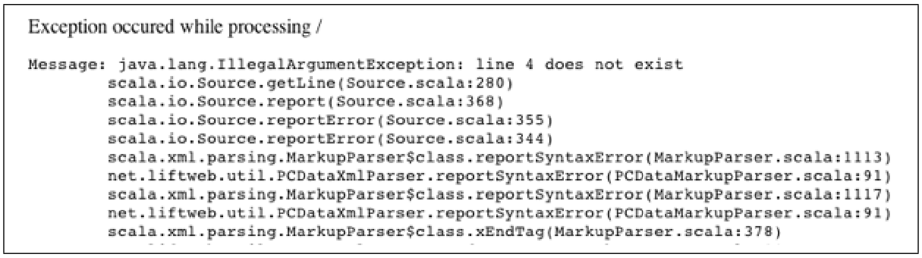
The user interface of web applications is generally implemented using a combination of HTML and CSS. All three frameworks leverage HTML directly to create the user interface. They do extend HTML with additional tags or escapes to the framework language. Proper (X)HTML has a strict syntax and clearly defines how page elements (tags) can be nested. However, browsers are very liberal when it comes to the interpretation of HTML. Therefore, faulty HTML code can result in surprising interpretations. By checking the validity of page elements and element nesting before a page is sent to a browser, interpretation problems can be avoided.

A.4.1 *Usage of Valid Page Elements*

	M	**R**	**C**
Rails	Runtime	+	–
Seam	Runtime	+	+
Lift	Runtime	–	–

While none of the frameworks check if used HTML tags are valid, they typically do perform checks on their own framework-specific extensions to HTML. This section focuses on these special page elements.

Rails' default template language ERB does not use standard XML-style tags



```
Exception occured while processing /

Message: java.lang.IllegalArgumentException: line 4 does not exist
        scala.io.Source.getLine(Source.scala:280)
        scala.io.Source.report(Source.scala:368)
        scala.io.Source.reportError(Source.scala:355)
        scala.io.Source.reportError(Source.scala:344)
        scala.xml.parsing.MarkupParser$class.reportSyntaxError(MarkupParser.scala:1113)
        net.liftweb.util.PCDataXmlParser.reportSyntaxError(PCDataMarkupParser.scala:91)
        scala.xml.parsing.MarkupParser$class.reportSyntaxError(MarkupParser.scala:1117)
        net.liftweb.util.PCDataXmlParser.reportSyntaxError(PCDataMarkupParser.scala:91)
        scala.xml.parsing.MarkupParser$class.xEndTag(MarkupParser.scala:378)
```

Figure A.2 Lift exception when opening invalid tag

for defining dynamic page elements, but instead uses escapes to Ruby code. The following code generates a link to another page:

```
<%= link_to "My Blog", posts_path %>
```

Using an undefined `linkto` page construct (instead of `link_to`) results in an undefined *method* error, instead of reporting an invalid *page element*. As Ruby simply checks for general errors instead of a domain-specific ones, a conceptual mismatch arises when reporting such errors. Still, the error does pinpoint exactly the line where the error occurs.

Seam uses XML tags to render controls and realize control flow within the user interface. The following code renders a label.

```
<h:outputLabel id="UsernameLabel" for="username">
  Login Name
</h:outputLabel>
```

When using an undefined page element, say `h:outputLabe` instead of `h:outputLabel`, the following error is reported when the user interface is loaded:

```
/home.xhtml @23,54 <h:outputLabe> Tag Library supports namespace:
  http://java.sun.com/jsf/html, but no tag was defined for name:
  outputLabe
```

While it is reported at run-time, the error provides a clear domain-specific error message and clear location of source of the error.

Lift, like Seam, uses XML tags to define dynamic page elements and page flow:

```
<lift:surround with="default" at="content">
  <h2>Welcome to your project!</h2>
  <p><lift:helloWorld.howdy /></p>
</lift:surround>
```

Using an undefined element `lift:surrond` instead of `lift:surround` can result in confusing errors as illustrated in Figure A.2. The error appears when the user interface is loaded, and cannot be traced back to its origin.

A.4.2 *User Interface Element Nesting*

While all three frameworks base their user interface specifications on HTML, they do not check HTML validity, i.e. the correctness of tags and their nesting. For instance, when a `<td>` tag is used outside a `<table>` tag, none of the frameworks report an error. When the page is loaded in the browsers, the invalid tag is simply ignored, a silent error. Unlike Rails, Lift and Seam do check whether the defined user interface is a well-formed XML document.

	M	**R**	**C**
Rails	Browser	–	–
Seam	Browser	–	–
Lift	Browser	–	–

A.4.3 *Consistency of References to the Data Model and to Pages*

We discuss consistency of references to data model entities and to other pages in Section 2.2.4.

A.4.4 *Consistency of Action and Controller Binding*

To submit information in a form, a target controller or action has to be specified to handle the action. The three frameworks handle this in different ways.

	M	**R**	**C**
Rails	Runtime	–	+
Seam	Runtime	+	–
Lift	Runtime	–	–

Rails provides a convenient way to generate a form at run-time that can be used to create or edit entities, using the `form_for` construct:

```
<% form_for(@post) do |f| %>
...
    <%= f.submit 'Update post' %>
<% end %>
```

This construct does not explicitly specify an action that should be used when the form is submitted. Instead, it follows the convention that entity controllers should have a `create` action for creating an entity, and an `update` action to edit it in case it already existed. An error is reported when submitting the form for an object for which the controller defines no `update` action (perhaps it provides a `modify` action instead). Rails then reports an unknown action error: “No action responded to update. Actions: create, destroy, edit, index, new, show, and modify.” Although no file or line number is provided, the error message is domain-specific and helpful.

Rails provides additional options when binding a form to an action. One option is the ability to let the user confirm the invocation of an action, e.g. when clicking a “Destroy” link. To this end, the `:confirm` keyword is used. However, when the `:confirm` keyword is mistyped (e.g., as `:confirmation`), Rails does not detect this in any way. The keyword is simply *ignored*, resulting in immediate deletion of the entry, without any confirmation:

```
<%= link_to 'Destroy', post, :confirmation => 'Are you sure?',
            :method => :delete %>
```

In Seam, the `commandButton` element links a form to controller actions:

```
<h:commandButton id="change" value="Change"
   action="#{changePassword.changePassword}"/>
```

As these elements are part of view templates, they are not checked at compile-time. Possible errors, such as links to undefined actions, are only detected at runtime, once the button is used. Using an undefined controller in the `action` attribute results in a Seam Debug screen; when scrolling down the actual exception can be seen:

```
Exception during request processing:
Caused by javax.servlet.ServletException with message:
    "#{changePassword.changePasswor}:
javax.el.MethodNotFoundException:
    /password.xhtml @37,91 action=
    "#{changePassword.changePasswor}":
Method not found: Proxy to
    jboss.j2ee:ear=jboss-seam-booking.ear,
    jar=jboss-seam-booking.jar,name=ChangePasswordAction,
    service=EJB3 implementing [interface
    org.jboss.seam.example.booking.ChangePassword]
    .changePasswor()"
```

The supplied `MethodNotFoundException` is hardly descriptive or domain-specific, but the error be traced back to its origin as the filename and line and column numbers are provided.

A.5 LOGIC CONSISTENCY CHECKING

The logic of a web application is typically defined in controllers, sometimes subdivided into actions. Like the user interface part of the web applications, controllers contain references to other parts of the applications, such as the user interface and data model. Consistency checking can ensure that these references are valid and remain valid as an application is changed.

A.5.1 *Consistency of Data Model References*

	M	**R**	**C**
Rails	Runtime	+	−
Seam	Compile	+	+
Lift	Compile	+	+

Controllers use references to the data model to persists data to the database, to read or write properties, or to perform queries.

In Rails, references to undefined entity types are reported as "uninitialized constant" errors when the code is invoked at runtime. The error exposes implementation details of the framework and can be confusing to developers, especially since the framework internally prefixes the entity name with the controller name.

For instance, when an undefined entity `E` is referenced from controller `C`, the following error is reported: "uninitialized constant C::E". Still, the accompanying stack trace refers back to the code in which the error occurred, so the error can be traced back to its source. Nonexistent properties are reported in a similar fashion, but identified as an "undefined method".

In Seam and Lift, controllers are written in Java and Scala, which are statically checked at compile time. References to undefined entity types are reported as "X cannot be resolved to a type". And non-existing properties in Scala are reported as "not a member of type X".

A.5.2 *Consistency of Redirects to Pages*

	M	**R**	**C**
Rails	Runtime	–	–
Seam	Runtime	–	–
Lift	Runtime	–	–

Similar to links in views, it is also common for controller code to redirect the user to a different page or controller.

In Rails, redirecting the user to different controllers and actions is done using `redirect_to`:

```
redirect_to :action => "index"
```

When an incorrect action name is used, for example by using `"home"` instead of `"index"`, an unexpected error occurs when the controller is invoked: "RecordNotFound" error: "Couldn't find Post with ID=home." Apparently, when an action is not defined, the action name is interpreted as an entity identifier in some cases. The error is reported as part of the `show` action of the controller, but there is no reference to the location of the actual error.

In Seam, redirects to pages are performed by returning the URL as the return value of an action:

```
return "/index.xhtml";
```

Lift has a `redirectTo` method for this purpose:

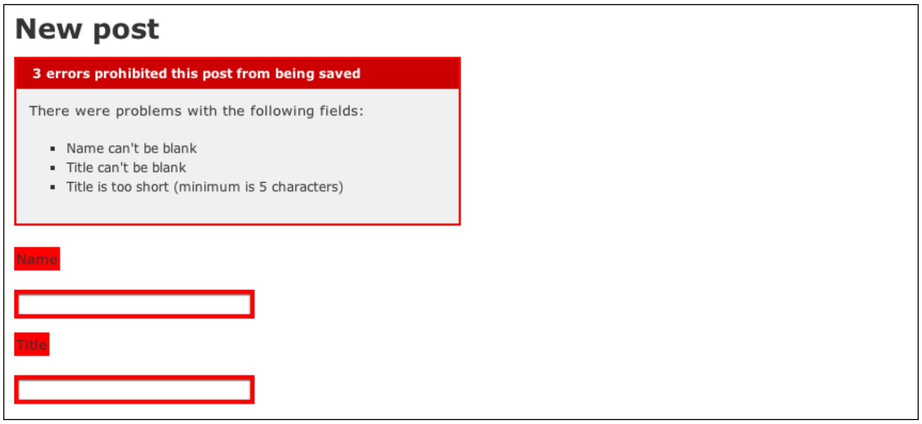
```
redirectTo("/index.html")
```

Similar to links in views, these redirects are not checked and specifying a redirect to an undefined page simply result in "404 not found" errors for the end-user.

A.5.3 *Consistency of Data Binding*

	M	**R**	**C**
Rails	Runtime	–	–
Seam	N/A	N/A	N/A
Lift	N/A	N/A	N/A

Forms can be used to create or modify entities in the data base. By specifying a *data binding* between form elements and entity properties, frameworks can directly interpret the results of a submitted form, creating or updating an entity.


New post

3 errors prohibited this post from being saved

There were problems with the following fields:

- Name can't be blank
- Title can't be blank
- Title is too short (minimum is 5 characters)

Figure A.3 Rails reports validation errors when making errors in data binding

In Rails, data can be bound to entities by passing the map containing the HTTP (POST/GET/PUT) request values to the constructor of a new object:

```
@post = Post.new(params[:post])
```

However, if a mistake is made in the expression, e.g. by inappropriately using :get when the form was changed to use a POST request or simply mistyping submit method, no error is reported. The result is that no data is bound to the properties of `@post` at all, often resulting in a validation error and empty input fields as can be seen in Figure A.3.

In Seam and Lift, controls are attached to an entity property and perform data binding themselves, they retrieve the value from the property and write back the value when a new value is entered. Therefore no data binding faults are possible, other than referring to non-existing properties and entities (discussed in Section A.5.1).

A.6 ACCESS CONTROL CONSISTENCY CHECKING

Access control can be used to restrict parts of web application to authenticated users. Access control rules that depend on the database (as they typically do) contain data model references that should be checked for consistency. Some frameworks allow access control rules to be defined separately from the user interface and controllers. Any bindings to pages and actions should also be checked for consistency. (We will not discuss these bindings here since they are treated in very similar to bindings from other parts of the application.)

A.6.1 *Consistency of Data Model References*

	M	**R**	**C**
Rails	Runtime	+	—
Seam	Runtime	-	—
Lift	Compile	+	+

Rails uses the `before_filter` construct to invoke a method before actions within a controller are invoked:

```
before_filter :authorize

def authorize
  auth_user = User.find_by_id(session[:user_id])
  unless auth_user && auth_user.age > 10
    redirect_to(:controller => "accessDenied",
                :action => "accessDenied")
  end
end
```

Errors are found when the authorization method is invoked, and are reported with clear indication of the source of the error.

In Seam, access control rules can be defined in a separate rule language:

```
rule CreateBlog
  no-loop
  activation-group "permissions"
when
  mbr: Member()
  acct: MemberAccount(member.memberId == mbr.memberId)
  check: PermissionCheck(target.memberId == mbr.memberId,
           action == "createBlog", granted == false)
then
  check.grant();
end
```

This DSL is not verified at compile-time. When a property is referred to that does not exist, e.g. `target.memberid` instead of `target.memberId`, the error is reported at the level of the page, instead of in the rule file: "RuntimeDroolsException: Exception executing predicate target.memberid == mbr.memberId." A location in the source code is supplied, but this refers to the location where the problem occurred, not the actual origin of the error: the rule file.

In Lift, access control rules are expressed using Scala expressions, in which invalid references to the data model are detected at compile time.

Samenvatting

METHODEN EN TECHNIEKEN VOOR ONTWERP EN IMPLEMENTATIE VAN DOMEIN-SPECIFIEKE TALEN

– Zef Hemel –

Software-ontwikkeling komt neer op het vertalen van eisen van gebruikers naar een computerprogramma — in weze machine-instructies gecodeerd als 1-en en 0-en, die het computerprogramma vormen dat voldoet aan de eisen van de gebruiker. Deze vertaalslag hoeft echter niet in een keer gemaakt te worden. Tussen gebruikerseisen en machine-instructies zijn in de loop de jaren een aantal lagen van abstractie bedacht die het proces versimpelen.

Een programmeertaal is zo'n abstractielaag. Een programmeertaal is een formele taal waarvan uitdrukkingen door een machinale vertaler (een *compiler*), te vertalen zijn naar machine-instructies, en daarnaast goed leesbaar zijn door de (getrainde) mens. Vergeleken met menselijke taal is een programmeertaal veel beperkter en gestructureerder.

De afgelopen decennia zijn er veel programmeertalen in gebruik genomen, waaronder C, C++, Java en Python. Dit zijn voorbeelden van algemeen toepasbare programmeertalen — programmeertalen die geschikt zijn om elk soort software te bouwen: van tekstverwerker tot software voor kernreactoren. Echter, er is ook een ander soort taal: de *domein-specifieke taal*. Een domein-specifieke taal is een programmeertaal die zich, zoals de naam al zegt, richt op één specifiek domein. Zo'n taal is bijvoorbeeld zeer geschikt om bedrijfsadministratiesoftware mee te maken, of webapplicaties, of software voor mobiele telefoons.

Doordat de taal zich op een beperkter domein richt, kan de taal expressiever gemaakt worden. Het expressiever maken van een taal zie je in de menselijke taal terug in de vorm van *vakjargon*. In verschillende bedrijfstakken krijgen complexe concepten een korte, bondige naam, die iedereen binnen het domein kent. Door het gebruik van jargon wordt de communicatie efficiënter. Dit is precies het idee achter domein-specifieke talen: door een kleine taal te ontwikkelen, gebruik makend van het vakjargon van het domein, kunnen domeinexperts sneller software ontwikkelen. Omdat deze programma's compacter zijn worden er minder fouten gemaakt, en eventuele fouten kunnen sneller gevonden en duidelijker gerapporteerd worden omdat de vertaler meer kennis heeft over het domein.

In dit onderzoek is bekeken hoe dergelijke domein-specifieke talen ontworpen en geimplementeerd kunnen worden. Dit is gedaan aan de hand van twee talen: WebDSL en Mobl.

ONTWERP

Met name in het kader van webapplicatie ontwikkeling worden veel domein-specifieke talen gebruikt, talen zoals HTML om de structuur van een webpagina te definiëren en CSS om de *lay-out* van de pagina te beschrijven. Daarnaast wordt er vaak gebruik gemaakt van een databank die bevraagd wordt met nog een andere domein-specifieke taal: SQL. Deze wirwar van verschillende talen zorgt ervoor dat het vinden van fouten lastig is en het systeem in de praktijk continu getest moet worden — vaak met de hand — om fouten te vinden. In Hoofdstuk 2 beschrijven we een alternatieve aanpak: de geïntegreerde domein-specifieke taal, waarin alle sub-talen die samen een groter domein beschrijven, geïntegreerd worden in een grote taal. We laten we zien hoe hierdoor gehele programma's, terwijl deze ontwikkeld worden, automatisch geanalyseerd kunnen worden, waardoor fouten eerder zichtbaar zijn.

WebDSL is zo'n geïntegreerde taal gericht op het snel kunnen ontwikkelen van webapplicaties waar data centraal staat. Voorbeelden variëren van een facturatiesysteem tot een sociaal netwerk. WebDSL kent het jargon van het webdomein, waardoor WebDSL programma's door middel van concepten als pagina's, templates, om een pagina uit samen te stellen, en entiteiten, om de datastructuur te definiëren, beschreven kunnen worden. Een domein waar webapplicaties steeds vaker ingezet worden is bedrijfssoftware. Bedrijfssoftware heeft het als doel het bedrijfsproces te ondersteunen. Dergelijke software kan al met WebDSL beschreven worden, maar dit vereist nog betrekkelijk veel code. Hoofdstuk 3 beschrijft daarom een uitbreiding van WebDSL, genaamd *WebWorkFlow*, die extra jargon toevoegt om op een compacte manier processen te beschrijven. Alhoewel er andere talen bestaan met hetzelfde doel, ondersteunt WebWorkFlow de beschrijving van de gehele applicatie op een flexibele manier.

IMPLEMENTATIE

Hoofdstuk 4 bekijkt hoe de vertaling van domein-specifieke code (bijvoorbeeld WebDSL) naar uitvoerbare code (bijvoorbeeld Java) beschreven kan worden. Hiervoor zijn een aantal technieken ontwikkeld waarbij door middel van modeltransformatietechnieken een vertaler op een nette en schaalbare manier gestructureerd kan worden.

Een van de voordelen van een domein-specifieke taal is dat deze software kan beschrijven op een platform-onafhankelijke manier. Daardoor zou een programma zowel naar, bijvoorbeeld, het Microsoft Windows platform als het Macintosh platform vertaald kunnen worden. Hoofdstuk 5 beschrijft een manier om code te vertalen naar veschillende platformen. Het beschrijft hoe een vertaler zo gestructureerd worden dat het makkelijker wordt code voor meerdere platformen tegelijk te genereren. De bedachte oplossing heet *PIL*, een platform-onafhankelijke tussentaal, die als tussenstap dient.

EEN NIEUW DOMEIN

Ten slotte worden in Hoofdstuk 6 de ontwikkelde technieken en methoden toegepast op een nieuw domein: het domein van applicaties voor moderne *smartphones*, zoals de iPhone en Android telefoons. Dit domein kampt met vergelijkbare problemen als het webdomein: applicaties worden gebouwd met meerdere domein-specifieke talen die slecht geïntegreerd zijn. Daarom beschrijft dit hoofdstuk een nieuwe domein-specifieke taal: Mobl. Mobl leent veel ideeën van WebDSL, maar werkt toch anders omdat de applicatie in zijn geheel op een telefoon draait in plaats van een machine op het internet, daarnaast heeft een mobiele telefoon veel beperkingen ten op zichte van een normale computer, bijvoorbeeld een kleiner scherm en een onbetrouwbare internet verbinding. Mobl lost deze problemen op door een aantal reeds bestaande programmeerparadigma's op een elegante manier te integreren.

RESULTAAT

Naast een aantal wetenschappelijke publicaties en nieuwe inzichten in het ontwikkelproces van domein-specifieke talen, heeft het onderzoek geresulteerd in een substantiële bijdrage aan de ontwikkeling van WebDSL en in de nieuwe taal Mobl. WebDSL wordt al gebruikt om een aantal nuttige web applicaties te bouwen, zoals Researchr[1] — een applicatie om bibliografie informatie mee te beheren (o.a. gebruikt voor de bibliografie van dit proefschrift) — en YellowGrass[2] — een applicatie om softwareprojecten mee te beheren, o.a. gebruikt voor veel projecten binnen de onderzoeksgroep. Mobl, een wat jonger project, heeft inmiddels een enthousiaste groep van meer dan honderd gebruikers vanuit de industrie. Meerdere bedrijven zijn bezig hun mobiele applicaties te ontwikkelen met behulp van Mobl.

[1] `http://researchr.org`

[2] `http://yellowgrass.org`

Curriculum Vitae

Zef Hemel

22 june 1983
Born in Groningen

1995–2001
VWO diploma
Belcampo College in Groningen
Nature & Technology profile

2001–2005
B.Sc. in Computer Science
University of Groningen
Department of Mathematics and Computing Science

2005–2006
M.Sc. in Computer Science
Trinity College, Dublin
Department of Computer Science

2007–2011
Ph.D. in Computer Science
Delft University of Technology
Department of Software Technology

2011–present
Senior Developer
Cloud9 IDE, Inc.

Titles in the IPA Dissertation Series Since 2005

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of As-*

pects. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03