

Delft University of Technology
Software Engineering Research Group
Technical Report Series

WebDSL: A Case Study in Domain-Specific Language Engineering

Eelco Visser

Report TUD-SERG-2008-023



TUD-SERG-2008-023

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
<http://www.se.ewi.tudelft.nl/techreports/>

For more information about the Software Engineering Research Group:
<http://www.se.ewi.tudelft.nl/>

This paper is a pre-print of:

E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. Tutorial for International Summer School GTTSE 2007

```
@InProceedings{Vis08,  
  author = {Eelco Visser},  
  title = {{WebDSL}: {A} Case Study in Domain-Specific Language  
          Engineering},  
  booktitle = {Generative and Transformational Techniques in  
              Software Engineering (GTTSE 2007)},  
  publisher = {Springer},  
  year = 2008,  
  editor = {R. Lammel and J. Saraiva and J. Visser},  
  series = {Lecture Notes in Computer Science}  
}
```

This technical report subsumes TUD-SERG-2007-017

© copyright 2008, Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

WebDSL: A Case Study in Domain-Specific Language Engineering

Eelco Visser

Software Engineering Research Group
Delft University of Technology
visser@acm.org

Abstract. The goal of domain-specific languages (DSLs) is to increase the productivity of software engineers by abstracting from low-level boilerplate code. Introduction of DSLs in the software development process requires a smooth workflow for the production of DSLs themselves. This requires technology for designing and implementing DSLs, but also a methodology for using that technology. That is, a collection of guidelines, design patterns, and reusable DSL components that show developers how to tackle common language design and implementation issues. This paper presents a case study in domain-specific language engineering. It reports on a project in which the author designed and built WebDSL, a DSL for web applications with a rich data model, using several DSLs for DSL engineering: SDF for syntax definition and Stratego/XT for code generation. The paper follows the stages in the development of the DSL. The contributions of the paper are three-fold. (1) A tutorial in the application of the specific SDF and Stratego/XT technology for building DSLs. (2) A description of an incremental DSL development process. (3) A domain-specific language for web-applications with rich data models. The paper concludes with a survey of related approaches.

1 Introduction

Abstraction is the key to progress in software engineering. By encapsulating knowledge about low level operations in higher-level abstractions, software developers can think in terms of the higher-level concepts and save the effort of composing the lower-level operations. By stacking layers of abstraction, developers can avoid reinventing the wheel in each and every project. That is, after working for a while with the abstractions at level n , patterns emerge which give rise to new abstractions at level $n + 1$.

Conventional abstraction mechanisms of general purpose programming languages such as methods and classes, are no longer sufficient for creating new abstraction layers [32, 82]. While libraries and frameworks are good at encapsulating functionality, the language which developers need to use to reach that functionality, i.e. the application programmers interface (API), is often awkward. That is, utterances take the form of (complex combinations of) method calls. In some cases, an API provides support for a more appropriate language,

but then utterances take the form of string literals that are passed to library calls (e.g. SQL queries) and which are not checked syntactically, let alone semantically, by the host language. Application programs using such frameworks typically consist of large amounts of boilerplate code, that is, instantiations of a set of typical usage patterns, which is needed to cover the variation points of the framework. Furthermore, there is often a considerable distance between the conceptual functionality of an application and its encoding in the program code, leading to disproportionate efforts required to make small changes. The general-purpose host language of the framework has no knowledge of its application domain, and cannot assist the developer with for instance verification or optimization.

In recent years, a number of approaches, including model-driven architecture [76], generative programming [33, 32], model-driven engineering [61, 82], model-driven software development [87], software factories [51, 30], domain-specific modeling [60], intentional software [84], and language oriented programming [36], have been proposed that aim at introducing new *meta-abstraction mechanisms* to software development. That is, mechanisms that enable the creation of new layers of abstraction.

Domain-Specific Languages Common to all these approaches is the encapsulation of design and implementation knowledge from a particular application or technical domain. The commonalities of the domain are implemented directly in a conventional programming language or indirectly in code generation templates, while the variability is configurable by the application developer through some configuration interface. This interface can take the form of a wizard for simple domains, or full fledged languages for domains with more complex variability [32]. Depending on the approach, such languages are called *modeling languages*, *domain-specific languages*, or even *domain-specific modeling languages*.

In this paper the term *domain-specific language* is used with the following definition:

A domain-specific language (DSL) is a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain.

Lets examine the elements of this definition:

A DSL is a *language*, that is, a collection of sentences in a textual or visual notation with a formally defined syntax and semantics. The structure of the sentences of the language should be defined by means of a grammar or meta-model, and the semantics should be defined by means of an abstract mathematical semantics, or by means of a translation to another language with a well understood semantics. Thus, the properties and behavior of a DSL program or model should be predictable.

A DSL is *high-level* in the sense that it abstracts from low-level implementation details, and possibly from particularities of the implementation platform. High-level is a matter of perspective, though. Algol was introduced as a language

for the *specification of algorithms* [8] and was high-level with respect to assembly language. Now we consider the Algol-like languages such as C and Java as low-level implementation languages.

A DSL should support *software implementation*. This does not require that a DSL be a procedural language, like many familiar programming languages. Indeed, declarative DSLs are preferable. However, DSLs should contribute in the creation of components of executable software systems. There are many examples of declarative languages that specify computations. For example, a context-free grammar does not consist of instructions to be executed ('directly') by a computer. Rather it is a declarative definition of the sentences of a language. Yet a grammar may also be used to generate an executable parser for that language.

Finally, the concepts and abstractions of a DSL are *related to a particular domain*. This entails that a DSL does not attempt to address all types of computational problems, or not even large classes of such problems. This allows the language to be *very expressive* for problems that fall in the domain and completely useless for other problems. For problems that are on the edge of the domain (as perceived by the DSL designer), the language may not be adequate. This gray area typically leads to pressure for the DSL to grow beyond its (original) domain. What makes a suitable domain cannot be determined in general; the closest we can get is maybe the circular definition that a domain is a coherent area of (software) knowledge that can be captured in a DSL.

The success of a DSL is measured in terms of the improvement of the software development process it enables. First, it is important that the DSL is actually effective in its intended domain, that is, applications that are considered to fit the domain should be expressible with the DSL¹. This can be expressed as the *completeness* of the DSL or its *coverage* of the domain. Next, building an application with a DSL should take substantially less *effort* than with other means. An approximation of this metric, is the number of DSL *lines of code* (LOC) that is needed for an application compared to what would be needed with conventional programming techniques. An *expressive* DSL requires few lines of code. There is a natural tension between coverage and expressivity. *Non-functional requirements* are just as important as functional requirements. In addition to providing the required functionality, a system should be efficient, safe, secure, and robust, to the extent required. Finally, first-time development of applications may be cheap, but systems usually have a long life span. The question then is how well the DSL supports *maintenance* and how *flexible* it is in supporting new requirements. Van Deursen and Klint [98] discuss maintainability factors of DSLs.

History Domain-specific languages pre-date the recent modeling approaches mentioned above by decades. The name of the programming language for scientific computing FORTRAN, developed by Backus in the late 1950s, is an ab-

¹ 'Application' can be read either as a complete software system or as a component of a software system; DSLs do typically not address all aspects of a software system.

abbreviation of 'formula translation' [7]. The language borrowed notation from mathematics so that programmers could write mathematical formulas directly, instead of encoding these in low-level stack and register operations, resulting in a dramatic improvement of programmer productivity. The 1970s Structured Query Language (SQL) [24] provided special notation for querying databases based on Codd's [27] relational database model. So called *little languages* [12] prospered in the Unix environment. Languages such as LEX (lexical analysis), YACC (parsing), PIC (for drawing pictures), and Make (for software building) were developed in the 1970s and 1980s. Another strand in the history are the so called *fourth generation languages* supported by application generators [91], which were supposed to follow-up the third generation general purpose languages. There are several surveys of domain-specific languages, including [86, 85, 98, 99, 73].

Textual vs Visual One aspect of the recent modeling approaches that could be perceived as novel is the preference for visual (graphical) languages in many approaches. For example, model-driven architecture and its derivatives are largely based on the use of UML diagrams to model aspects of software systems. Using UML profiles, the general purpose UML can be used for domain-specific modeling. MetaCase [60] and the Visual Studio DSL Tools [30] provide support for defining domain-specific diagram notations. There is no fundamental difference in expressivity between visual and textual languages. The essence of a language is that it defines structures to which meaning is assigned. Viewing and creating these structures can be achieved with a variety of tools, where various representations are interchangeable. On the one hand, visual diagrams can be trivially represented using text, for instance by taking an XML rendering of the internal structure. On the other hand, textual models can be trivially represented 'visually' by displaying the tree or graph structure resulting from parsing followed by static semantic analysis. Of course, there are non-trivial visualizations of textual models that may provide an alternative view. Some notations are more appropriate for particular applications than others. However, most (successful) DSLs created to date are textual, so text should not be easily discarded as a medium. Another factor is the impact on tools required for viewing and creating models.

Systematic Development Rather than a preference for visual languages, more significant in recent approaches is the emphasis — with support from industry (e.g. Microsoft) and standardization organizations (e.g. OMG) — on the *systematic development and deployment* of DSLs in the software development process. While the DSLs and 4GLs of the past were mostly designed as one-off projects by a domain stakeholder or tool vendor, DSLs should not just be *used* during the software development process, but the *construction* of DSLs should also become part of that process. Where developers (or communities of developers across organizations) see profitable opportunities for linguistic abstraction, new DSLs should be developed. Rather than language design artistry, this requires a solid engineering discipline, which requires an effective collection of techniques

and methods for developing domain-specific languages. In their survey of DSL development methods, Mernik et al. [73] describe patterns for decision, analysis, design, and implementation of DSLs. They conclude that most existing work focuses on supporting the implementation of DSLs, but fails to provide support, be it methodological or technological, for earlier phases in the DSL life cycle. Thus, a challenge for a software engineering discipline in which DSLs play a central role is a systematic and reproducible DSL development methodology. As for the *use* of DSLs, important criteria for the effectiveness of such a methodology are the effort it takes to *develop* new DSLs and their subsequent *maintainability*.

In previous work I have focused on the creation of language implementation technology, that is, a set of DSLs and associated tools for the development and deployment of language processing tools. The SDF syntax definition formalism [53, 101], the Stratego/XT program transformation language and tool set [103, 17, 19], and the Nix deployment system [39, 37] provide technology for defining languages and the tools needed for their operation. Publications resulting from this research typically present innovations in the technology, illustrated by means of case studies. This paper for a change does not present technological innovations in meta technology, but rather an application of that technology in domain-specific language engineering, with an attempt at exploring the design space of DSL development methodology.

WebDSL This paper presents a case study in domain-specific language engineering. The paper tracks the design and implementation of WebDSL, a DSL for web applications with a rich data model. The DSL is implemented using Stratego/XT and targets high-level Java frameworks for web engineering. The contributions of this paper are

- A tutorial on DSL design, contributing to the larger goal of building a methodology for the design and implementation of domain-specific languages. This includes an incremental (agile) approach to analysis, design, and implementation, and the illustration of best practices in language design, such as the use of a core language and the introduction of syntactic abstractions to introduce higher-level abstractions.
- A tutorial on the application of Stratego/XT to building (textual) domain-specific languages, illustrating the utility of techniques such as term rewriting, concrete object syntax, and dynamic rewrite rules.
- The introduction of WebDSL, a domain-specific language for the implementation of web applications with a rich data model.

The next section describes the development process and introduces the setup of sections 3 to 9, which discuss the stages in the development of WebDSL. Sections 10 to 12 evaluate the resulting WebDSL language and its development process, also with respect to related work.

2 Process Definition and Domain Analysis

According to the DSL development patterns of Mernik et al. [73], the DSL life cycle consists of (1) a decision phase in which the decision whether or not to build a DSL is taken, (2) an analysis phase in which the application domain is analyzed, (3) a design phase in which the architecture and language are designed, and finally, (4) an implementation phase in which the DSL and supporting run-time system are constructed. We can add (5) a deployment phase, in which DSLs and the applications constructed with them are used, and (6) a maintenance phase in which the DSL is updated to reflect new requirements. In this paper, I propose an incremental, iterative, and technology-driven approach to DSL development in which analysis, design, and implementation are combined in the spirit of agile software development [11]. Deployment and maintenance are left for future work. In this section, I describe and motivate this process model and relate it to the patterns of Mernik et al. [73]. The bulk of the paper will then consist of a description of the iterations in the design of WebDSL.

2.1 When to Develop a DSL?

The development of a DSL starts with the decision to develop one in the first place. Libraries and frameworks form a good alternative for developing a DSL. Many aspects of application development can be captured very well in libraries. When a domain is so fresh that there is little knowledge about it, it does not make sense to start developing a DSL. First the regular software engineering process should be applied in order to determine the basic concepts of the field, develop a code base supported with libraries, etc. When there is sufficient insight in the domain and the conventional programming techniques fail to provide the right abstractions, there may be a case for developing a DSL. So, what were the deciding factors for developing WebDSL?

The direct (personal) inspiration for developing WebDSL are wiki systems such as MediaWiki used for wikipedia, and more concretely TWiki used for program-transformation.org and other web sites maintained by the author. Wikis enable a community — the entire web population or the members of an organization — to contribute to the content of a site using a browser as editor. However, the data model for that content is poor, requiring all structured information to be encoded in the text of a page. This lack of structure entails that querying data and data validation depend on text operations. The initial goal of WebDSL is to combine the flexible, online editing of content as provided by wikis with a rich data model that allows presentation of and access to the underlying data in a variety of ways.

The scope of WebDSL is *interactive dynamic web applications with a rich application-specific data model*. That is, web applications with a database for data storage and a user interface providing several views on the data in the database, but also the possibility to modify those data via the browser. An additional assumption is that the data model is static, i.e. it is designed during development and cannot be changed online.

The engineering of web applications is a fairly mature field. There is an abundance of libraries and frameworks supporting the construction of web applications. The state-of-the art for the construction of robust industrial strength web applications are the Java and C# web engineering platforms. Based on the portability of Java and the availability of infrastructure for generation of Java in Stratego/XT, I have decided to restrict my attention to this platform for this case study. While current frameworks provide good support for the basic mechanics of web applications — such as handling requests, parsing form data, and producing XHTML — there is a strong case for the development of a DSL for this domain; several of the decision patterns of Mernik et al. [73] apply to the domain of web applications.

Task Automation Compared to the CGI programming of early web applications, a mature web engineering platform takes care of low-level concerns. For example, Java servlets deal with the mechanics of receiving requests from and sending replies to clients. Java Server Faces (JSF) deal with the construction of web pages *and* with the analysis of form data received from the client. Despite such facilities, web programming often requires a substantial amount of boilerplate code; many Java classes or XML files that are very similar, yet not exactly the same either. Conventional abstraction mechanisms are not sufficient for abstracting over such patterns. Thus, one case for a web DSL is programming-*task automation*, i.e. preventing the developer from having to write and maintain boilerplate code.

Notation The current platform provides an amalgam of often verbose languages addressing different concerns, which are not integrated. For example, the Java-JPA-JSF-Seam platform is a combination of XHTML extended with JSF components and EL expressions (Java-*like* expressions embedded in XML attributes), Java with annotations for declaration of object-relational mapping and *dependency injection*, and SQL queries ‘embedded’ in Java programs in the form of string literals. A concise and consistent notation, that linguistically integrates the various aspects of web application construction would lighten development and maintenance. Note that linguistic integration does not necessarily mean a loss of separation of concerns, but rather that different concerns can be expressed in the same language.

Verification Another consequence of the lack of integration of web application technologies is the lack of static verification of implementations. Components linked via dependency injection are only checked at run-time or deployment-time. Queries embedded in strings are not checked syntactically or for compatibility with the data model until run-time. References in EL expressions in XHTML files are only checked at run-time. These issues clearly illustrate that the abstraction limits of GPLs have been reached; the static typechecking of Java programs does not find these problems. A static verification phase, which would be enabled by an integrated language would avoid the tedious debugging process that these problems cause.

GUI Construction The user interface portion of a web application is typically defined by means of a template mechanism. JSP-style templates consist of plain text with anti-quotations in which fragments of Java code are used to insert ‘dynamic’ content derived from data objects. The framework has no knowledge of the structure of the HTML code generated by the template, so it is very easy to generate non well-formed documents. Java Server Faces templates are more advanced in that they define the complete document by means of a structured XML document, which is parsed at deployment-time. XHTML is generated by rendering this structure. Insertion of content from data object is achieved by means of ‘EL expressions’ in XML attributes. Still, templates are very verbose and concerned with low-level details. Furthermore, the EL expressions are only parsed and checked at run-time.

Analysis and Optimization There are also opportunities for domain-specific analysis and optimization. For example, optimization of database queries in the style of Wiedermann and Cook [108] might be useful in improving the performance of applications without resorting to manual tuning of generated queries. These concerns are not (yet) addressed in WebDSL.

2.2 Domain Analysis

Domain analysis is concerned with the analysis of the basic properties and requirements of the problem domain. For example, a first analysis of the domain would inform us that the development of a web application involves a data model, an object-relational mapping, a user interface, data input and output methods, data validation, page flow, and access control. Additionally, it may involve file upload, sending and receiving email, versioning of data, internationalization, and higher-level concerns such as work-flow. A more thorough analysis studies each of the concerns of a domain in more detail, and establishes terminology and requirements, which are then input for the design of a DSL.

Deductive The traditional development process for domain-specific languages follows a top-down or deductive track and starts with an exhaustive domain analysis phase, e.g. [29, 98, 73]. The advantage of this approach is a thorough analysis. The risk of such a deductive (top-down) approach is that the result is a language that is difficult to implement. Furthermore, a process developing an all encompassing DSL for a domain runs the usual risks of top-down design, such as over design, late understanding of requirements, leading to discovery of design and implementation problems late in the process.

Inductive Rather than designing a complete DSL before implementation, this paper follows an inductive approach by incrementally introducing abstractions that allow one to capture a set of common programming patterns in software development for a particular domain. This should enable a quick turn-around time for the development of such abstractions. Since the abstractions are based

on concrete programming patterns, there are no problems with implementing them.

Technology-driven Rather than designing a DSL based on an analysis of the domain in the abstract, the approach is *technology-driven*, i.e. considers best practices in the implementation of systems in the domain. This is similar to *architecture-centric* model-driven software development [87] or designing DSLs based on a *program family* [28]. After the initial determination of the scope of the domain, domain analysis then is concerned with exploring the technology that is available, and analyzing how it is typically used.

Selecting a specific technology helps in keeping a DSL design project grounded; there is a specific reference architecture to target in code generation. However, a risk with this approach is that the abstractions developed are too much tied to the particularities of the target technology. In domains such as web applications there are many *virtual machines*. Each combination of implementation languages, libraries, and frameworks defines a virtual machine to target in software development. Each enterprise system/application may require a different virtual machine. This is similar to the situation in embedded systems, where the peculiarities of different hardware architectures have to be dealt with. Thus, a consideration for the quality of the resulting DSL is the amount of leakage from the (concrete) target technology into the abstractions of the DSL; how easy is it to port the DSL to other virtual machines?

Iterative Developing the DSL in iterations can mitigate the risk of failure. Instead of a big project that produces a functional DSL in the end, an iterative process produces useful DSLs for sub-domains early on. This can be achieved by extending the coverage of the domain incrementally. First the domain concerns addressed can be gradually extended. For example, the WebDSL project starts with a data model DSL, addressing user interface issues only later in the project. Next, the coverage within each concern does not have to be complete from the start either. The WebDSL coverage of user interface components is modest at first, concentrating on the basic architecture, rather than covering all possible fancy features. This approach has the advantage that DSLs for relevant areas of the domain are available early and can start to be used in development. The feedback from applying the DSL under development can be very valuable for evaluating the design of abstractions and improving them. Considering the collection of patterns will hopefully lead to a deeper insight in how to make even better abstractions for the application domain.

2.3 Outline

The rest of this paper discusses the iterations in the design and implementation of WebDSL. These iterations are centered around three important DSL design patterns: *finding programming patterns*, *designing a core language*, and *building syntactic abstractions* on top of the core language.

Programming Patterns The first step in developing a new DSL is to explore the technology for building systems in the domain to find common programming patterns. That is, program fragments that occur frequently with slight variations. This exploration can take the form of inspecting legacy code, but preferably the technical literature and reference implementations. These typically present ideal programming patterns, as opposed to legacy code exposed to design erosion. The idea then is to capture the variability in the patterns by an appropriately designed abstraction. The commonality in the patterns is captured in code templates used in the generator that translates the abstractions to target code.

In Sections 3 to 5 we explore the domain of web applications built with Java/JSF/JPA/Seam and the techniques for implementing a DSL for this domain. Section 3 starts with looking at programming patterns for the implementation of *data models* using the Java Persistence API (JPA). A simple DSL for declaration of JPA entities is then developed, introducing the techniques for its implementation, including syntax definition and term rewriting in Stratego/XT². Section 4 develops a generator for deriving from a data model declaration, standardized pages for viewing and editing objects. In Section 5 the coverage of the data model DSL is increased in various directions.

Core Language The abstractions that result from finding programming patterns tend to be coarse grained and capture large chunks of code. In order to implement a variation on the functionality captured in the generator templates, complete new templates need to be developed. The templates for generating view and edit pages developed in Section 4 are very specific to these interaction patterns. Extending this approach to include other, more sophisticated, interaction patterns would lead to a lot of code duplication *within the generator*. To increase the coverage of the DSL it is a good idea to find the *essential abstractions* underlying the larger templates and develop a *core language* that supports freely mixing these abstractions. In Section 6 a core language for web user interfaces is developed that covers page flow, data views, and user interface composition. In Section 7 the core language is extended with typechecking, data input, and queries.

Abstraction Mechanisms A good core language ensures an adequate coverage of the domain. However, this may come at a loss of abstraction. Core language constructs are typically relatively low-level, which leads to frequently occurring patterns combining particular constructs. To capture such patterns and provide high-level abstractions to DSL programmers we need abstraction mechanisms.

Some of these patterns can be captured in templates or modules in a library of common components. In Section 8 WebDSL is extended with abstraction

² While the *concepts* underlying Stratego/XT are explained (to the extent necessary for the tutorial), the details of operating Stratego/XT are not. To get acquainted with the tools the reader should consult the Stratego/XT tutorial and manual [18].

mechanisms for web developers. Template definitions allow developers to create reusable page elements. Modules support the division of an application into reusable files.

Other patterns require reflection over types or other properties of program elements, which may not be so easily defined using the abstraction facilities of the language. Advanced reflection and analysis mechanisms carry a run-time cost and considerably increase the complexity of the language. Such patterns are typically more easily defined using *linguistic abstraction*, i.e. the extension of the language with *syntactic abstractions*, which are implemented by means of transformations to the core language — as opposed to transformations to the target language. Building layers of abstractions on top of a core language is a key feature of software development with DSLs; new abstractions are defined relatively easily, by reusing the implementation knowledge captured in the generator for the core language. Section 9 illustrates this process by defining a number of syntactic abstractions for data input and output.

3 Programming Patterns: Data Model

The first step in the process of designing a DSL is to consider common programming patterns in the application domain. We will turn these patterns into templates, i.e. program fragments with holes. The holes in these templates can be filled with values to realize different instantiations of the programming pattern. Since the configuration data needed to fill the holes is typically an order of magnitude smaller than the programming patterns they denote, a radical decrease in programming effort is obtained. That is, when exactly these patterns are needed, of course. With some thought the configuration data can be turned into a proper domain-specific language. Instead of doing a ‘big design up front’ to consider all aspects a DSL for web applications should cover and the language constructs we would need for that, we develop the DSL in iterations. We start with relatively large patterns, i.e., complete classes.

3.1 Platform Architecture

As argued before, we take a particular technology stack as basis for our WebDSL. That is, this technology stack will be the platform on which code generated from DSL models will run. That way we have a concrete implementation platform when considering design and implementation issues and it provides a concrete code base to consider when searching for programming patterns. Hopefully, we will arrive at a design of abstractions that transcend this particular technology.

In this work we use the Seam architecture for web applications. That is, applications consist of three layers or tiers. The presentation layer is concerned with producing web pages and interpreting events generated by the user. For this layer we use JavaServer Faces (JSF) [72]. The persistence layer is concerned with storing data in the database and retrieval of data from the database. This

layer really consists of two parts. The database proper is a separate service implemented by a relational database. In the implementation of a web application, however, we approach the database via an object-relational mapping (ORM) framework, which takes care of the communication with the database and translates relational data into objects that can be used naturally in an object-oriented setting. Thus, after defining a proper mapping between objects and database tables, we need no longer worry about the database side. Finally, to connect the JSF pages defining the user-interface with the objects obtained from the database we use EJB3 session beans [56, 74].

While it used to be customary for these types of frameworks to require a large portion of an application to be implemented in XML configuration files, this trend has been reversed in the Seam architecture. Most of the configuration is now expressed as annotations in Java classes building on the concept of *dependency injection* [46]. A little XML configuration remains, for instance, to define where the database is to be found. This configuration is mostly static and will not be a concern in this paper.

In this section, we start with considering *entity beans*, i.e. Java classes that implement persistent objects. We will build a generator for such classes, starting with a syntax definition for a data model language up to the rewriting rules defining Java code generation. As such, this section serves as an introduction to these techniques. In the next section we then consider the generation of basic web pages for viewing and editing the content of persisted objects.

3.2 Programming Patterns for Persistence

The Java Persistence API (JPA) [90] is a standard proposed by Sun for object-relational mapping (ORM) for Java. The API is independent of vendor-specific ORM frameworks such as Hibernate; these frameworks are expected to implement JPA, which, Hibernate 3 indeed does [10]. While earlier versions of Hibernate used XML configuration files to define the mapping between database schemas and Java classes, the JPA approach is to express these mappings using Java 5 annotations in Java classes. Objects to be persisted in a database are represented using ‘plain old Java objects (POJOs)’. Classes are mapped to database tables and properties (fields with getters and setters) are mapped to database columns. We will now inspect the ingredients of such classes as candidates for code generation.

Entity Class An *entity class* is a Java class annotated with the `@Entity` annotation and with an empty constructor, which guarantees that the persistence framework can always create new objects.

```
@Entity
public class Publication {
    public Publication () { }
    // properties
}
```

An entity class is mapped to a database table with the same name. If desired, an alternative name for the table can be specified, but we will not be concerned with that (for the time being at least). In general, for many of the patterns we consider here there are alternatives that have (subtly) different semantics. For now, we consider ‘vanilla’ patterns. Later, if and when the need arises we can introduce more variability.

Identity Entities should have an *identity* as primary key. This identity can be any value that is a unique property of the object. The annotation `@Id` is used to indicate the property that represents the identity. However, the advice is to use an identity that is not directly linked to the logic of the object, but rather to use a synthetic identity, for which the database can generate unique values [10]. This then takes the following pattern:

```
@Id @GeneratedValue
private Long id;
public Long getId() { return id; }
private void setId(Long id) { this.id = id; }
```

Properties The values of an object are represented by *properties*, class member fields with getters and setters. Such properties are mapped to columns in the database table for the enclosing class.

```
private String title;
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

Entity Associations No annotations are needed for properties with simple types. However, properties referring to other entities, or to collections of entities, require annotations. The following property defines an association to another entity:

```
@ManyToOne
private Person author = new Person();
public Person getAuthor() { return author; }
public void setAuthor(Person author) { this.author = author; }
```

The `@ManyToOne` annotation states that many `Publications` may be authored by a single `Person`. Alternatively, we could use a `@OneToOne` annotation to model that only one `Publication` can be authored by a `Person`, which implies ownership of the object at the other end of the association.

3.3 A Data Model DSL

Entity classes with JPA annotations are conceptually simple enough. However, there is quite a bit of boilerplate involved. First of all, the setters and getters are completely redundant, and also the annotations can become fairly complex. However, the essence of an entity class is simple, i.e., a class name, and a list

of properties, i.e., (name, type) pairs. This information can be easily defined in a structure of the form `A{ prop* }` with `A` a name (identifier) and `prop*` a list of properties of the form `x : t`, i.e., a pair of a field name `x` and a type `t`. For example, the following entity declarations

```
entity Publication {
  title    : String
  author   : Person
  year     : Int
  abstract : String
  pdf      : String
}
entity Person {
  fullname : String
  email    : String
  homepage : String
}
```

define the entities `Publication` and `Person`, which in Java take up easily 100 lines of code.

The collection of data used in a (web) application is often called the *domain model* of that application. While this is perfectly valid terminology it tends to give rise to confusion when considering domain-specific languages, where the domain is the space of all applications. Therefore, in this paper, we stick to the term *data model* for the data in a web application.

3.4 Building a Generator

In the rest of this section we will examine how to build a generator for the simple data modeling language sketched above. A generator typically consists of three main parts, a parser, which reads in the model, the code generator proper, which transforms an abstract syntax representation of the model to a representation of the target program, and a pretty-printer, which formats the target program and writes it to a text file. Thus, we need the following ingredients. A definition of the concrete syntax of the DSL, for which we use the syntax definition formalism SDF2. A parser that reads model files and produces an abstract representation. A definition of that abstract representation. A transformation to the abstract representation of the Java program to be generated, for which we use term rewrite rules. And finally, a definition of a pretty-printer.

3.5 Syntax Definition

For syntax definition we use the syntax definition formalism SDF2 [101]. SDF2 integrates the definition of the lexical and context-free syntax. Furthermore, it is a modular formalism, which makes it easy to divide a language definition into reusable modules, but more importantly, it makes it possible to *combine* definitions for different languages. This is the basis for rewriting with concrete syntax and language embedding; we will see examples of this later on.

The syntax of the basic domain modeling language sketched above is defined by the following module `DataModel`. The module defines the lexical syntax of identifiers (`Id`), integer constants (`Int`), string constants (`String`)³, whitespace

³ Integer and string constants are not used in this version of the language.

and comments (LAYOUT). Next the context-free syntax of models, entities, properties, and sorts is defined. Note that SDF productions have the non-terminal being defined on the right of the \rightarrow and the body on the left-hand side.

```

module DataModel
  exports
    sorts Id Int String Definition Entity Property Sort
  lexical syntax
    [a-zA-Z][a-zA-Z0-9\_]* -> Id
    [0-9]+                 -> Int
    "\"" ~["\n]* "\""     -> String
    [\\ \t\n\r]           -> LAYOUT
    "//" ~[\n\r]* [\n\r]  -> LAYOUT
  context-free syntax
    Definition*           -> Model      {cons("Model")}
    Entity                -> Definition
    "entity" Id "{" Property* "}" -> Entity  {cons("Entity")}
    Id ":" Sort           -> Property  {cons("Property")}
    Id                    -> Sort      {cons("SimpleSort")}

```

Abstract Syntax An SDF syntax definition defines the concrete syntax of strings in a language. For transformations we want an abstract representation, i.e. the tree structure underlying the grammar. This structure can be expressed concisely by means of an algebraic signature, which defines the constructors of abstract syntax trees. Such a signature can be derived automatically from a syntax definition (using `sdf2rtg` and `rtg2sig`). Each context-free production gives rise to a constructor definition using the name declared in the `cons` attribute of the production as constructor name, and the non-literal sorts as input arguments. Thus, for the `DataModel` language defined above, the abstract syntax definition is the following:

```

signature
  constructors
    Model      : List(Definition) -> Model
               : Entity -> Definition
    Entity     : Id * List(Property) -> Entity
    Property   : Id * Sort -> Property
    SimpleSort : Id -> Sort
               : String -> Id

```

Signatures describe well-formed terms. Terms are isomorphic with structures of the following form:

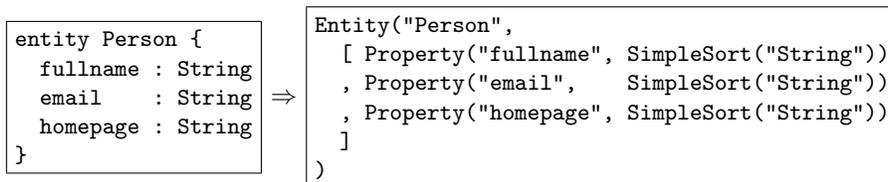
$$t := c(t_1, \dots, t_n)$$

That is, a term is an application of a constructor c to zero or more terms t_i . In practice, the syntax is a bit richer, i.e., terms are defined as

$$t := s \mid i \mid f \mid c(t_1, \dots, t_n) \mid [t_1, \dots, t_n] \mid (t_1, \dots, t_n)$$

including special notation for string (s), integer (i), and float (f) constants, and for lists (`[]`), and tuples (`()`). A *well-formed term* according to a signature is defined according to the following rules. (1) If t_1, \dots, t_n are well-formed terms of sorts s_1, \dots, s_n , respectively, and $c : s_1 * \dots * s_n \rightarrow s_0$ is a constructor declaration in the signature, then $c(t_1, \dots, t_n)$ is a well-formed term of sort s_0 . (2) If t_1, \dots, t_n are well-formed terms of sort s , then $[t_1, \dots, t_n]$ is a well-formed term of sort `List(s)`. (3) If t_1, \dots, t_n are well-formed terms of sorts s_1, \dots, s_n , respectively, then (t_1, \dots, t_n) is a well-formed term of sort (s_1, \dots, s_n) .

Parsing A parser reads a textual representation of a model, checks it against the syntax definition of the language, and builds an abstract syntax representation of the underlying structure of the model text. Parse tables for driving the `sgr` parser can be generated automatically from a syntax definition (using `sdf2table`). The `sgr` parser produces an abstract syntax representation in the Annotated Term (ATerm) Format [96], as illustrated by the following parse of a data model:



3.6 Code Generation by Rewriting

Programs in the *target* language can also be represented as terms. For example, Figure 1 shows the abstract representation of the basic form of an entity class (as produced by the `parse-java` tool, which is based on an SDF definition of the syntax of Java 5). This entails that code generation can be expressed as a term-to-term transformation. Pretty-printing of the resulting term then produces the program text. The advantage of generating terms over the direct generation of text is that (a) the structure can be checked for syntactic and type consistency, (b) a pretty-printer can ensure a consistent layout of the generated program text, and (c) further transformations can be applied to the generated code. For example, in the next section we will see that an interface can be derived from the generated code of a class.

Term rewriting Term rewriting is a formalism for describing term transformations [6]. A *rewrite rule* $p_1 \rightarrow p_2$ defines that a term matching the term pattern p_1 can be replaced with an instantiation of the term pattern p_2 . A term pattern is a term with *variables*. In standard term rewriting, rewrite rules are applied exhaustively until a normal form is obtained. Term rewriting engines employ a built-in *rewriting strategy* to determine the order in which subterms are rewritten. *Stratego* [105, 20] is a transformation language based on term rewriting. Rewrite rules are *named* and can be *conditional*, i.e., of the form $l : p_1$

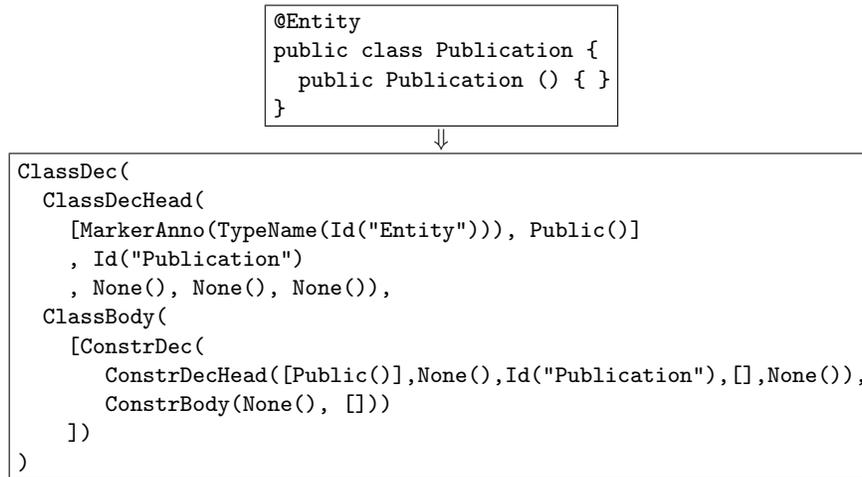


Fig. 1. Abstract syntax for a Java class.

-> p_2 where s , with l the name and s the condition. Stratego extends basic term rewriting by providing *programmable rewriting strategies* that allow the developer to determine the order in which rules are applied. The rewrite rule in Figure 2 defines the transformation of an `Entity` term in the data model language to the basic Java class pattern that we saw above. Note that the rule generalizes over the particular class by using instead of the name "Publication", a *variable* x for the class and the constructor. Thus, the rule generates for an arbitrary `Entity` x , a class x .

In Stratego, a rewrite rule is a special case of a rewriting *strategy* [105]. A strategy is an algorithm that transforms a term into another term, or fails. A strategy definition can invoke rewrite rules and other strategies by name. Strategies can be parametrized with strategies and terms, supporting the definition of reusable strategies.

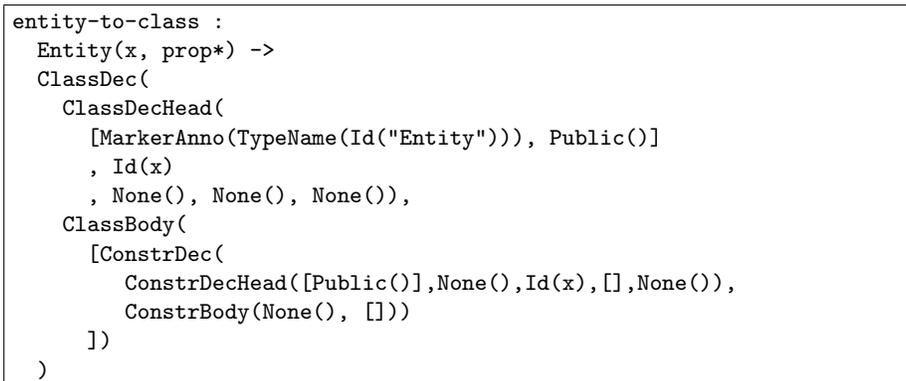


Fig. 2. Term rewrite rule.

Concrete Syntax The `entity-to-class` rewrite rule defines a template for code generation. However, the term notation, despite its advantages for code generation as noted above, is not quite as easy to read as the corresponding program text. Therefore, Stratego supports the definition of rewrite rules using the *concrete syntax* of the subject language [102]. For example, the following rule is the concrete syntax equivalent of the rule in Figure 2:

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

Note that the identifier `x_Class` is recognized by the Stratego parser as a *meta-variable*, i.e. a pattern variable in the rule.

While rewrite rules using concrete syntax have the readability of textual templates, they have all the properties of term rewrite rules. The code fragment is parsed using the proper syntax definition for the language concerned and thus syntax errors in the fragment are noticed at compile-time of the generator. The transformation produces a term and not text; in fact, the rule is equivalent to the rule using terms in Figure 2. And thus the advantages of term rewriting discussed above hold also for rewriting with concrete syntax.

3.7 Pretty-printing

Pretty-printing is the inverse of parsing, i.e. the conversion of an abstract syntax tree (in term representation) to a, hopefully readable, program text. While this can be done with any programmatic method that prints strings, it is useful to abstract from the details of formatting program texts by employing a specialized library. The GPP library [35] supports formatting through the *Box language*, which provides constructs for positioning text blocks. For pretty-printing Java and XML, the Stratego/XT tool set provides custom built mappings to Box. For producing a pretty-printer for a new DSL that is still under development it is most convenient to use a pretty-printer *generator* (`ppgen`), which produces a pretty-print *table* with mappings from abstract syntax tree constructors to Box expressions. The following is a pretty-print table for our `DataModel` language:

```
[
Entity          -- V[V is=2[ KW["entity"] H[_1 KW["{"] ] _2] KW["}"]],
Entity.2:iter-star -- _1,
Property        -- H[_1 KW[":" ] _2],
SimpleSort      -- _1
]
```

Here *V* stands for *vertical* composition, *H* stands for *horizontal* composition, and *KW* stands for *keyword*. While a pretty-printer generator can produce a *correct* pretty-printer (such that `parse(pp(parse(prog))) = parse(prog)`), it is not possible to automatically generate pretty-printers that generate a *pretty* result

(although heuristics may help). So it is usually necessary to tune the pretty print rules.

3.8 Generating Entity Classes

Now that we have seen the techniques to build the components of a generator we can concentrate on the rules for implementing the `DataModel` language. Basically, the idea is to take the program patterns that we found during the analysis of the solution domain, and turn them into transformation rules, by factoring out the application-specific identifiers. Thus, an entity declaration is mapped to an entity class as follows:

```
entity-to-class :
| [ entity x_Class { prop* } ] | ->
| [ @Entity public class x_Class {
  public x_Class () { }
  @Id @GeneratedValue private Long id;
  public Long getId() { return id; }
  private void setId(Long id) { this.id = id; }
  ~*cbds
} ] |
where cbds := <mapconcat(property-to-gettersetter(|x_Class))> prop*
```

Since an entity class always has an identity (at least for now), we include it directly in the generated class. Furthermore, we include, through the anti-quotation `~*`, a list of class body declarations `cbds`, which are obtained by mapping the properties of the entity declaration with `property-to-gettersetter`. Here `mapconcat` is a strategy that applies its argument strategy to each element of a list, concatenating the lists resulting from each application.

Value Types The mapping for value type properties simply produces a private field with a public getter and setter.

```
property-to-gettersetter(|x_Class) :
| [ x_prop : s ] | ->
| [ private t x_prop;
  public t get#x_prop() { return title; }
  public void set#x_prop(t x) { this.x = x; } ] |
where t := <builtin-java-type> s
```

This requires a bit of *name mangling*, i.e. from the name of the property, the names of the getter and setter are derived. This is achieved using an extension of Java for name composition. The `#` operator combines two identifiers into one, observing Java naming conventions, i.e. capitalizing the first letter of all but the first identifier. Note that the name of the enclosing class (`x_Class`) is passed to the rule as a term parameter. Stratego distinguishes between strategy and term parameters of a rule or strategy by means of the `|`; the (possibly empty) list of parameters before the `|` are strategies, the ones after the `|` are terms.

The fact that the property is for a value type is determined using the strategy `builtin-java-type`, which defines a mapping for the built-in types of the `DataModel` language to types in Java that implement them. For example, the `String` type is defined as follows:

```
builtin-java-type :
  SimpleSort("String") -> type|[ java.lang.String ]|
```

Reference Types Properties with a reference to another type are translated to a private field with getters and setters with the `@ManyToOne` annotation. For the time being, we interpret such an association as a non-exclusive reference.

```
property-to-gettersetter(|x_Class) :
  |[ x_prop : s ]| ->
  |[ @ManyToOne
    private t x_prop;
    public t get#x_prop() { return x_prop; }
    public void set#x_prop(t x_prop) { this.x_prop = x_prop; } ]|
  where t := <defined-java-type> s
```

Propagating Declared Entities The previous rule decides that the property is an association to a reference type using the strategy `defined-java-type`, which maps entities declared in the data model to the Java types that implement them. Since the collection of these entity types depends on the data model, the `defined-java-type` mapping is defined *at run-time* during the transformation as a *dynamic rewrite rule* [20]. That is, before generating code for the entity declarations, the following `declare-entity` strategy is applied to each declaration:

```
declare-entity =
  ?Entity(x_Class, prop*)
; rules(
  defined-java-type :
    SimpleSort(x_Class) -> type|[ x_Class ]|
)
```

This strategy first matches (*?p* with *p* a term pattern) an entity declaration and then defines a rule `defined-java-type`, which inherits from the match the binding to the variable `x_Class`. Thus, for each declared entity a corresponding mapping is defined. As a result, the `property-to-gettersetter` rule fails when it is applied to a property with an association to a non-existing type (and an error message might be generated to notify the user). In general, dynamic rewrite rules are used to add new rewrite rules at run-time to the transformation system. A dynamic rule inherits variable bindings from its definition context, which is typically used to propagate context-sensitive information.

3.9 Composing a Code Generator

Using the ingredients discussed above, the basic version of the code generator for WebDSL is defined as the following Stratego strategy:

```
webdsl-generator =
  xtc-io-wrap(webdsl-options,
    parse-webdsl
    ; alltd(declare-entity)
    ; collect(entity-to-class)
    ; output-generated-files
  )
```

The strategy invokes `xtc-io-wrap`, a library strategy for handling command-line options to control input, output, and other aspects of a transformation tool. The argument of `xtc-io-wrap` is a sequence of strategy applications ($s_1; s_2$ is the sequential composition of two strategies). `parse-webdsl` parses the input model using a parse table generated from the syntax definition, producing its abstract syntax representation. The `alltd` strategy is a generic traversal, which is used here to find all entity declarations and generate the `defined-java-type` mapping for each. The generic `collect` strategy is then used to create a set of Java entity classes, one for each entity declaration. Finally, the `output-generated-files` strategy uses a Java pretty-printer to map a class to a program text and write it to a file with the name of the class and put it in a directory corresponding to the package of the class.

4 Programming Patterns: View/Edit Pages

The next step towards full fledged web applications is to create pages for viewing and editing objects in our `DataModel` language. That is, from a data model generate a basic user interface for creating, retrieving, updating and deleting (CRUD) objects. For example, consider the following data model of `Persons` with `Addresses`, and `Users`.

```
entity Person {
  fullname : String
  email    : String
  homepage : String
  photo    : String
  address  : Address
  user     : User
}

entity Address {
  street : String
  city   : String
  phone  : String
}

entity User {
  username : String
  password : String
  person   : Person
}
```

For such a data model we want to generate view and edit pages as displayed in Figures 3 and 4. Implementing this simple user interface requires an understanding of the target architecture. Figure 5 sketches the architecture of a JSF/Seam application for the `editPerson` page in Figure 4. The `/editPerson.seam` client view of the page on the far left of Figure 5 is a plain web page implemented in

File	Edit	View	Go	Favorites	Tools
Fullname	Eelco Visser				
Email	visser@acm.org				
Homepage	http://www.eelcovisser.net				
Photo	/img/eelcovisser.jpg				
Address					
Street	Mekelweg 4				
City	Delft				
Phone	+31 (015) 27 87088				
user	EelcoVisser				
	Edit				

Fig. 3. person page

File	Edit	View	Go	Favorites	Tools	Help
Fullname	Eelco Visser					
Email	visser@acm.org					
Homepage	http://www.eelcovisser.net					
Photo	/img/eelcovisser.jpg					
Address						
Street	Mekelweg 4					
City	Delft					
Phone	+31 (015) 27 87088					
user	EelcoVisser					
	<input type="button" value="Save"/>					

Fig. 4. editPerson page

HTML, possibly with some JavaScript code for effects and cascading style sheets for styling. The rendered version of this code is what is shown in Figure 4. The HTML is *rendered* on the server side from the JavaServer Faces (JSF) component model [72] defined in the `editPerson.xhtml` file. In addition to regular HTML layout elements, the JSF model has components that interact with a *session bean*. The `EditPersonBean` session bean retrieves data for the JSF model from the database (and from session and other contexts). For this purpose the session bean obtains an `EntityManager` object through which it approaches the database, with which it synchronizes objects such as `Person p`. When the input field at the client side gets a new value and the form is submitted by a push of the `Save` button, the value of the input field is assigned to the field pointed at by the expression of the `h:inputText` component (by calling the corresponding setter method). Subsequently, the `save()` action method of the session bean, which is specified in the `action` attribute of the `h:commandButton` corresponding to the `Save` button, is called. This method then invokes the entity manager to update the database.

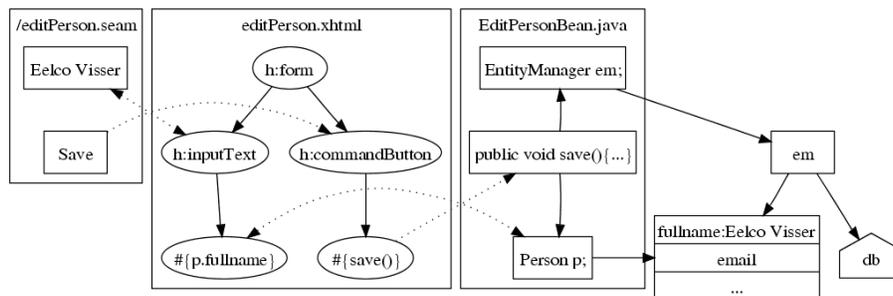


Fig. 5. Sketch of JSF/Seam architecture.

```

<html ...> ... <body>
<h:form>
  <table>
    <tr><td> <h:outputText value="Fullname"/> </td>
      <td> <h:inputText value="#{editPerson.person.fullname}"/>
        </td> </tr>
    <tr><td><h:commandButton value="Save" action="#{editPerson.save()}" />
      </td> <td></td></tr>
  </table>
</h:form>
</body> </html>

```

Fig. 6. editPage.xhtml with JSF components.

Thus, to implement a view/edit interface for data objects, the generator must produce for each page a JSF XHTML document that defines the layout of the user interface and the data used in its elements, and a Seam session bean that manages the objects referred to in the JSF document.

4.1 Generating JSF Pages

Figure 6 illustrates the structure of the JSF XHTML document for the edit page in Figure 4. Besides common HTML tags, the document uses JSF components such as `h:form`, `h:outputText`, `h:inputText`, and `h:commandButton`. Such a document can again be generated using rewrite rules transforming entity declarations to XHTML documents.

```

entity-to-edit-page :
| [ entity x_Class { prop* } ] | ->
%><html ...> ... <body><h:form><table>
  <%= rows ::* %>
  <tr><td>
    <h:commandButton value="Save" action="#{<%=editX%>.save()}" />
  </td><td></td></tr>
</table></h:form></body></html><%
where editX := <concat-strings>["edit", x_Class]
; x_obj := <decapitalize-string> x_Class
; rows := <map(row-in-edit-form(|editX, x_obj))> props

```

This rule generates the overall setup of an edit page from an entity declaration. Just as was the case with generation of Java code, this rule uses the concrete syntax of XML in the right-hand side of the rule [15]. (The quotation marks `%>` and `<%` were inspired by template engines such as JSP [100]). The XML fragment is syntactically checked at compile-time of the generator and the rule then uses the underlying abstract representation of the fragment. For this syntax embedding we do not have `#` operator to create composite identifiers. Instead names are create by simple string manipulation (concatenation in this case). Note that the ellipses `...` are not part of the formal syntax, but just indicate that some elements were left out of this paper to save space.

The `entity-to-edit-page` rule calls `row-in-edit-form` to generate for each property a row in the table.

```
row-in-edit-form(|editX, x_obj) :
  prop@[ x_prop : s ]| ->
  %><tr><td><h:outputText value="#{<%=x_prop%>}" /></td>
    <td><%= input %></td></tr><%
  where input := <property-to-edit-component(|editX, x_obj)> prop
```

The left column in the table contains the name of the property, and the right column an appropriate input component, which is generated by the `property-to-edit-component` rule. In the case of the `String` type a simple `inputText` component is generated.

```
property-to-edit-component(|editX, x_obj) :
  |[ x_prop : String ]| ->
  %><h:inputText value="#{<%=editX%>.<%=x_obj%>.<%=x_prop%>}" /><%
```

Other types may require more complex JSF configurations. For instance, an entity association (such as the `user` property of `Person`) requires a way to enter references to existing entities. The page in Figure 4 uses a drop-down selection menu for this purpose, which is generated by the following rule:

```
property-to-edit-component(|editX, x_obj) :
  |[ x_prop : s ]| ->
  %> <h:selectOneMenu value="#{<%=editX%>.<%=x_obj%>.<%=x_prop%>}">
    <s:selectItems value="#{<%=editX%>.<%=x_prop%>List}"
      var="<%= x %>" label="#{<%= x %>.name}"
      noSelectionLabel="" />
    <s:convertEntity />
  </h:selectOneMenu> <%
  where SimpleSort(_) := s; <defined-java-type> s; x := <new>
```

The `h:selectOneMenu` JSF component sets the value of `editX.x_prop` to the object corresponding to the item selected from the `editX.x_prop#List` list. This list should be provided by the `editX` session bean with the objects to select from, which could be a list of all objects of type `s`.

The generation of a view page is largely similar to the generation of an edit page, but instead of generating an `inputText` component, an `outputText` component is generated:

```
property-to-view-component(|editX, x_obj) :
  |[ x_prop : String ]| ->
  %><h:outputText value="#{<%=editX%>.<%=x_obj%>.<%=x_prop%>}" /><%
```

4.2 Seam Session Beans

As explained above, the JSF components get the data to display from an EJB session bean. The Seam framework provides an infrastructure for implementing

session beans such that the connections to the environment, such as the application logger and the entity manager, are made automatically via *dependency injection* [46]. To get an idea, here is the session bean class for the `editPerson` page:

```
@Stateful
@Name("editPerson")
public class EditPersonBean implements EditPersonBeanInterface{
    @Logger private Log log;
    @In private EntityManager em;
    @In private FacesMessages facesMessages;
    @Destroy @Remove public void destroy() { }
    // specific fields and methods
}
```

EJB3 and Seam use Java 5 annotations to provide application configuration information within Java classes, instead of the more traditional XML configuration files. The use of annotations is also an alternative to implementing interfaces; instead of having to implement a number of methods with a fixed name, fields and methods can be named as is appropriate for the application, and declared to play a certain role using annotations.

The `@Stateful` annotation indicates that this is a stateful session bean, which means that it can keep state between requests. The `@Name` annotation specifies the Seam *component* name. This is the prefix to object and method references from JSF documents that we saw in Figure 6. Seam scans class files at deployment time to link component names to implementing classes, such that it can create the appropriate objects when these components are referenced from a JSF instance. The `destroy` method is indicated as the method to be invoked when the session bean is `@Removed` or `@Destroyed`.

The fields `log`, `em`, and `facesMessages` are annotated for *dependency injection* [46]. That is, instead of creating the references for these objects using a factory, the application context finds these fields based on their annotations and injects an object implementing the expected interface. In particular, `log` and `facesMessages` are services for sending messages, for system logging, and user messages, respectively. The `em` field expects a reference to an `EntityManager`, which is the JPA database connection service.

All the above was mostly boilerplate that can be found in any session bean class. The real meat of a session bean is in the fields and methods specific for the JSF page (or pages) it supports. In the view/edit scenario we are currently considering, a view or edit page has a property for the object under consideration. That is, in the case of the `editPerson` page, it has a property of type `Person`:

```
private Person person;
public void setPerson(Person person) { this.person = person; }
public Person getPerson() { return person; }
```

Next, a page is called with URL `/editPerson.seam?person=x`, where `x` is the identity of the object being edited. The problem of looking up the value

of the `person` parameter in the request object, is also solved by dependency injection in Seam. That is, the following field definition

```
@RequestParam("person") private Long personId;
```

declares that the value of the `@RequestParam` with the name `person` should be bound to the field `personId`, where the string value of the parameter is automatically converted to a `Long` value.

To access the object corresponding to the identity passed in as parameter, the following `initialize` method is defined:

```
@Create
public void initialize() {
    if (personId == null) {
        person = new Person();
    } else {
        person = em.find(Person.class, personId);
    }
}
```

The method is annotated with `@Create` to indicate that it should be called upon creation of the bean (and thus the page). The method uses the entity manager `em` to find the object with the given identity. The case that the request parameter is `null` occurs when no identity is passed to the request. Handling this case supports the creation of new objects.

Finally, a push of the `Save` button on the `editPage` leads to a call to the `save()` method of the bean class, which invokes the entity manager to save the changes to the object to the database:

```
public String save() {
    em.persist(this.getPerson());
    return "/person.seam?person=" + person.getId();
}
```

The return value of the method is used to determine the page flow after saving, which is in this case to go to the view page for the object just saved.

4.3 Generating Session Beans

Generating the session beans for view and edit pages comes down to taking the programming patterns we saw above and generalizing them by taking out the names related to the entity under consideration and replacing them with holes. Thus, the following rule sketches the structure of such a generator rule:

```
entity-to-session-bean :
|[ entity x_Class { prop* } ]| ->
|[ @Stateful @Name("~viewX")
    public class x_ViewBean implements x_ViewBeanInterface {
    ...
}
```

```

    @Destroy @Remove public void destroy() { }
  } ]|
  where viewX := ...; x_ViewBean := ...; x_ViewBeanInterface := ...

```

Such rules are very similar to the generation rules we saw in Section 3.

4.4 Deriving Interfaces

A stateful session bean should implement an interface declaring all the methods that should be callable from JSF pages. Instead of having a separate (set of) rule(s) that generates the interface from an entity, such an interface can be generated automatically from the bean class. This is one of the advantages of generating structured code instead of text. The following strategy and rules define a (generic) transformation that turns a Java class into an interface with all the public methods of the class.

```

create-local-interface(|x_Interface) :
  class -> |[ @Local public interface x_Interface { ~*methodsdecs } ]|
  where methodsdecs := <extract-method-signatures> class

extract-method-signatures =
  collect(method-dec-to-abstract-method-dec)

method-dec-to-abstract-method-dec :
  MethodDecHead(mods, x, t, x_method, args, y) ->
  AbstractMethodDec(mods, x, t, x_method, args, y)
  where <fetch(?Public())> mods

```

The name of the interface defined is determined by the parameter `x_Interface`. The `collect(s)` strategy is a generic traversal that collects all subterms for which its parameter strategy `s` succeeds. In this case the parameter strategy turns a method declaration header into the declaration of an abstract method, if the former is a public method.

5 Programming Patterns: Increasing Coverage

In the previous two sections we analyzed basic patterns for persistent data and view/edit pages in the Seam architecture. We turned these patterns into a simple DSL for data models and a generator for entity classes and view/edit pages. The analysis has taught us the basics of the architecture. We can now use this knowledge to expand the DSL and the generator to cover more sophisticated web applications; that is, to *increase the coverage* of our DSL. Surely we should consider creating custom user interfaces, instead of the rigid view/edit pages that we saw in the previous section. However, before we consider such an extension, we first take a look at the coverage that the data model DSL itself provides.

5.1 Strings in Many Flavors

The association types that we saw in the previous sections were either **Strings** or references to other defined entities. While strings are useful for storing many (if not most) values in typical applications, the type name does not provide us with much information about the nature of those data. By introducing application-domain specific value types we can generate a lot of functionality ‘for free’. For example, the following data models for **Person** and **User** still use mostly string valued data, but using alias types the role of those data is declared.

```
entity Person {
  fullname : String
  email    : Email
  homepage : URL
  photo    : Image
  address  : Address
  user     : User
}

entity User {
  username : String
  password : Secret
  person   : Person
}
```

Thus, the type **Email** represents email addresses, **URL** internet addresses, **Image** image locations, **Text** long pieces of text, and **Secret** passwords. Based on these types a better tuned user interface can be generated. For example, the following rules generate different input fields based on the type alias:

```
property-to-edit-component(|x_component) :
|[ x_prop : Text ]| ->
%><h:inputTextarea value="#{<%=x_component%>.<%=x_prop%>}" /><%

property-to-edit-component(|x_component) :
|[ x_prop : Secret ]| ->
%><h:inputSecret value="#{<%=x_component%>.<%=x_prop%>}" /><%
```

A text-area, providing a large input box, is generated for a property of type **Text**, and a password input field, turning typed characters into asterisks, is generated for a property of type **Secret**.

5.2 Collections

Another omission so far was that associations had only singular associations. Often it is useful to have associations with collections of values or entities. Of course, such collections can be modeled using the basic modeling language. For example, define

```
entity PersonList { hd : Person tl : PersonList }
```

to model lists of **Person**. However, in the first place this is annoying to define for every collection, and furthermore, misses the opportunity for attaching standard functionality to collections. Thus, we introduce a general notion of generic sorts, borrowing from Java 5 generics the notation $X\langle Y, Z \rangle$ for a generic sort X with sort

parameters *Y* and *Z*. For the time being this notation is only used to introduce collection associations using the generic sorts `List` and `Set`. For example, a `Publication` with a list of authors and associated to several projects can then be modeled as follows:

```
entity Publication {
  title    : String
  authors  : List<Person>
  year     : Int
  abstract : Text
  projects : Set<Project>
  pdf      : File
}
```

Many-to-Many Associations Introduction of collections requires extending the generation of entity classes. The following rule maps a property with a list type to a Java property with list type and persistence annotation `@ManyToMany`, assuming that objects in the association can be referred to by many objects from the parent entity:

```
property-to-property-code(|x_Class) :
|[ x_prop : List<y> ]| ->
|[ @ManyToMany private List<t> x_prop = new ArrayList<t>(); ]|
```

Collections also require an extension of the user interface. This will be discussed later in the paper.

5.3 Refining Associations

Yet another omission in the data modeling language is with regard to the nature of associations, i.e. whether they are *composite aggregations* or not. That is, does the referring entity *own* the objects at the other end of the association or not? Since both scenarios may apply, we cannot fix a choice for all applications, but need to let the developer define it for each association. Thus, we refine properties to be either value type (e.g. `title :: String`), composite (e.g. `address <> Address`), or reference (e.g. `authors -> List<Person>`) associations. Figure 7 illustrates the use of special value types, collections, and composite and reference associations

Based on the association type different code can be generated. For example, the objects in a composite collection, i.e. one in which the referrer owns the objects in the collection, are deleted with their owner. In contrast, in the case of a reference collection, only the references to the objects are deleted when the referring object is deleted. Furthermore, collections of value types are treated differently than collections of entities.

<pre>entity Publication { title :: String authors -> List<Person> year :: Int abstract :: Text projects -> Set<Project> pdf :: File }</pre>	<pre>entity Person { fullname :: String email :: Email homepage :: URL photo :: Image address <> Address user -> User }</pre>	<pre>entity Address { street :: String city :: String phone :: String }</pre>
--	---	--

Fig. 7. Data model with composite and reference associations.

Unfolding Associations One particular decision that can be made based on association type is to unfold composite associations in view and edit pages. This is what is already done in Figures 3 and 4. In Figure 7 entity `Person` has a composite association with `Address`. Thus, an address is owned by a person. Therefore, when viewing or editing a person object we can just as well view/edit the address. The following rule achieves this by *unfolding* an entity reference, i.e. instead of including an input field for the entity, the edit rows for that entity are inserted:

```
row-in-edit-form(|editY) :
  |[ x_prop <> s ]| ->
  %><tr><td><h:outputText value="<%=x_prop%>" /></td><td></td></tr>
  <%= row* ::*%><%
  where <defined-java-type> s
    ; prop* := <properties> s
    ; editYX := <concat-strings>[editY, ".", x_prop]
    ; row*   := <map(row-in-edit-form(|editYX))> prop*
```

As an aside, note how the EL expression passed to the recursive call of `row-in-edit-form` is constructed using string concatenation (variable `editYX`). This rather suspect style is an artifact of the XML representation for JSF; the attributes in which EL expressions are represented are just strings without structure. This can be improved upon by defining a proper syntax of JSF XML by embedding a syntax of EL expressions.

6 Core Language: Scrap your Boilertemplate

In the previous sections we have developed a data model DSL with fairly sophisticated types and associations. Furthermore, we have developed a generator for a standard view/edit user interface for objects in the data model. The DSL and generator in the previous sections are based on the analysis of the programming patterns for entity classes and for view/edit pages implemented using JSF and Seam. We factored out the commonality in these programming patterns and turned them into code generation rules with the data modeling DSL as input.

The boilerplate in the generated code is considerable. For example, for the entity `Publication` in Figure 7 the table in Figure 8 contains a breakdown of the source files generated and their size.

With 8 lines of model input, the ratio of generated lines of code to source lines of code is over 100! Now the question is what that buys us. If there was a market for boring view/edit applications this would be great, but in practice we want a much richer application with fine tuned view and edit pages. If we would continue on the path taken here, we could add new sets of generator rules to generate new types of pages. For example, we might

want to have pages for searching objects, pages that list all objects of some type, pages providing selections and summaries, etc. But then we would hit an interesting barrier: code duplication in the code generator. The very phenomenon that we were trying to overcome in the first place, code duplication in application code, shows up again, but now in the form of target code fragments that appear in more than one rule (in slightly different forms), sets of generator rule that are very similar, but generate code for a different type of page, etc. In other words, this smells like boilerplate templates, or *boilertemplates*, for short.

The boilerplate smell is characterized by similar target coding patterns used in different templates, only large chunks of target code (a complete page type) considered as a reusable programming pattern, and limited expressivity, since adding a slightly different pattern (type of page) already requires extending the generator.

High time for some generator refactoring. The refactoring we are going to use here is called **find an intermediate language** also known as *scrap your boilerplate*. In order to gain expressivity we need to better cover the variability in the application domain. While implementing the data model DSL, we have explored the capabilities of the target platform, so by now we have a better idea how to implement variations on the view/edit theme by combining the basics of JSF and EJB in different ways. What we now need is a language that sits in between the high-level data modeling language and the low-level details of JSF/Seam and allows us to provide more variability to application developers while still maintaining an advantage over direct programming.

Frameworks such as JSF provide a large number of features (components) for composing user interfaces. It would be tempting to expose all these components to the DSL programmer to allow for maximal expressivity. However, this is not a good idea for productivity. Rather we would like to provide a small set of basic combinators for declaring the UI, and relying on different sets of definitions for their implementation. A good analogue is the complexity of \TeX vs the standardization of \LaTeX . \TeX provides low-level expressivity for typesetting [66]. With it one can do amazingly complex things. However, for common writing of articles, this complexity is not necessary. \LaTeX harnesses the power of \TeX by

file	LOC
Publication.java	121
EditPublicationBeanInterface.java	56
EditPublicationBean.java	214
ViewPublicationBeanInterface.java	28
ViewPublicationBean.java	117
editPublication.xhtml	181
viewPublication.xhtml	153
total	870

Fig. 8. LOCs generated for Publication.

providing interfaces (APIs) for building documents with a standardized structure (e.g. `\section`, `\item`, etc.) [69]. Using different style files, documents using this interface can be typeset in very different formats. While one could say that HTML serves a similar goal, the customization to implement a certain style requires quite a bit of HTML coding.

Consider the data model for an entity `ResearchGroup` in Figure 9. While a standard edit page is sufficient for this model, we want to create custom presentation pages that highlight different elements. We will use this example to design a basic language for page flow and presentation. Then we develop a generator that translates page definitions to JSF pages and supporting Seam session beans.

```
entity ResearchGroup {
  acronym    :: String (name)
  fullname   :: String
  mission    :: Text
  logo       :: Image
  members    -> Set<Person>
  projects   -> Set<ResearchProject>
  colloquia  -> Set<Colloquium>
  news       -> List<News>
}
```

Fig. 9. Entity `ResearchGroup`.

6.1 Page Flow

The pages in Section 4 had URLs of the form `/researchGroup.seam?g=x` with `x` the identity of the object to be presented. Thus, a page has a name and arguments, so analogously to function definitions, a natural syntax for page definitions is:

```
define page researchGroup(g : ResearchGroup) {
  <presentation>
}
```

The parameter is a variable local to the page definition. The URL to request a page uses object identities. Within a page definition the parameter variable can be treated as referring to the corresponding object. Of course, a page definition can have any number of parameters, including zero.

If a page definition is similar to a function definition, page *navigation* should be similar to a function call. Thus, if `pers.group` refers to a `ResearchGroup` object, then `researchGroup(pers.group)` refers to the `researchGroup` page for that object. However, a link in a web page not only requires the destination of the link, but also a name to display it with. The `navigate` form

```
navigate(researchGroup(pers.group)){text(pers.group.acronym)}
```

combines a page reference with a name for the link. The first argument is a ‘call’ to the appropriate page definition. The second argument is a specification of the text for the anchor, which can be a literal string, or a string value obtained from some data object.

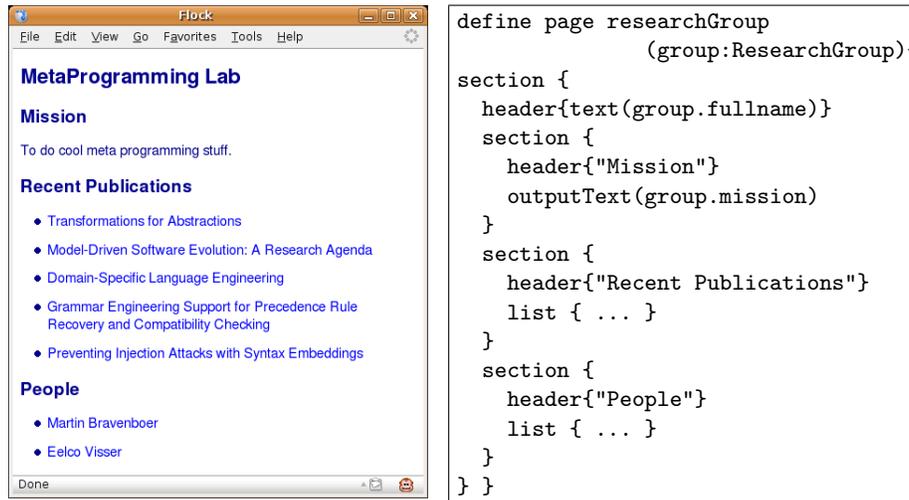


Fig. 10. View of `ResearchGroup` object: (a) screenshot, (b) markup

6.2 Content Markup and Layout

Next we are concerned with presenting the data of objects on a page. For instance, a starting page for a research group might be presented as in Figure 10(a). The layout of such a page is defined using a presentation markup language that can access the data objects passed as arguments to a page. The elements for composition of a presentation are well known from document definition languages such as \LaTeX , HTML, and DocBook and do not require much imagination. We need things such as sections with headers, paragraphs, lists, tables, and text blocks. Figure 10(b) shows the top-level markup for the view in Figure 10(a). It has sections with headers, nested sections, lists, and a text block obtained by taking the `Text` from `group.mission`. The intention of these markup constructs is that they do not allow any configuration for visual formatting. That is, `section` does not have parameters or attributes for declaring the font-size, text color, or text alignment mode. The markup is purely intended to indicate the *structure* of the document. Visual formatting can be realized using cascading style sheets [106], or some higher level styling language.

While the presentation elements above are appropriate for text documents, web pages often have a more two-dimensional layout. That is, in addition to the body, which is laid out as a text document, a web page often contains elements such as a toolbar with drop-down menus, a sidebar with (contextual) links, a logo, etc. Figure 11 illustrates this by an extension of the `ResearchGroup` view page of Figure 10 with a sidebar, menubar with drop-down menus and a logo.

WebDSL takes a simple view at the problem of two-dimensional layout. A page can be composed of `blocks`, which can be nested, and which have a name as in the right-hand side page definition in Figure 11. This definition states that a page is composed of two main blocks, `outersidebar` and `outerbody`, which form the left and right column in Figure 11. These blocks are further subdivided into

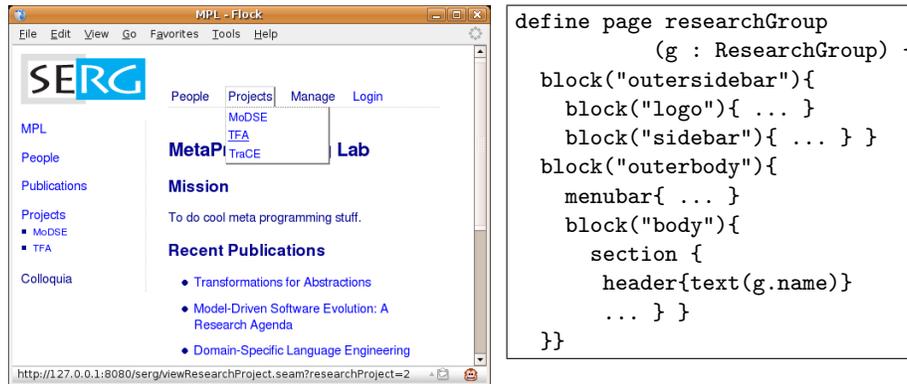


Fig. 11. Two-dimensional layout with logos, sidebars, drop-down menus.

logo and sidebar, and menubar and body, respectively. By mapping blocks to divs in HTML with the block name as CSS class, the layout can be determined again using CSS.

Other layout problems can be solved in a similar way using CSS. For example, the sidebar in Figure 11 is simply structured as a list:

```

block("sidebar"){
  list {
    listitem { navigate(researchGroup(group)){text(group.acronym)} }
    listitem { navigate(groupMembers(group)){"People"} }
    listitem { navigate(groupPublications(group)){"Publications"} }
    listitem { navigate(groupProjects(group)){"Projects"} list{ ... } }
  }
}

```

Using CSS the default indented and bulleted list item layout can be redefined to the form of Figure 11 (no indentation, block icon for sub lists, etc.).

Drop-down menus can be defined using a combination of CSS and some javascript, which can be generated from a declarative description of the menus. For example, the drop-down menus of Figure 11 are defined using elements such as `menu` and `menuitem`:

```

menubar{
  menu{ menuheader{"People"} menuitem{...} ...}
  menu{ menuheader{"Projects"} menuitem{...} ...}
  ...
}

```

Thus, using simple structural markup elements without visual configuration, a good separation of the definition of the structure of a page and its visual layout using cascading style sheets can be achieved. This approach can be easily extended to more fancy user interface elements by targetting java-script in addition to pure HTML. There again the aim should be to keep WebDSL specifications free of visual layout.

6.3 Language Constructs

We have now developed a basic idea for a page presentation language with concepts such as sections, lists, and blocks. The next question is how to define a language in which we can write these structures. The approach that novice language designers tend to take is to define a syntactic production for each markup element. Experience shows that such language definitions become rather unwieldy and make the language difficult to extend. To add a new markup construct, the syntax needs to be extended, and thus all operations that operate on the abstract syntax tree. Lets be clear that a rich syntax is a good idea, but only where it concerns constructs that are really different. Thus, rather than introducing a syntactic language construct for each possible markup element, we use the generic *template call* syntactic construct (why it is called *template call* will become clear later).

Template Call A template call has the following form:

$$f(e_1, \dots, e_m) \{elem_1 \dots elem_n\}$$

That is, a template call has a **name** f , a list of *expressions* e_1, \dots, e_m and a list of *template elements* $elem_1 \dots elem_n$. Both the expression and element argument lists are optional.

The name of the call determines the type of markup and is mapped by the back-end to some appropriate implementation in a target markup language.

The element arguments of a call are nested presentation elements. For example, a **section** has as arguments, among others, headers and paragraphs

```
section{ header{ ... } par{ ... } par{ ... } }
```

a **list** has as elements **listitems**

```
list { listitem { ... } ... }
```

and a **table** has rows

```
table { row{ ... } row{ ... } }
```

The expression arguments of a call can be simple strings, such as the name of a block:

```
block("menu") { list { ... } }
```

However, mostly expressions provide the mechanism to access data from entity objects. For example, the **text** element takes a reference to a string value and displays it:

```
text(group.name)
```

Similarly, the **navigate** element takes page invocation as expression argument and nested presentation elements to make up the text of the link.

```
navigate(publication(pub)){text(pub.name)}
```

Iteration While the template call element is fairly versatile, it is not sufficient for everything we need to express. In particular, we need a mechanism for iterating over collections of objects or values. This is the role of the `for` iterator element, which has the following concrete syntax:

```
for( x : s in e ) {elem*}
```

The reason that this construct cannot be expressed using the syntax of a template call is the variable which is bound locally in the body of the iterator. The iterator is typically used to list objects in a collection. For example, the following fragment of a page involving `g` of type `ResearchGroup`, which has a collection of `projects`, presents a list of links to the projects in `g`.

```
list {
  for(p : ResearchProject in g.projects) {
    listitem { navigate(researchProject(p)){text(p.acronym)} }
  } }
```

6.4 Mapping Pages to JSF+Seam

In Section 4 we saw how to generate a web application for viewing and editing objects in a data model using a row-based interface targetting the JSF and Seam frameworks. We can now use the knowledge of that implementation approach to define a mapping from the new fine grained presentation elements to JSF+Seam. Figure 12 illustrates the mapping for a tiny page definition. The mapping from a page definition to JSF involves creating an XML JSF document with as body the body of the page definition, mapping presentation elements to JSF components and HTML, and object access expressions to JSF EL expressions. The mapping from a page definition to a Seam session bean involves creating the usual boilerplate, `@RequestParameters` with corresponding properties (using `property` as an abbreviation to indicate a private field with a getter and a setter), and appropriate statements in the initialization method. In the rest of this section we consider some of the translation rules.

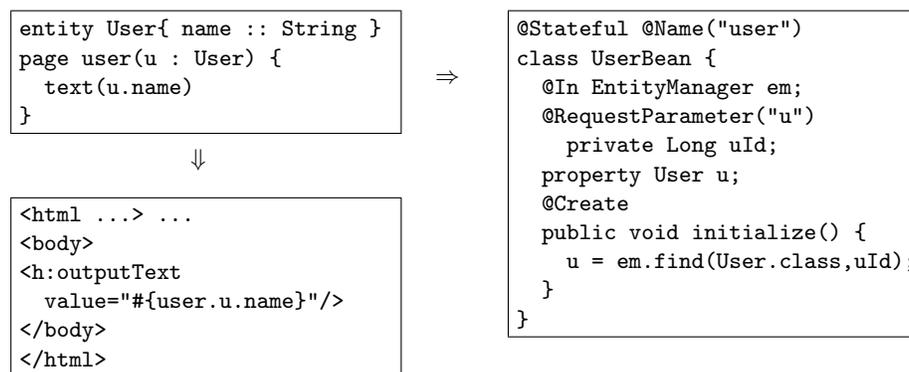


Fig. 12. Mapping from page definition (upper left) to session bean (right) and JSF (lower left).

6.5 Generating JSF

The mapping from page elements to JSF is a fairly straightforward set of recursive rules that translate individual elements to corresponding JSF components. Note that while the syntax of template calls is generic, the mapping is *not* generic. First, while the syntax allows to use arbitrary identifiers as template names, only a (small) subset is actually supported. Second, there are separate generation rules to define the semantics of different template calls. The essence of the domain-specific language is in these code generation rules. They store the knowledge about the target domain that we reuse by writing DSL models. We consider some representative examples of the mapping to JSF.

Text The rule for `text` is a base case of the mapping. A `text(e)` element displays the string value of the `e` expression using the `outputText` JSF component.

```
elem-to-xhtml :
  |[ text(e) ]| -> %> <h:outputText value="<%=e1%>"/> <%
  where e1 := <arg-to-value-string> e
```

The `arg-to-value-string` rules translate an expression to a JSF EL expression.

Block The rule for `block` is an example of a recursive rule definition. Note the application of the rule `elems-to-xhtml` in the antiquotation.

```
elem-to-xhtml :
  |[ block(str){elem*} ]| ->
  %><div class="<%= str %>">
    <%= <elems-to-xhtml> elem* :.*%>
  </div><%
```

The auxiliary `elems-to-xhtml` strategy is a map over the elements in a list:

```
elems-to-xhtml = map(elem-to-xhtml)
```

Iteration While iteration might seem one of the complicated constructs of WebDSL, its implementation turns out to be very simple. An iteration such as the following

```
list{ for ( project : ResearchProject in group.projectsList ) {
  listitem { text(group.project.acronym) }
}}
```

is translated to the JSF `ui:repeat` component, which iterates over the elements of the collection that is produced by the expression in the `value` attribute, using the variable named in the `var` attribute as index in the collection.

```

<ul> <ui:repeat var="project"
      value="#{researchGroup.group.projectsList}">
  <li> <h:outputText value="#{project.acronym}" </li>
</ui:repeat> </ul>

```

This mapping is defined in the following rule:

```

elem-to-xhtml :
| [ for(x : s in e) { elem1* } ] | ->
%><ui:repeat var="<%= x %>" value="<%= el %>">
  <%= elem2* ::*%>
</ui:repeat><%=
where el := <arg-to-value-string> e
      ; elem2* := <elems-to-xhtml> elem1*

```

Navigation The translation of a navigation element is slightly more complicated, since it involves context-sensitive information. As example, consider the following `navigate` element:

```

navigate(viewPerson(prs)){text(prs.name)}

```

Such a navigation should be translated to the following JSF code:

```

<s:link view="/person.xhtml">
  <f:param name="p" value="#{prs.id}" />
  <h:outputText value="#{prs.name}" />
</s:link>

```

While most of this is straightforward, the complication comes from the parameter. The `f:param` component defines for a URL parameter the name and value. However, the name of the parameter (`p` in the example) is not provided in the call (`person`). The following rule solves this by means of the dynamic rule `TemplateArguments`:

```

elem-to-xhtml :
| [ navigate(p(e*)){elem1*} ] | ->
%><s:link view = "/<%= p %>.xhtml">
  <%= <conc>(param*,elem2*) ::*%>
</s:link><%=
where <IsPage> p
      ; farg* := <TemplateArguments> p
      ; param* := <zip(bind-param)> (farg*, e*)
      ; elem2* := <elems-to-xhtml> elem1*

```

In a similar way as `declare-entity` in Section 3 declares the mapping of declared entities to Java types, for each page definition, dynamic rules are defined that (1) record the fact that a page with name `p` is defined (`IsPage`), and (2) map the page name to the list of formal parameters of the page (`TemplateArguments`). Then, creating the list of `f:params` is just a matter of zipping together the list of formal parameters and actual parameters using the following `bind-param` rule:

```

bind-param :
  (| [ x : $X ]|, e) ->
    %><f:param name="<%= x %>" value="<%= e1 %>" /><%
    where <defined-java-type> $X
      ; el := <arg-to-value-string> |[ e.id ]|

```

The rule combines a formal parameter `x` and an actual parameter expression `e` into an `f:param` element with as name the name of the formal parameter, and as value the EL expression corresponding to `e`.

Sections A final example is that of nested sections. Contrary to the custom of using fixed section header levels, WebDSL assigns header levels according to the section nesting level. Thus, a fragment such as

```

section { header{"Foo"} ... section { header{"Bar"} ... } }

```

should be mapped to HTML as follows:

```

<h1>Foo</h1> ... <h2>Bar</h2> ...

```

This is again an example of context-sensitive information, which is solved using a dynamic rule. The rules for `section` just maps its argument elements. But before making the recursive call, the `SectionDepth` counter is incremented.

```

elem-to-xhtml :
  |[ section() { elem1* } ]| -> %> elem2* <%
  where {| SectionDepth
    : rules( SectionDepth := <(SectionDepth <+ !0); inc> )
    ; elem2* := <elems-to-xhtml> elem1*
    |}

```

The dynamic rule scope `{| SectionDepth : ... |}` ensures that the variable is restored to its original value after translating all elements of the section.

The rule for the `header` element uses the `SectionDepth` variable to generate an HTML header with the correct level.

```

elem-to-xhtml :
  |[ header(){ elem* } ]| ->
    %><~n:tag><%= <elems-to-xhtml> elems ::*%></~n:tag><%
    where n := <SectionDepth <+ !1>
      ; tag := <concat-strings>["h", <int-to-string> n]

```

Interesting about this example is that the dynamic rules mechanism makes it possible to propagate values during translation without the need to store these values in parameters of the translation rules and strategies.

6.6 Generating Seam Session Beans

The mapping from page definitions to Seam is less interesting than the mapping to JSF. At this point there are only two aspects to the mapping. First, a page definition gives rise to a compilation unit defining a stateful session bean using the name of the page as Seam component name, and the usual boilerplate for session beans.

```
page-to-java :
|[ define page x_page(farg*) { elem1* } ]| ->
|[ @Stateful @Name("~x_page")
  public class x_Page#Bean implements x_Page#BeanInterface {
    @In private EntityManager em;
    @Create public void initialize() { bstm* }
    @Destroy @Remove public void destroy() {}
    cbd*
  }]|
where x_Page := <capitalize-string> x_page
      ; cbd*   := <map(argument-to-bean-property)> farg*
      ; bstm*  := <map(argument-to-initialization)> farg*
```

Second, for each argument of the page, a `@RequestParameter` with corresponding property is generated as discussed in Section 4.

```
argument-to-bean-property :
|[ x : x_Class ]| ->
|[ @RequestParameter("~x") private Long x#Id;
  private x_Class x;
  public void set#x(x_Class x) { this.x = x; }
  public x_Class get#x() { return x; } ]|
```

Finally, code is generated for initializing the property by loading the object corresponding to the identity when the session bean is created.

```
argument-to-initialization :
|[ x : x_Class ]| ->
|[ if (x_Id == null) { x = new x_Class(); }
  else { x = em.find(x_Class.class, x_Id); } ]|
where x_Id := <concat-strings>[x, "Id"]
```

6.7 Boilertemplate Scrapped

This concludes the generator refactoring ‘scrap your boilertemplate’. We have introduced a language that provides a much better coverage of the user interface domain, and which can be used to create a wide range of presentations. The resulting mapping now looks much more like a compiler; each language construct expresses a single concern and the translation rules are fairly small. Next we consider several extensions of the language.

7 Core Language: Extensions

In the first design of the core language for page definitions some aspects were ignored to keep things simple. In this section we consider several necessary extensions.

7.1 Type Checking

Java is a statically typed language, which ensures that many common programming errors are caught at compile-time. Surprisingly, however, this does not ensure that web applications developed with frameworks such as JSF and Seam are free of ‘type’ errors after compilation.

JSF pages are ‘compiled’ at run-time or deployment-time, which means that many causes of errors are unchecked. Typical examples are missing or non-supported tags, references to non-existing properties, and references to non-existing components. Some of these errors cause run-time exceptions, but others are silently ignored.

While this is typical of template-like data, it is interesting to observe that a framework such as Seam, which relies on annotations in Java programs for configuration, has similar problems. The main cause is that Seam component annotations are scanned and linked at deployment-time, and not checked at compile-time for consistency. Thus, uses of components (e.g. in JSF pages) are not checked. Dependency injection enables loose coupling between components/classes, but as a result, the compiler can no longer check data flow properties, such as guaranteeing that a variable is always initialized before it is used. Another symptom of interacting frameworks is the fact that a method that is not declared in the `@Local` interface of a session bean, is silently ignored when invoked in JSF.

Finally, JPA and Hibernate queries are composed using string concatenation. Therefore, syntactic and type errors (e.g. non-existing column) become manifest only at run-time. Most of these types of errors will show up during testing, but vulnerabilities to injection attacks in queries only manifest themselves when the system is attacked, unless they are tested for.

Type Checking WebDSL To avoid the kind of problems mentioned above, WebDSL programs are statically type checked to find such errors early. The types of expressions in template calls are checked against the types of definition parameters and properties of entity definitions to avoid use of non-existing properties or ill-typed expressions. The existence of pages that are navigated to is checked. For example, for the following WebDSL program

```
entity User { name :: String }
define page user(u : User) {
  text(u.fullname)
  text(us.name)
  navigate(foo()){ "bar" }
}
```

the type checker finds the following errors:

```
$ dsl-to-seam -i test.app
[error] entity 'User' has no property 'fullname'
[error] variable 'us' has no declared type
[error] link to undefined page 'foo'
```

Type Checking Rules The type checker is a transformation on WebDSL programs, which checks the type correctness of expressions and annotates expressions with their type. These annotations will turn out useful when considering higher-level abstractions. The following type checking rule for the iterator construct, illustrates some aspects of the implementation of the type checker.

```
typecheck-iterator :
  |[ for(x : s in e1){elem1*} ]| -> |[ for(x : s in e2){elem2*} ]|
  where in-tc-context(id
    ; e2 := <typecheck-expression> e1
    ; <should-have-list-type> e2
    ; {| TypeOf
      : if not(<java-type> s) then
        typecheck-error(["index ", x, " has invalid type ", s])
      else
        rules( TypeOf : x -> s )
      end
    ; elems2 := <typecheck-page-elements> elems1
    |}
  | ["iterator ", x, "/" ] )
```

First, the type checker performs a *transformation*, that is, rather than just checking, constructs are transformed by adding annotations. Thus, in this rule, the iterator expression and elements in the body are replaced by the result of type checking them. Next, constraints on the construct are checked and errors reported with `typecheck-error`. The `in-tc-context` wrapper strategy is responsible for building up a context string for use in error messages. Finally, the local iterator variable `x` is bound to its type in the `TypeOf` dynamic rule [20]. The dynamic rule scope `{| TypeOf : ... |}` ensures that the binding is only visible while type checking the body of the iterator. The binding is used to annotate variables with their type, as expressed in the `typecheck-variable` rule:

```
typecheck-variable :
  Var(x) -> Var(x){Type(t)}
  where if not(t := <TypeOf> x) then
    typecheck-error(["variable ", x, " has no declared type"])
    ; t := "Error"
  end
```

7.2 Data Input and Actions

The language of the previous section only dealt with *presentation* of data. Data *input* is of course an essential requirement for interactive web applications. To

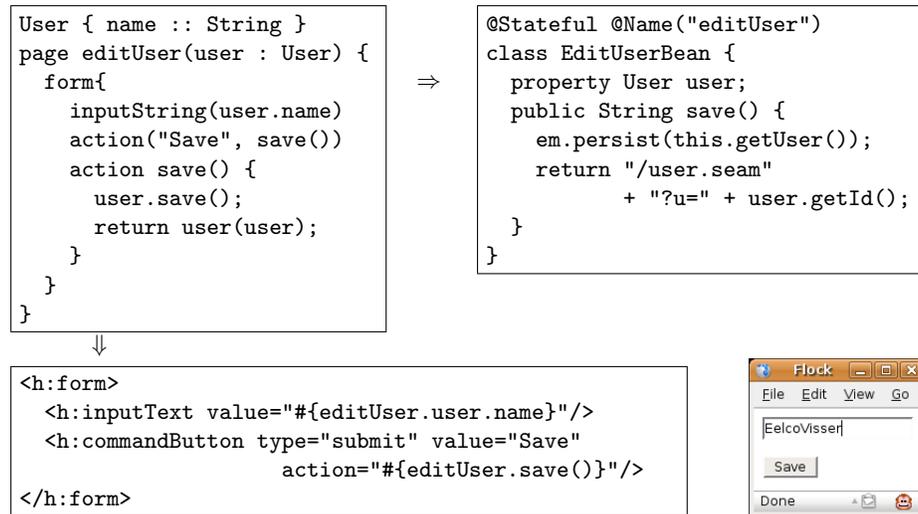


Fig. 13. Mapping form, input field, and action to JSF and Java/Seam.

make edit pages, we need constructs to create input components that bind data to object fields, forms, and buttons and actions to save the data. Figure 13 shows a WebDSL page definition for a simple edit page with a single input field and a Save button, as well as the mapping to JSF and Java/Seam. The language constructs are straightforward. The `form` element builds a form, the `inputString(e)` element creates an input field bound to the contents of the field pointed at by `e`, and the `action` element creates a button, which executes a call to a defined action when pushed. The mapping to Seam is straightforward as well. The action definition is mapped to a method of the session bean.

Action Language The statement language that can be used in action definitions is a simple imperative language with the usual constructs. Assignments such as `person.blog := Blog{title := name}`; bind a value to a variable or field. Method calls such as `publication.authors.remove(author)`; invoke an operation on an object. Currently, the language only supports a fixed set of methods, such as some standard operations on collections, and persistence operations such as `save`. The latter can be applied directly to entity objects, hiding the interaction with an entity manager from the WebDSL developer. The return statement is somewhat unusual, as it is interpreted as a page-flow directive, that is, a statement `return user(u)`; is interpreted as a page redirect with appropriate parameters. Conditional execution is achieved with the usual control-flow constructs.

Expressions consist of variables, constant values (e.g. strings, integers), field access, and object creation. Rather than having to assign values to fields after creating an object, this can be done with the creation expression. Thus, object creation has the form `Person{ name := e ... }`, where fields can be directly

given a value. There is also special syntax for creating sets ($\{e_1, e_2, \dots\}$) and lists ($[e_1, e_2, \dots]$).

Java Embedding The current design of the action language is somewhat ad hoc and should be generalized. A conventional critique of domain-specific languages is that they require the redesign of such things as statements and expressions, which is hard to get right and complete.

An alternative approach would be to directly embed the syntax of Java statements and expressions, and insert the embedded Java fragments into the generated session bean classes. This would give complete access to the full expressivity of Java. Indeed this is what is done with the Hibernate Query Language later in this section. However, Java is a large and complex language; an embedding would entail importing a type checker for Java as well. Furthermore, it would entail tying the DSL to the Java platform and preclude portability to other platforms. HQL and SQL are more portable than Java. That is, as long as we rely on a platform with a relational database, chances are that we can access the data layer through an SQL query. A more viable direction seems to keep the action language simple, but provide a foreign function interface, which gives access to functionality implemented in external libraries to be linked with the application.

7.3 Page Local Variables

So far we have considered pages that operate on objects passed as parameters. Sometimes it is necessary for a page to have local variables. For example, a

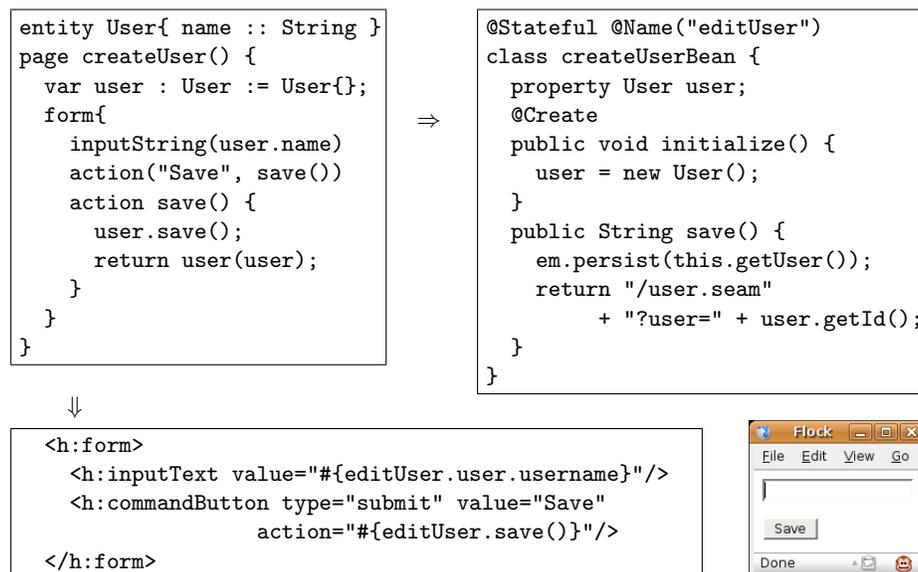


Fig. 14. Page local variables.

page for creating a new object cannot operate on an existing object and needs to create a fresh object. Page local variables support this scenario. Figure 14 illustrates the use of a local variable in the definition of a page for creating new `User` objects, which is mostly similar to the edit page, except for the local variable.

7.4 Queries

The presentation language supports the access of data via (chained) field accesses. Thus, if we have an object, we can access all objects to which it has (indirect) associations. Sometimes, we may want to access objects that are not available through associations. For example, in the data model in Figure 7, a `Publication` has a list of `authors` of type `User`, but a `User` has no (inverse) association to the publications he is author of. In these situations we need a query mechanism to reconstruct the implicit association. In general, queries allow filtering of data.

There is no need to invent a DSL for querying. The Hibernate Query Language (HQL), an adaptation of the relational query language SQL to ORM, provides an excellent query language [10]. To make HQL available in WebDSL we follow the language embedding pattern described in earlier work [102]. Figure 15 illustrates the embedding and its implementation. The query retrieves

```

entity User{ name :: String }
entity Publication{ authors -> List<User> }
page user(user : User) {
  var pubs : List<Publication> :=
    select pub from Publication as pub, User as u
      where (u = ~user) and (u member of pub.authors)
      order by pub.year descending;
  for(p : Publication in pubs) { ... }
}

```

⇓

```

class UserBean {
  property List<Publication> pubs;
  @Factory("pubs") public void initPubs() {
    pubs = em.createQuery(
      "select pub from Publication as pub, User as u" +
      " where (u = :param1) and (u member of pub.authors)" +
      " order by pub.year descending"
    ).setParameter("param1", this.getUser())
    .getResultList();
  }
}

```

Fig. 15. Mapping embedded HQL queries to string-based query construction in Java.

the publications for which the `user` is an author. An HQL query is added to the WebDSL syntax as an expression. For now we assume the result of a query is assigned to a local page variable, which can then be accessed anywhere on the page. Queries can refer to values of page objects by means of the antiquotation `~`. In Figure 15, this is used to find the user with the same identity as the `user` object of the page. The query is translated to a `@Factory` method, which uses the entity manager to create the query using string composition. Antiquoted expressions become parameters of the query.

While the use of HQL in WebDSL does not provide a dramatic decrease in code size, there are some other advantages over the use of HQL in Java. In Java programs, Hibernate queries are composed as strings and parsed at run-time. This means that syntax errors in queries are only caught at run-time, which is hopefully during testing, but maybe during production if testing is not thorough. The `getParameter` mechanism of HQL takes care of escaping special characters to avoid injection attacks. However, use of this mechanism is not enforced and developers can splice values directly into the query string, so the risk of injection attacks is high. In WebDSL, queries are not composed as strings, but integrated in the syntax of the language. Thus, syntactic errors are caught at compile-time and it is not possible to splice in strings without escaping. This embedding of HQL in WebDSL is a variant of the StringBorg approach, which provides a safe way of embedding query-like languages without the risk of injection attacks [16]. Another advantage is that the WebDSL type checker can check the consistency of queries against the data model and local variable declarations. The consistency of HQL queries in Java programs is only checked at run-time.

8 Abstraction Mechanisms: Templates and Modules

In the previous two sections we have extended the data modeling language with a core language for presentation, data input, and page flow. The generator now encapsulates a lot of knowledge about basic implementation patterns. The resulting language provides the required flexibility such that we can easily create different types of pages without having to extend or change the generator. However, this same flexibility entails that page definitions will consist of fragments that occur in other definitions as well. We need to balance the flexibility of the core language with abstraction mechanisms that allow developers to abstract from low-level implementation patterns. We can distinguish two forms; generative and non-generative abstraction mechanisms.

Literal code duplication can be addressed by providing a mechanism for naming and parametrizing code fragments. In this section we extend the language with *templates*, named pieces of code with parameters and hooks. Next, we add *modules*, named collections of definitions defined in a separate file, which can be imported into other modules. Modules are essential for organizing a code base and to form a library of reusable code. These mechanisms are *non-generative*, in the sense that the definitions of patterns are done by the DSL programmer and do not require an extension of the generator.

```

entity Blog {
  title  :: String (name)
  author -> Person
  entries <> List<BlogEntry>
}

entity BlogEntry {
  title  :: String (name)
  created :: Date
  intro  :: Text
}

```

Fig. 16. Data model for blogs and blog entries.

In the next section, we consider *syntactic abstractions*, extensions to the language providing higher-level abstractions, which are implemented by means of ‘model-to-model’ transformations in the generator. These abstraction mechanisms are *generative* (like the ones we saw before). Implementation in the generator allows reflection over the model and non-local transformations.

8.1 Reusing Page Fragments with Template Definitions

Template definitions provide a mechanism for giving a name to frequently used page fragments. A *template definition* has the form

```
define f(farg*){elem*}
```

with f the name of the template, $farg^*$ a list of formal parameters, and $elem^*$ a list of template elements. The use of a defined template in a template call, leads to the replacement of the call by the body of the definition. The markup elements we introduced in Section 6 are also template calls; these are not defined by template definitions, but by the generator. To illustrate the use of template definitions, we consider pages such as the one in Figure 17. The body of the page presents entries in a blog, as represented in the data model in Figure 16, but surrounding that are elements that appear in many other pages as well. The following parameterless template definitions define the literal fragments `logo`, `footer`, and `menu`:

```

define logo() { navigate(home()){image("/img/serg-logo.png")} }
define footer() {
  "generated with "
  navigate(url("http://www.strategoxt.org")){"Stratego/XT"}
}
define menubar() {
  menu{ menuheader{"People"} for(p : Person){ menuitem{...} } } ...
}

```

Such fragments can be reused in many pages, as in the following page definition:



Fig. 17. Instance of blog page.

```

define page home() {
  block("menubar"){ logo() menubar() }
  section{ ... }
  footer()
}

```

Literal template definitions are of limited use. To support reuse of partial fragments, which have holes that should be filled in by the reuse context, templates can have hooks in the form of template calls that can be locally (re)defined. For example, the following `main` template calls `logo`, `sidebar`, `menu`, `body`, and `footer`.

```

define main() {
  block("outersidebar") { logo() sidebar() }
  block("outerbody") {
    block("menubar") { menubar() }
    body()
    footer()
  }
}

```

Some of these templates may have a global definition, such as the ones above, but others may be defined locally in the context where `main` is called. For example, the following page definition calls the `main` template and defines `sidebar` and `body` (overriding any top-level definitions), thus instantiating the calls to these templates in the definition of `main`:

```

define page blog(b : Blog) {
  main()
  define sidebar(){ blogSidebar(b) }
  define body() {
    section{ header{ text(b.title) }
      for(entry : BlogEntry in b.entries) { ... }
    } } }

```

Templates may need to access objects. Therefore, templates can have parameters. For example, the following definition for a sidebar defines links specific to a particular `Person` object `p`.

```

define personSidebar(p : Person) {
  list {
    listitem { navigate(person(p)){text(p.name)} }
    listitem { navigate(personPublications(p)){"Publications"} }
    listitem { navigate(blog(p.blog)){"Blog"} blogEntries() }
    listitem { "Projects" listProjectAcronyms(p) }
  } }

```

This allows templates to be reused in different contexts. For example, the template above can be used to create the sidebar for the view page for a `Person`, as well as for the publications page of that person.

```

define page person(p : Person) {
  main()
  define sidebar() { personSidebar(p) } ...
}
define page personPublications(p : Person) {
  main()
  define sidebar() { personSidebar(p) } ...
}

```

Note that the template mechanism is a form of dynamic scoping; template calls may be instantiated depending on the use site of the enclosing template definition. However, the variables used in expressions are statically bound and can only refer to lexically visible variable declarations, i.e. template parameters, local variables, or global variables. The combination is similar to method overriding in object oriented languages, where variables are lexically scoped, but method invocations may be dynamically bound to different implementations. The template calls in a template definition provide a requires interface of internal variation points.

Template Expansion Template expansion is a context-sensitive transformation, which again relies on dynamic rules for its implementation. For each template definition a dynamic rule `TemplateDef` is defined that maps the name of the template to its complete definition.

```

declare-template-definition =
  ?def@[ define mod* x(farg*){elem*} ] |
  ; rules( TemplateDef : x -> def )

```

The dynamic rule is used to retrieve the definition when encountering a template call. Subsequently, all bound variables in the definition are renamed to avoid capture of free variables.

```

expand-template-call :
  |[ x(e*){elem1*} ] | -> |[ elem2* ] |
  where <TemplateDef; rename> x => |[ define mod* x(farg*){elem3*} ] |
  ; { | Subst
    : <zip(bind-variable)> (farg*, <alltd(Subst)> e*)
    ; elem2* := <map(expand-element)> elem3*
    ; str := x
    | }

```

The formal parameters of the template are bound to the actual parameters of the call in the dynamic rule `Subst`:

```

bind-variable = ?(Arg(x, s), e); rules( Subst : Var(x) -> e )

```

8.2 Modules

A module system allows a code base to be organized into coherent and possibly reusable units, which is a requirement for building a library. Module systems come in different levels of complexity. Module systems supporting separate compilation can become quite complex, especially if the units of compilation in the DSL do not match the units of compilation of the target platform. For this version of WebDSL a very simple module system has been chosen that supports distributing functionality over files, without separate compilation. A module is a collection of domain model and template definitions and can be imported into other modules as illustrated in Figures 18 and 19. The generator first reads in all imported modules before applying other transformations. The implementation of import chasing is extremely simple:

```
import-modules =
  topdown(try(already-imported <+ import-module))

already-imported :
  Imports(name) -> Section(name, [])
  where <Imported> name

import-module :
  Imports(name) -> mod
  where mod := <parse-webdsl-module>FILE(<concat-strings>[name, ".app"])
           ; rules( Imported : name )
```

The dynamic rule `Imported` is used to prevent importing a module more than once.

9 Abstraction Mechanisms: Syntactic Sugar

With the core language introduced in Sections 6 and 7 we have obtained expressivity to define a wide range of presentations. With the templates and modules from the previous section we have obtained a mechanism for avoiding code duplication. However, there are more generic patterns that are tedious to encode for which templates are not sufficient. Even if a language provides basic expressivity, it may not provide the right-level of abstraction. So if we encounter reoccurring programming patterns in our DSL, the next step is to design higher-level abstractions that capture these patterns. Since the basic expressivity is present we can express these abstractions by means of transformations from the extended DSL to the core DSL. Such transformations are known as *desugarings*, since the high-level abstractions are known as *syntactic sugar*. In this section we discuss three abstractions and their corresponding desugarings.

9.1 Output Entity Links

A convention in WebDSL applications is to define for each entity type a corresponding page definition for viewing objects of that type with the name of

```

module publications
section domain definition
  Publication {
    title    :: String (name)
    year     :: Int
    authors  -> List<Person>
    abstract :: Text
  }
section presenting publications
define showPublication(pub : Publication) {
  for(author : Person in pub.authors){
    navigate(person(author)){text(author.name)} " , " }
  navigate(publication(pub)){text(pub.name)} " , "
  text(pub.year) "."
}

```

Fig. 18. Module definition.

```

application org.webdsl.serg
imports templates
imports people
imports blog
imports publications

```

Fig. 19. Application importing modules.

the entity in lowercase. For example, for entity `Publication`, a page definition `publication(p : Publication)` is defined. Given an object, say `pub : Publication`, creating a link to such a page is then realized with `navigate` as follows:

```
navigate(publication(pub)){text(pub.name)}
```

While not a lot of code to write, it becomes tedious, especially if we consider that the code can be derived from the *type* of the variable. Thus, we can replace this pattern by the simple element

```
output(pub)
```

This abstraction is implemented by the following desugaring rule, which uses the *type* of the expression to determine that the expression points to an entity object:

```

DeriveOutputSimpleRefAssociation :
  |[ output(e){} ]| -> |[ navigate($y(e)){text(e.name)} ]|
  where |[ $Y ]| := <type-of> e
         ; <defined-java-type> |[ $Y ]|
         ; $y := <decapitalize-string> $Y

```

This desugaring is enabled by the type annotations on expressions produced by the type checker. Similar desugaring rules can be defined for other types, as illustrated by the following rules:

```
DeriveOutputText :
  |[ output(e){} ]| -> |[ navigate(url(e)){text(e)} ]|
  where |[ URL ]| := <type-of> e
```

```
DeriveOutputText :
  |[ output(e){} ]| -> |[ image(e){} ]|
  where |[ Image ]| := <type-of> e
```

As a consequence of this abstraction, it is sufficient to write `output(e)` to produce the default presentation of the object indicated by the expression `e`.

9.2 Editing Entity Collection Associations

Editing a collection of entities is not as simple as editing a string or text property. Instead of typing in the value we need to select an existing object from some kind of menu. Consider the edit page for a publication in Figure 20. Editing the `authors` association requires the following ingredients: a list of names of entities already in the collection; a link `[X]` to remove the entity from the collection; a select menu to add a new (existing) entity to the collection. This is implemented by the following WebDSL pattern:

```
list { for(person : Person in publication.authors) {
  listitem{ text(person.name) " "
            actionLink("[X]", removePerson(person)) }
} }
select(person : Person, addPerson(person))
action removePerson(person : Person) {
  publication.authors.remove(person);
}
action addPerson(person : Person) {
  publication.authors.add(person);
}
```

The `select` creates a drop-down menu with (names of) objects of some type. Upon selection of an element from the list, the corresponding action (`addPerson` in this case), is executed. This fragment illustrates the flexibility of the presentation language; a complex interaction pattern can be composed using basic constructs. However, repeating this pattern for each entity association is tedious. Creating this pattern can be done automatically by considering the type of the association, which is expressed by the first desugaring rule in Figure 21. Thus, `input(pub.authors)` is now sufficient for producing the implementation of an association editor⁴. Similar rules can be defined for other types, as illustrated in

⁴ At the time of producing the final version of this paper, the editing of collection associations has been replaced with a different implementation.

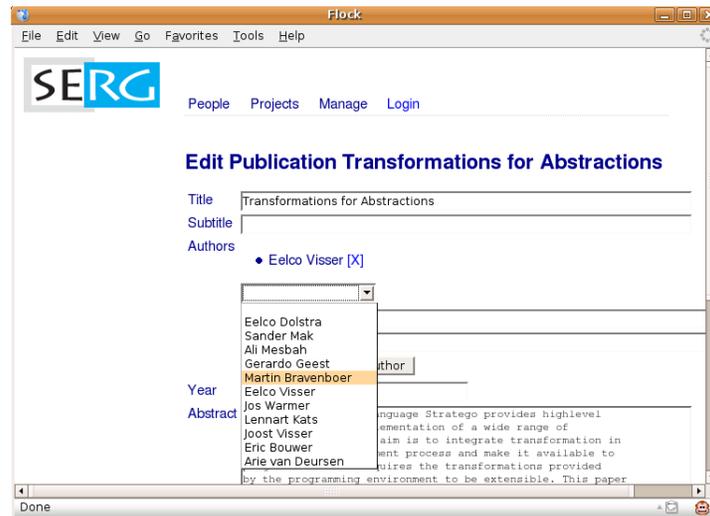


Fig. 20. Editing collection association.

```

DeriveInputAssociationList :
elem|[ input(e){} ]| ->
elem|[ list { for(x : $X in e){
  listitem{text(x.name) " " actionLink("[X]", $removeX(x))}
} }
select(x : $X, $addX(x))
action $removeX(x : $X) { e.remove(x); }
action $addX(x : $X) { e.add(x); } ]|
where |[ List<$X> ]| := <type-of> e
; x := <decapitalize-string; newname> $X
; $removeX := <concat-strings; newname>["remove", $X]
; $addX := <concat-strings; newname>["add", $X]

DeriveInputText :
|[ input(e){} ]| -> |[ inputText(e){} ]|
where SimpleSort("Text") := <type-of> e

DeriveInputSecret :
|[ input(e){} ]| -> |[ inputSecret(e){} ]|
where SimpleSort("Secret") := <type-of> e

```

Fig. 21. Desugaring rules for input

Figure 21. As a consequence, the `input(e)` call is now sufficient for producing the appropriate input interface.



Fig. 22. Edit BlogEntry

9.3 Edit Page

The presentation language supports the flexible definition of custom user interfaces. Based on this language the generation of the standard view/edit interface can now be reformulated as a model-to-model transformation. Rather than directly generating Java and JSF code, a presentation model can be generated from an entity declaration. The generator for the core language then generates the implementation. We consider edit pages such as in Figure 22, which consist of an input box for each property of an entity, organized in a table, and **Save** and **Cancel** buttons. The pattern for the (body of) an edit page is:

```
form {
  table {
    row{ "Blog"      input(entry.blog) }
    row{ "Title"    input(entry.title) }
    row{ "Created"  input(entry.created) }
    row{ "Category" input(entry.category) }
    row{ "Intro"    input(entry.intro) }
    row{ "Body"     input(entry.body) }
  }
  action("Save", save()) action("Cancel", cancel())
  action cancel() { cancel blogEntry(entry); }
  action save() { entry.save(); return blogEntry(entry); }
}
```

Generation of pages of this form is now defined by the **entity-to-edit-form** rule in Figure 23. Note that **\$x** is used both as the argument of the edit page *and* the name of the view page. For each property a table row with an **input** element is generated using the **property-to-edit-row** rule. Application of the

```

entity-to-edit-form :
  |[ entity $X { prop* } ]| ->
  |[ define page $editX($x : $X) {
    form {
      table { elem* }
      action("Save",  save())
      action("Cancel", cancel())
    }
    action cancel() { return $x($x); }
    action save() { $x.save(); return $x($x); }
  } ]|
  where $x      := <decapitalize-string> $X
         ; $editX := <concat-strings>["edit", $X]
         ; elem*  := <map(property-to-edit-row(|$x))> prop*

property-to-edit-row(|x) :
  |[ y k s (anno*) ]| -> |[ row { str input(x.y) } ]|
  where str := <capitalize-string> y

```

Fig. 23. Derivation of edit page from entity declaration.

previously defined desugaring rules for `input` then take care of implementing the interaction pattern corresponding to the type of the property.

10 Discussion: Web Engineering

The development of WebDSL in this paper touches on the development of domain-specific languages and on abstractions for web engineering. WebDSL was intended in the first place as a case study in the development of domain-specific languages. By now it has turned into a practically useful language. Since the first version of WebDSL, which is described in this paper, the language has been improved to increase coverage and has been extended with higher-level abstractions. List comprehensions support easy filtering and ordering of lists. Entity and global function definitions are useful for separating logic from presentation. Entity inheritance and extension support separation of concerns in data modeling. Recursive templates support the presentation of nested structures. Declarative access control rules regulate the access to pages and actions [52]. Furthermore, some of the implementation patterns have been replaced by others, without affecting the design of the language.

This section gives an assessment of WebDSL as a solution in the domain of web engineering. The criteria for the success of a DSL from the introduction are reiterated and the WebDSL project is evaluated with respect to these criteria. WebDSL is compared to alternative web engineering approaches, giving rise to ideas for further improvements and extensions. The next section considers other approaches and techniques for DSL engineering with respect to the criteria. Section 12 considers several challenges for language engineering.

10.1 DSL Engineering Evaluation Criteria

For the *process* of developing a domain-specific language we consider the following criteria:

- Productivity: What is the expected time to develop a new language? Distinguish the costs of domain analysis, language design, and language implementation.
- Difficulty: How *difficult* is it to develop a language? Can it be done by an average programmer or does it require special training? Does it require special infrastructure?
- How *systematic* and *predictable* is the process?
- Maintainable: How well does the process support language *evolution*? How difficult is it to change the language? Can languages be easily *extended* with new abstractions?

For the domain-specific language produced by a language engineering project we consider the following criteria:

- Expressivity: Do the language abstractions support *concise* expression of applications? What is the effect on the *productivity* of developing applications using the DSL compared to the traditional programming approach?
- Coverage: Are the abstractions of the language adequate for developing applications in the domain? Is it possible to express every application in the domain?
- Completeness: Does the language implementation create a complete implementation of the application or is it necessary to write additional code?
- Portability: Can the abstractions be implemented on a different platform? Does the language *encapsulate* implementation knowledge? To what extent do the abstractions leak implementation details of the target platform?
- Code quality: Is the generated code correct and efficient?
- Maintainability: How well does the language support *evolution*? What is the impact of changing a model? What is the impact of changes to the language?

In the following we evaluate the WebDSL design and development with respect to these criteria.

10.2 Evaluation of the WebDSL Development Process

The version of WebDSL described in this paper emerged from a project conducted by the author (non full-time) between September 2006 to June 2007. Several master's students conducted related research activities that provided input for the project. In particular, Sander Mak developed a concurrent DSL for web applications [71] from which the idea of page definitions and navigations analogous to function definitions and calls originated.

Productivity and Difficulty The effort of a language engineering project is divided into domain analysis, language design, and language implementation. In the WebDSL project, by far the most effort was spent in the first stage, i.e. becoming adequately knowledgeable in (one configuration of) the Java web programming platform. To give an indication of the effort involved, here is a brief description of the time line of the project.

In September 2006 a simple wiki application was built with MySQL, JSP, JDBC, and Java Servlets. The application included a wiki markup parser and HTML renderer. In February and March 2007 the wiki application was rewritten using Hibernate as object-relational mapping solution, greatly simplifying the implementation and improving the code quality. The reimplementa-tion consisted of several iterations and introduced some complex features such as nested wiki pages and uploading legacy wiki content from XML data. At the end of March 2007, refactoring the code of the wiki application to try out new architectural ideas became too painful, and a start was made with building WebDSL. In April 2007, JSF, Seam, and Hibernate with annotations (instead of XML configuration) were ‘discovered’ and used as target platform in the emerging generator. Generation of a basic CRUD application (Section 4) and refinement of the data model DSL (Section 5) were realized by mid April. With this basic generator in place it was now possible to experiment with much larger data models than the one for the wiki application. The running example was changed to the ‘research group’ application with publications, home pages, project, blogs, etc. that features in this paper. The presentation language and desugaring transformations for higher-level abstractions (Section 6) were developed in May 2007. The embedding of HQL queries, the module system, and numerous refinements and improvements were realized in June 2007.

Language design can be further divided into discovering the conceptual abstractions and formalizing these abstractions by means of a syntax definition. Again, most of the effort was spent in abstraction discovery; syntax definition with SDF is straightforward once the desired notation has been designed. The data model notation is not particularly original; it is basically a variation on record declarations in C or Pascal. The presentation layer language took a while to emerge. Although with hindsight it is a fairly obvious abstraction from JSF templates. In general, WebDSL liberally borrows designs from existing languages, which is a good idea since these designs will be familiar to developers.

Language implementation was heavily interleaved with design. The author has ample experience in language design and implementation, and is, as primary designer, intimately familiar with the Stratego/XT implementation technology. Thus, implementation of the generator required mainly the ‘encoding’ of the implementation patterns as rewrite rules and strategies using standard Stratego/XT practices. Getting to this level of language implementation productivity requires training in language design and a particular implementation technology such as Stratego/XT. A few innovations of Stratego/XT were made during the development of WebDSL. In particular, some utilities for the generation of multiple output files were developed. Furthermore, in a refactoring

of the WebDSL generator several measures were taken to increase the *locality* of generation rules [54]. In particular, an extension of Java has been developed to support identifier composition, partial classes, partial methods, and interface derivation.

Systematic The inductive, technology driven approach to DSL design adopted in the WebDSL project ensures a natural scope. The domain is defined by whatever is being programmed in practice. Abstractions are discovered by studying programming patterns; common codes ends up as constant code in templates, variable parts are inserted based on information in the model. This approach initially just leads to straightforward abstractions from existing programming practice. However, identification of these abstractions leads to better insight in the domain, which may give rise to reformulations not directly inspired by programming patterns. For example, the access control extension of WebDSL [52] is not based on the facilities for access control provided by the Seam framework. Rather an expressive and declarative mechanism is developed enabled by the possibility to perform desugaring transformations on the DSL itself.

Language design requires some creativity and cannot be very predictable. At first, abstractions can be formulated as enumeration of configuration data, possibly in some XML schema. However, good DSLs require a readable concrete syntax. Language design can be inspired by existing language design patterns. For example, the design of the user interface language of WebDSL was inspired took some inspiration from L^AT_EX, not so much in its concrete syntax, as in concepts of separation of structure and style. A catalog of reusable language design patterns could be helpful in the design of new DSLs.

The implementation of WebDSL follows standard architectural patterns for DSL generators.

Maintainable The extensibility of Stratego strategy definitions makes a generator naturally extensible to support new constructs of the same nature as existing ones. However, the extension of WebDSL with access control and the addition of new user interface components, eventually required a number of refactorings to maintain the modularity of the generator [54].

10.3 Evaluation of the WebDSL Language

Expressivity Programming web applications in WebDSL is a breeze compared with programming in the underlying Seam architecture. Implementations are small and the data model and presentation are easily adapted when insights in the design of an application change. To objectively measure the decrease in effort (say lines of code) that is obtained by using WebDSL it is necessary to simultaneously develop the same web application in WebDSL and using some other techniques. Alternatively, we can exactly rebuild existing web applications and compare the two implementations. As an approximation we can take metrics from WebDSL projects as an indication.

For the website of `webdsl.org` we are developing a software project management application using WebDSL. The current prototype counts 2800 lines of WebDSL code and provides blog, forum, wiki, and issue tracker sub-applications. Access to the applications is controlled by a declarative access control policy (see below). The various applications support cross-linking from user-provided content via wiki-like links, which can address pages symbolically, for example `[[issue(WEBDSL-10)]]` creates a link from a blog entry to an issue in the issue tracker. The generated implementation of this application takes about 44K lines of Java code (3.6K for entity classes, the rest for beans) and some 25K lines of XHTML. Of course, this code is not necessarily as compact as it would be programmed manually. But a factor of 5 to 10 decrease in size compared to manually programmed applications appears to be a realistic (conservative) estimate.

The order of magnitude decrease in code size implies a significant increase in productivity. In particular, refactoring the design of an application can be realized much faster than is the case in the target platform, simply because less code is involved. However, the reduction of accidental complexity reduces application development to the hard part of development, i.e., requirements analysis and application design. Once it is known what the structure and functionality of an application should be, it is easy to realize that. However, WebDSL does not (yet) provide much help for coming up with a design. Further abstractions, such as for workflow, can help guide the design of certain ‘genres’ of applications.

While macro productivity is increased, micro productivity is not ideal. The time it takes to generate code, compile it, and deploy it in a JBoss application server determine the development feedback cycle. This cycle entails a penalty that is felt most when making small changes. A better model for incremental compilation and deployment should improve this factor.

Coverage The WebDSL language supports the creation of a wide range of web applications with a rich data model. There are numerous ways in which the coverage of WebDSL can be extended and refined. In the rest of this section several ideas are discussed.

Completeness The WebDSL generator generates complete code. There is no need to fill in or manually tune generated code skeletons. Sometimes it is necessary to add new built-in types. For instance, to represent patches for version management of the wiki application of `webdsl.org`, a patch library implemented in Java was added to the collection of libraries comprising the run-time system. Such built-in types are implemented as a separate module with rules plugging into the type checker and code generator. This extensibility should be made less intrusive by supporting the declaration of new types and operations in the language itself.

Portability The portability of WebDSL to other Java web frameworks, or other implementation platforms such as PHP or C# has not yet been realized, so

no hard claims about the quality of the WebDSL abstractions can be made. However, there is some evidence that the abstractions are fairly robust and target platform independent. Several of the programming patterns that gave rise to the WebDSL abstractions have been replaced by others, without changing the language constructs that they gave rise to. In Section 8 the template mechanism is implemented through expansion. This precludes the use of recursive template invocations, which would be useful for the presentation of hierarchical, nested structure such as a document with sections and subsections. Recently, we figured out how to translate separate template definitions. This required a change in the back-end of the generator, but the language itself already supported the expression of recursive template invocations.

Code Quality WebDSL applications inherit properties such as performance, robustness, and safety from the target architecture. The technology driven approach underlying the design of WebDSL starts from the assumption that the target architecture is solid. However, Seam itself is new and under development. No experiments have been performed yet to establish these properties in a production setting.

Evolution Complete code generation ensures that *regular* evolution of an application is a matter of reapplying the generator to obtain an implementation for a new version. Otherwise, the evolution of web applications and the version of WebDSL they are constructed with has been ignored in this paper. It is however, an important consideration in a software development process based on DSLs. Section 12 outlines (research) challenges for evolution of DSL-based software development.

10.4 Static Verification

WebDSL statically checks application definitions. Expressions accessing, manipulating, and creating data are checked for consistency with the declared entities and the variable declarations in scope. The existence of pages in navigations is checked, the types of actual parameters to page navigations are checked against the formal parameters of page definitions. Embedded HQL queries can also be checked against the declared entities; implementation of this feature is not yet complete. The remaining errors are logical errors in actions (e.g. accessing a property with null value), and errors in the composition of web pages. In practice, most errors that occur during development are application design errors. That is, realization during testing that pages and interactions should be organized differently. Due to code generation, the generated code correctly implements the specification. Errors normally made in boilerplate code are avoided. Any remaining errors are bugs in the generation templates, which only need to be repaired once.

Logical errors cannot be completely eliminated. Well-formedness of generated web pages could be checked statically by extending the type checker to check for

valid combinations. The only error of this kind encountered in practice, is forgetting to embed form elements in a `form{...}`. The other template elements can be combined fairly liberally due to the leniency of browsers. However, checking such properties would ensure better HTML documents. This is done in systems such as `<bigwig>` [14], `JWIG` [25], `WASH` [92] and `Ocsigen` [9]. In particular, the `<bigwig>` and `JWIG` systems provide sophisticated correctness checks of document well-formedness. Templates in these systems are used to dynamically create documents, including the use of recursive definitions. Data-flow analysis is used to verify that all possible documents that can be generated by a program are valid.

10.5 Input Validation and Data Integrity

Properties and entities may need to satisfy more strict constraints than can be expressed using types alone. First, in some cases it is required to restrict the form of value types. For example, the syntax of an email address should be checked on submission and an error reported if not conforming. Next, constraints on combinations of objects should be checked. For example, in a conference system, the author of a paper may not be a reviewer of that same paper. Violations to this constraint should be detected when changes are made. Both types of constraints can be expressed declaratively, using regular expressions for input validation and Boolean expressions over object graphs for structural invariants. The `PowerForms` tool of the `<bigwig>` project provides a declarative language for declaring the client-side validation of form fields using regular expressions and interdependencies between form fields [13]. We plan to include support for the specification of data integrity constraints in a future version of `WebDSL`.

10.6 Access Control

A related concern is controlling the access to data and the pages that present and modify them. Access control checks can be expressed in `WebDSL` page definitions by means of a conditional content construct (if condition holds, show this content). However, directly expressing access control with that mechanism would result in a tangling of concerns. We have designed an extension of `WebDSL` with declarative rules for user authentication and access control that supports separate specification [52].

10.7 Presentation

Presentations in `WebDSL` depend on the basic page elements defined by the generator. The elements supported currently cover the basics of HTML, abstracting from visual layout by relying on cascading stylesheets (CSS). Fancier elements can be added by extending the generator with new mappings from page elements to `JSF` components. It should be possible to provide such extensions as a plug-in to the generator, which requires an extensibility mechanism. Using the

extensibility of strategy definitions in Stratego and an extension of Java to support partial classes, such extensibility is realized in a refactoring of the WebDSL generator [54]. A concern in the design of such extensions should be a proper separation between declaration of the structure of page content and visual formatting. Many JSF components are variations on the same theme, e.g. a list, vs a table, vs a grid, which are different visualizations of the same information.

The current design of WebDSL is page-centric, with actions and navigations leading to requests of complete new pages. The trend in web application design is towards inclusion of elements from rich (desktop) user interfaces, in which only parts of the page get updated as a reaction to user actions. An experiment with targetting the Echo2 Ajax framework [2] has shown that it might be feasible to develop rich user interfaces with the WebDSL abstractions. The central idea of the experiment was to use templates as the components to be replaced as a response to user actions. A less ambitious approximation of richer user interfaces can be obtained by targetting Ajax JSF components, which is already done to some extent.

10.8 Control-Flow

WebDSL provides a high-level language for implementing web applications by abstracting away from low-level details. However, in its core the language has the same page-centric model as the underlying Seam architecture. It could even be observed that WebDSL makes this architecture more explicit; where in Seam a page is defined by means of a number of separate artifacts, WebDSL unifies the elements of a page in a single definition. This architecture implies that user interactions take the shape of a series of requests and responses.

The Mawl [4] form processing language introduced a paradigm for modeling web interactions in the form of traditional console interaction. That is, web pages are considered as the input and output actions of a sequential program that control the interaction. The following Mawl example defines a session in which first the user should provide a name (`GetName`), which is echoed in the next step (`ShowInfo`) [4]:

```
global int access_cnt = 0;
session Greet {
  local form {} -> { string id } GetName;
  local form { string id, int cnt } -> {} ShowInfo;
  local string i = GetName.put({}).id;
  ShowInfo.put({i, ++access_cnt});
}
```

Here `GetName` and `ShowInfo` are the names of separately defined HTML templates with parameters filled by the `put` operation. The statelessness of the http protocol requires the server to remember where to resume the program after the user submits a request.

In an application of Scheme to web applications, Queinnec [80] observed that capturing of the interaction state can be implemented elegantly by means

```
entity Counter { accesses :: Int }

globals { var stats : Counter := Counter { accesses := 0 }; }

entity Visitor { name :: String }

define page getname() {
  form {
    var n : String;
    "Enter your name: " input(n)
    action("Go", go())
    action go() {
      var v : Visitor := Visitor{ name := n };
      stats.accesses := stats.accesses + 1;
      v.persist();
      return greet(v); } } }

define page greet(v : Visitor) {
  "Hello, " output(v.name)
  " you are visitor number " output(stats.accesses)
}
```

Fig. 24. Interaction sequence using pages in WebDSL.

of continuations, in particular the `call/cc` feature of Scheme. This approach has subsequently been adopted and refined in the PLT Scheme web server [67]. The Seaside Smalltalk web programming environment uses callbacks with closures to model control flow [40]. The OCaml web framework Ocsigen uses continuation passing style and stores continuations server-side on disk between requests [9]. The WASH [92] framework uses a monad to capture the continuation of a response. While continuations appear to be a very elegant formalization of sequential series of interactions with a single user, it is not clear that continuations can also be used to capture interactions involving (many) different users over multiple sessions as is needed for implementing workflows.

The Seam [56, 74] framework, which WebDSL targets, supports a notion of *conversations* to deal with the problem of keeping state in different threads of the same session separate. The solution here is basically to encode the continuation in a combination of data and context, i.e., the page being visited. In WebDSL it has not appeared necessary yet to build on this mechanism. First of all, the typical interaction that consists of presenting a form and receiving its inputs can be realized with a single page definition (based on the JSF facilities for forms). Next, WebDSL has session entities for storing data relevant for all interactions in a session (a feature not discussed in this paper). We have chosen to model state in sequential interactions, as well as in more complex interaction scenarios such as workflows, using regular WebDSL entities. Figure 24 illustrates this by encoding the Mawl example discussed above (including the forms for presentation). The definition introduces a `Counter` entity to keep track of the number of visits using

an application global variable. The `Visitor` entity is used to store the name of a visitor obtained in the `getname` page. The object is then passed as a parameter of the `greet` page, where it is used to obtain the name. The `go()` action of the `getname` page creates the `Visitor` object and *makes it persistent*. This is the difference with the Mawl approach, where the session data is transient and restricted to the session. The advantage is that interactions become naturally persistent such that users can come back to an interaction in later sessions. Scenarios in which multiple stakeholders in different roles need to interact are naturally modeled in this style as well. Using an appropriate access control policy, the visibility of the objects can be restricted. While this mechanism provides flexible expressivity for implementing all kinds of control flows, we will consider adding higher-level abstractions for defining complex workflows. For short-lived conversations (e.g. filling in a multi-page form) it would still be useful to have in-memory non-persistent (transient) state, for which the Seam conversations model may be the right implementation solution.

10.9 Testing

An important open issue is the testing of web application developed with WebDSL. We need two types of tests. First, regression testing for the language and generator, is needed to make sure that the implementations generated by the generator are correct. For this purpose we would need to make a set of small test applications, that exercise specific constructs of the language. Secondly, WebDSL application developers need to test that their program satisfies its specification. It should not be necessary to test basic, low-level functionality, since correctness of the language construct should ensure their functionality. Thus, application tests should test application behavior. For both kinds of tests we need a DSL for expressing high-level tests.

10.10 Model-View-Controller

WebDSL programs combine the user interface implementation with the logic associated with user interface events. This design violates the model-view-controller pattern, which dictates that the user interface (view) should be separated from the controller [48]. There are several reasons why such a separation is desirable.

First, to distribute functionality over different nodes in the network in order to distribute the load to more than one server. Typically, the application is separated into tiers, each of which is implemented as a process on a different server. This goal is not precluded by the WebDSL approach. Even while an application *definition* integrates UI and logic, in the *implementation* these are separated into JSF pages and session beans, which are designed for a layered architecture.

Secondly, motivation for applying the MVC pattern is to be able to use different views with the same logic and/or to let developers with different skills work on view and controller separately. This requires not so much that logic and view should be separated (as a policy), but rather requires mechanisms that

allows them to be separated when that is necessary. The template mechanism of WebDSL allows views and actions, performed in those views, to be implemented separately, where the view calls an abstract template, defined by the controller, as illustrated in the following example:

```

define view(field1 : String, field2 : String) {
  form{ input(field1) input(field2) submit(field1, field2) }
}
define control(m : Model) {
  view(o.field1, o.field2)
  define submit(field1 : String, field2 : String) {
    action("Submit", submit())
    action submit(){ m.field1 := field1; m.field2 := field2; }
  }
}

```

Here the `view` template definition can be an elaborate structure definition, which only takes basic data types as input values. Invoking an action is delegated to an abstract `submit` template. The `control` uses the `view` to display the data, and defines a concrete `submit` to implement the action.

11 Discussion: Language Engineering Paradigms

An application domain is a collection of concepts. The description of an application in a domain is a collection of statements involving those concepts using the ‘language of the domain’. For example, ‘make a page that displays the properties of this object’ is a sentence in the domain of web applications. A conceptual domain language can be implemented in many different forms, even as a library in a ‘conventional’ general-purpose programming language. *Language engineering* is concerned with the design and implementation of languages in all their different forms. This section provides a brief survey of existing language engineering paradigms and their impact on the language development process. A complete and in-depth survey of language engineering is out of the scope of this paper. There are many surveys on domain-specific languages and their development from different perspectives, including [86, 85, 99, 33, 73, 32, 87, 60].

Approaches The discussion is organized by considering the distance of the approach to the implementation platform. *Application frameworks* are based on the concept of ‘a library as a language’ (Section 11.1). *Domain-specific embedded languages* encode a language using the syntactic facilities of the host language (Section 11.2). *Interpreted DSLs* are separate languages, which are passed to an interpreter library (Section 11.3). *Domain-specific language extensions* add new syntax to a general purpose language (Section 11.4). *Compiled domain-specific languages* are defined completely separately from the implementation platform, and can in principle be translated to more than one platform (Section 11.5).

Technologies These approaches entail fundamentally different architectures for capturing domain-specific knowledge with implications for development and usage. Somewhat orthogonal to these basic approaches are specific *technologies* for realizing them. Technological frameworks are typically designed for use with a particular approach, but their use may be stretched to other approaches as well. Section 11.6 outlines the main ingredients of language implementations, and gives an overview of some typical tool sets.

Criteria In the previous section we applied the set of evaluation criteria to WebDSL. In this section we use these criteria to compare the properties of different approaches. Of course, it is not possible to make generic statements about all products of a particular approach. For example, the quality of generated code is not magically guaranteed by using a particular generator technology, but will depend the efforts of the generator developer performing meticulous research into the properties of the target platform. However, certain approaches may facilitate better results in some area than others.

11.1 Application Frameworks

The most accessible approach to encapsulating domain knowledge is by means of a library or (object-oriented) framework. The language defined by a library is the *application programmer's interface (API)*. That is, a library provides data structures (objects) with operations (methods). The basic elements of the language are the calls to operations. The composition mechanism is generic, that is, not specific for the domain. For example, an object-oriented programming language provides object creation, method calls, subclassing, and inversion of control [46].

Developing Frameworks An application framework is directly implemented in a third-generation general-purpose programming language such as Java. Thus, framework development can directly use all the productivity advantages provided by modern programming languages and their interactive development environments. While frameworks are developed in a basic programming language, designing a good framework is not easy and requires well trained software developers. However, there is a rich literature with design patterns [48] for developing object-oriented software and frameworks. Maintenance of frameworks is tricky when many client applications exist. Changing the interface breaks the build of client code, but only changing the implementation may not be safe either, since client code may depend on implementation details of the framework.

Developing with Frameworks The primary advantage of a framework compared to other approaches discussed later, is that they integrate well with other programming tasks. However, the implementation technology does not support domain-specific verification. Only constraints that can be encoded in the host

type system can be checked at compile time. Frameworks are expected to cover a complete (technical) domain, which tends to make them large and complex. The expressivity of a framework is low, as the notation based on the generic composition mechanisms of the host language are typically not tuned to the application domain. Modern frameworks such as Hibernate [10] and Seam [56, 74] are fairly high-level due to the use of *annotations* and dependency injection, which are targeted by run-time or deployment-time compilation and instrumentation. Software developed with a particular framework is not portable to a different platform. The framework ties client code to its host language. To support all possible functionality, frameworks may provide multiple layers of abstractions which are not removed by the compiler. The internal structure is not completely encapsulated and are for example manifest in stack traces produced by exceptions. Frameworks use mechanisms such as inheritance and annotation processing to allow client code to specialize the generic functionality it provides. This form of extensibility is built into the infrastructure.

11.2 Domain-Specific Embedded Languages

While one could view the API provided by a framework as a language, this is not typically the perspective of programmers. The idea behind *domain-specific embedded languages (DSELS)* is to build DSLs in the form of libraries in a general-purpose language. Hudak argues that combinator libraries in higher-order functional languages such as Haskell are especially suited for building domain-specific languages [55]. In essence, DSELS are the same as frameworks, but the differences in abstraction mechanisms between object-oriented and functional languages, give them a different flavor.

Developing Combinator Libraries The core advantage of DSELS is the reuse of abstraction mechanisms in the host language. It is not necessary to design and implement a mechanism for functional or modular abstraction. Also control-flow constructs are easily defined in a *lazy* functional language such as Haskell. Infix operators get a long way to approach domain-specific notation. Thus, the developer can concentrate on the truly domain-specific aspects of the language. Furthermore, there is no need to write code generators; language ‘constructs’ are combinators, which are defined by means of function definition.

Developing with Combinator Libraries DSELS share with frameworks the good integration with the host language, the lack of portability, and the lack of domain-specific verification, syntax, optimization, and error messages. However, domain-specific type checking can be achieved to some extent using phantom types [70].

11.3 Interpreted Domain-Specific Languages

Interpreted DSLs are proper languages with their own syntax and semantics separately defined from a host language by means of an interpreter, which executes sentences.

Developing Interpreters Developing an interpreted language requires development of a syntax (with corresponding parser) and the interpreter itself. The problem of building an interpreter can be mitigated by organizing an interpreter as a factory that creates instantiations of a class hierarchy. After initialization of the objects, it functions as a ‘normal’ program. Thus, an existing framework can be given domain-specific notation through an interpreter.

Developing with Interpreters When the interpreter is built into a library, it can be invoked from a general-purpose program and may fit in a software development approach otherwise based on a general-purpose language. For example, SQL and XSLT can be used in this fashion. Models can be executed on any platform with an interpreter, which entails that the interpreter is needed at run-time. It is typically not easy to support interaction between interpreted code and code in a GPL. However, a combination of the factory approach mentioned above and reflection may support some form of interaction, e.g. a foreign function interface that supports calling (host) library functions from the DSL program. Usually, interpretation incurs overhead compared to compiled code, since the interpreter must parse and inspect the (abstract) representation of the model. Extension of the language may not be easy, as it requires extension of the interpreter.

11.4 Domain-Specific Language Extension

The idea of *domain-specific language extension* is to extend a general-purpose *host* language with domain-specific *guest* notation. In contrast to domain-specific embedded languages, the syntax of the host language is actually extended to truly accommodate the domain-specific notation. An *assimilation* transformation maps extension back to base language [21]. This can be implemented as a pre-processor of the base language or by a proper extension of the host language compiler. Dmitriev advocates this approach with the name *language oriented programming* [36],

Developing Language Extensions Developing a good language extension implementation is difficult, since it requires extension or reimplementing of a considerable part of the host language infrastructure. First, a complete syntax definition of the host language is needed, which can be extended with the domain-specific notation. This requires some form of syntactic extensibility. Second, the extension needs to be implemented either by extending the host compiler or by means of a translation down to the base language. (This is basically similar

to DSL compilation, discussed below.) Third, the type checker of the host language needs to be extended. There are a number of approaches for realizing this scenario.

Extensible languages are languages that are prepared (to some extent) for extension with new syntactic constructs. The prototypical example of an extensible language is Scheme, which provides macros for introducing new ‘syntactic forms’ [26]. Macro definitions define a translation from the new language construct to more basic language constructs. Macros are applied by the interpreter. Thus, programs can introduce and use extensions. Other incarnations of this approach are Template Haskell [83], which supports compile-time generation of program fragments (but no syntactic extensions), and Converge [95], which provides compile-time meta-programming support for the definition of new embedded languages and their assimilation. Language workbenches [47] are IDEs supporting the creation of macro-like language extensions.

Pre-processing is another popular approach to realize language extension. The advantage over extensible languages is that a pre-processor can be built for any base language, also those not designed with macro-like facilities. An example of a pre-processor based language extension approach is MetaBorg [21], which relies on the modularity of SDF to create the syntactic extension of a language and on Stratego for expressing assimilation rules. MetaBorg extends the framework approach to DSL implementation with proper syntax, thus providing a domain-specific notation for the abstract syntax defined by an API. A particular instance of MetaBorg is StringBorg [16], a technique for providing proper syntax checking for interpreted DSLs such as SQL. Instead of encoding queries in string literals, which makes applications vulnerable to injection attacks, queries are defined in an embedded DSL, which is syntactically checked. Under the hood a string representation of the query is eventually constructed, but without the risks of malicious injections. The disadvantage of pre-processors is that they do usually not provide proper integration with the semantic checking of the host language, since that requires re-implementation of those parts of the compiler in the pre-processor.

Extensible compilers avoid the incompleteness of pre-processors by exposing the internal structure of the compiler to extensions. Thus, the implementation of an extension can extend the type checker to guarantee that only statically correct programs are compiled, and that error messages are phrased in terms of the source program, not the assimilated one. Examples of extensible compilers for Java are Polyglot [75], Silver [109] and JastAddJ [45]. The latter two are based on extensible attribute grammars formalisms, which supports declarative and compositional specification of the type system of a language [109, 44].

The disadvantage of an extensible compiler is that an extension is based on white box reuse of the base compiler, rather than a semantic description of the language. This requires intimate knowledge of the implementation of the compiler and exposes extensions to changes in the implementation. The approach of *compilation by normalization* [58] avoids this problem by providing a mixed source and byte code language as target for a pre-processor. By means of tracing

information, type and run-time errors can be reported in terms of the original source code. By exposing the target language as part of the source language, pre-processors can produce low-level implementations where needed without invasive extension of a compiler.

While extending compilers to support extended languages is understood to some extent, modern languages require rich interactive development environments. Exploration of the design and implementation of such IDEs for embedded languages is only recently started [59].

Developing with Language Extensions Provided that also the IDE is extended, a general purpose language with domain-specific extensions can provide a very expressive programming environment that allows to use a DSL where needed, and the general-purpose language for ‘normal’ programming. As is the case with frameworks and combinator libraries, models in an embedded languages are tied to their host language and cannot be used with a different platform. It is important that assimilations do not leak, that is, expose the developer to the result of translating embedded models to host code, for example in the form of error messages at compilation or run-time.

11.5 Compiled Domain-Specific Languages

WebDSL falls in the category of *compiled domain-specific languages*, that is, a language dedicated to a particular application domain, not embedded in a particular host language or implementation platform. Models in such languages are implemented by *compilation* or *code generation*, i.e. translation to a program in some target language.

The main disadvantage of the approach is that implementation of a DSL compiler can be a significant undertaking. Unlike DSEs, there is no linguistic reuse of abstraction facilities of a host language, implying that all the basic constructs that a language requires, need to be implemented in addition to the actual domain-specific elements. For example, WebDSL has an action language, which is a subset of imperative language with object-oriented elements.

The main advantage is that the language can be designed to be independent of the target platform, and that models in the language can thus be implemented on more than one platform. To achieve portability one should guard against leakage of implementation details from the target platform. While abstractions cannot be borrowed from a host language, the gain is that there are no constraints imposed on the design of abstractions. Furthermore, the compiler can provide domain-specific error checking and optimization.

There are many variant approaches including generative programming [33, 32] and model-driven engineering [82] and technologies for realizing them. However, the essential architecture is the same in all approaches. In addition, to proper DSL compilers there are less complete variations, *scaffolding* and *light weight languages*.

Scaffolding The term ‘code generation’ is understood in some contexts as the generation of incomplete code skeletons from configuration data, e.g. a UML model. For example, from a class diagram a set of Java classes is generated with the attributes and operations as specified in the diagram, but the implementation of the methods needs to be filled in. Another example is Ruby on Rails [93], a framework for web application implementation based on the Ruby programming language, which generates boilerplate code from a database schema.

The advantage of a scaffolding generator is that it is relatively easy to build. There is no need not design and implement abstractions for areas where the developer is expected to do heavy customization. The big disadvantage is that it requires maintenance at the level of the generated code. This requires round-trip engineering or carefully marking in the generated code which parts were generated and which parts customized, such that only generated parts can be re-generated. However, this will remain fragile and prone to inconsistencies between model and code. Often, re-generation is not supported as it carries a substantially higher implementation cost than the scaffolding generator itself. More importantly, the approach exposes the developer to the implementation, which breaks encapsulation of the generator and limits its scalability.

Lightweight Languages Another category of DSL implementations is that of *lightweight languages* [86]. These are languages with a very restricted scope, possibly used in a single software project. Such languages are economically viable because they are implemented cheaply, for instance using regular expressions in Perl. The translation consists of simple local translations and does not include static error checking, placing the burden of creating a correct model on the programmer. This approach does not scale to languages that need to be used in many projects and/or by many developers.

Heavyweight Languages A proper domain-specific language is constructed according to well established architectural patterns for compilers [3]. A generator consists of a front-end that parses the model from a concrete syntax representation (be it a visual or textual) to an abstract representation. This representation is subsequently checked against the static semantic constraints. After optionally applying a number of transformations to the model itself, it is translated to code in some target language. There is a long tradition of tool kits with DSLs for reducing the effort of building compilers, e.g. [63, 57, 81, 49, 62, 5]. Stratego/XT fits in this tradition and so do the various MDE tool sets introduced recently. Within these architectural boundaries there are different styles for implementing the various aspects of a generator.

11.6 Language Engineering Tools

For the development of a framework or combinator library only an appropriate host language is required. For the other approaches discussed above, i.e. interpreted DSLs, language extensions, and compiled DSLs, tool infrastructure for

language engineering is required. A language implementation requires parsing, analysis, transformation, generation, and/or interpretation as discussed in Section 3.4. As with any domain, these tasks can be expressed in general purpose programming languages. However, by its nature this domain is a fertile breeding ground for tools and domain-specific languages. The rest of this section gives a brief summary of the main variation points and illustrates how some existing tool sets bind these variation points.

Parsing The definition of a textual DSL requires a parser that turns the text of a model into a structured representation, which can be used for further processing. Most parser generators are based on deterministic subsets of the set of all context-free grammars, such as LL (recursive descent) implemented by ANTLR [77] or LR [64] as implemented by YACC [57]. While these subsets guarantee unambiguous syntax definitions and (near) linear time parsing, the restrictions can require awkward encodings of linguistic constructs. Generalized parsing algorithms such as Earley [41], GLR [94], or SGLR [101] do not suffer these limitations. However, the support for error messages and error recovery is typically not as good as with deterministic parsers.

Model Representation The abstract representation of a model is the data structure that analysis, transformation, and generation operate on. The properties of a representation determine how costly (in terms of time and space) it is to perform certain operations. Unfortunately there is no single representation that makes all operations equally cheap [104].

With a functional representation such as the Annotated Terms (ATerms) used in Stratego [96], or the algebraic data types in (pure) functional languages such as Haskell [79], performing transformations is cheap since copying of subtrees constitutes of copying references, instead of cloning. Also, a functional representation is *persistent* in the sense that a transformation does not destroy the old representation. However, the directed acyclic graph (DAG) structure does not admit extending the tree with references to other parts of the tree. Hence, context information needs to be stored in symbol tables or similar data structures.

In contrast, graph structures (including object graphs in object-oriented languages) allow extension of nodes with arbitrary cross references in the graph, which can be used to make context information into local information. For example, add a reference from a variable to its declaration. This makes the result of analyses much easier to express. The downside is that transformations on graphs are not persistent, i.e. require a destructive update, or copying of the entire graph structure. Meta models in modeling frameworks such as EMF [23] define graph structures, and thus require graph transformation solutions. Of course, EMF can be used to model more restricted representations, including functional representations.

Analysis and Transformation Analysis and transformations of models are used to prepare the model for code generation, for example by enriching it with type annotations (Section 7.1) or by desugaring high-level constructs as lower-level ones (Section 9). In principle, analyses and transformations can be expressed in any functional, imperative, or logical programming language. However, specialized transformation languages may allow more declarative and/or more concise expression of transformations. As discussed above, the representation of models has consequences for the applicable transformation paradigms.

Term rewriting [6] is a useful paradigm for transformation of a functional representation. Rewrite rules are declarative specifications of one step transformations. Exhaustive application of rewrite rules is performed by an implied rewriting strategy. Rewriting is useful for repeated, cascading transformations such as desugaring, where model elements are rewritten to combinations of other model elements, which can subsequently again be rewritten. This approach requires an easy way to construct large patterns of model elements. Concrete object syntax [102] enables the natural construction of model fragments of hundreds of nodes, which is extremely tedious using abstract object construction techniques. In pure term rewriting, rewrite rules are applied exhaustively to the entire term. Because of non-confluence and non-termination more control over the application of rules may be necessary. Various approaches for controlling rules have been developed [104], among which the programmable rewriting strategies of Stratego.

Analysis typically requires non-local information, e.g. the declaration of a variable and its use. While rewriting approaches can express context-sensitive analyses and transformations, e.g. the type checker in Section 7.1), a more declarative approach to expressing analyses is provided by *attribute grammars* [65], which are supported by systems such as JastAdd [44] and Silver [109]. An attribute grammar assigns values to attributes of tree nodes. Attribute values are defined by means of attribute equations in terms of other attributes. The scheduling of attribute value computations is left to the attribute grammar compiler. The value of an attribute may depend on the entire tree. Applying just a single local transformation in principle invalidates all attribute values in the tree, and requires re-computing all attribute values. Therefore, attribute grammars are useful for performing analyses of static trees, while rewriting approaches are more suitable for performing transformations. It is a research challenge to find a combination of the formalisms such that analysis and transformation can be mixed.

There are numerous approaches to transformation of graph representations as occur in modeling approaches. Czarnecki and Helsen [34] give an extensive survey of features of model transformation approaches.

Generation Many tool sets provide a *template engine* such as Velocity [89], StringTemplate [78], or Xpand [43] for translation of models to program text. A template is a quotation of a static piece of code. Variability in the code is realized by means of anti-quotation expressions that allow insertion of names,

expressions, or sub-templates specialized for the input model. Templates are an improvement over the practice of printing string literals in a regular programming language, which require escaping of special characters and often do not support multi-line fragments. Textual templates do not check the syntax of the quoted code fragments. This makes the technique easily adaptable to any target language. However, it may result in syntactically incorrect code being generated. More importantly, the generator does not have access to the structure of the generated code. This makes it impossible to apply transformations, e.g. instrumentation, to the generated code.

The approach used in this paper can be characterized as ‘code generation by model transformation’ [54]. The generator produces a model representation of the target program, which is amenable to further transformation. Producing large fragments of target models is often inconvenient using the abstract syntax notation. Concrete object syntax combines the surface syntax used in a template engine with the underlying model representation of the generated code. Implementation of concrete syntax requires a grammar formalism that supports the modular composition of the context-free and lexical syntax of languages [102, 22]. Eventually, the model representation needs to be rendered as text. This is a straightforward one-to-one rendering of each node also known as pretty-printing.

Tool Sets A tool set for language engineering provides a particular combination of support choosing some point in the design space sketched above. In addition, this configuration is realized on a particular programming platform, which may be a specific operating system and usually a particular programming language. Thus, while in principle the architectures of the tool sets is comparable, in practice the choice for a particular tool set may be based on other factors than just the techniques supported. Furthermore, for branding purposes, tool producers, be it industrial, open source, or academic, tend to emphasize the differences between tools, rather than their commonalities. The following list of tool sets gives an impression of the variability in the domain, without pretending to be complete.

Rewriting languages

- ASF+SDF [97] is a compiled language based on first-order term rewriting with traversal functions, providing concrete syntax for patterns in rules.
- TXL [31] is an interpreted, rule-based functional language with concrete syntax patterns, and a form of deep application of rules.
- Stratego/XT [17] is a compiled transformation language based on rewriting with programmable rewriting strategies; rules can use abstract or concrete syntax.
- Strafunski [68] is a combinator library for strategic programming (in the sense of Stratego) in Haskell.

Attribute grammar formalisms

- Eli [50] is a composition of language processing tools including statically scheduled attribute grammars.
- JastAdd [44] is a compiled language based on rewriteable reference attributed grammars.
- Silver [109] is a compiled attribute grammar formalism with forwarding and dynamic scheduling of attribute evaluation.

Modelware

- Open ArchitectureWare [43] is an Eclipse-based tool set for textual DSL definition and code generation. It uses EMF [23] for the representation of models. The xText grammar formalism, which is based on ANTLR, is used to define textual syntax of DSLs and the generation of an Eclipse editor plugin. The xTend ‘functional’ language is used for model transformation, and the xPand textual template language is used for model to text transformation.
- MetaCase [60] supports the creation of visual domain-specific modeling languages.
- Visual Studio DSL Tools [30] is a meta-modeling framework for visual modeling languages. Code generation is achieved using a textual template engine.

12 Discussion: Language Engineering Challenges

A discussion of some challenges for research in language engineering.

DSL Interaction WebDSL is a composition of several languages, that is, a data model language, a presentation language, a query language (HQL), and an expression and action language. The language is a good basis for further abstractions, such as ones for access control and workflow. Template definitions and modules support the creation of reusable components. While these different languages support different aspects of web applications, they are integrated into a composite language to ensure smooth interaction between the different aspects; as opposed to the heterogeneous architecture of web applications implemented in Java. Although inspired by similar features in other languages, the language was designed and implemented from scratch. It would be useful to have language design and implementation patterns to be reused when creating new languages, if possible supported by tools or reusable libraries of language components.

A particular issue that arises in domain-specific language engineering is the design of language interaction. Software development typically requires the interaction between several technical and application domains. How can programs in different languages refer to each other? Can modules be compiled, or even type checked separately? Warmer [107] has developed a collection of DSLs for web applications using the Microsoft DSL Tools. In that work the assumption is that separate models are compiled to separate target files. Interaction of models is achieved using a registry that records interface information (key, value pairs). This approach precludes weaving of code from different models. Mak [71] has

explored the separation and interaction of languages in a variant of WebDSL. Basically, the separation was into a data model language and presentation language, which map to separate target code components.

Development Environment Software developers, especially those developing in Java or C# are accustomed to sophisticated development environments (IDEs), which help the programmer by means of syntax highlighting, cross-referencing, access to documentation, and code completion. When developing a new DSL, the barrier to being used can be lowered considerably, if such interactive support would be available as well. The challenge here is to generate from the definition of a language an IDE, for example by creating an Eclipse plug-in supporting syntax highlighting, syntax checking, typechecking, refactoring, code completion, and cross-referencing. Despite research projects such as the Synthesizer Generator [81] and the ASF+SDF MetaEnvironment [62], the creation of an IDE for a new language remains a laborious process. The Eclipse IDE Meta-tooling Platform [1] may reduce the effort to develop IDEs for new languages. A first step on the path to the integration of the language definition techniques used in this paper (Stratego and SDF), is the generation of Eclipse plug-ins based on the IMP framework from SDF definitions [59].

Deployment A DSL generator only automates one step in the development process of a software system. While the generator encapsulates knowledge about developing applications in the domain, more knowledge is required for successfully deploying an application. Therefore, a good DSL should also hide irrelevant deployment details. Ideally, the DSL programming environment offers a virtual machine for operating DSL programs, which completely hides its run-time system. Thus, in the domain of web applications such a virtual machine would appear to run WebDSL applications directly, and behind the scenes generate the Java/XML implementation code, compile it, and activate the application server to run the application. The Nix software deployment system [39, 37] provides a suitable infrastructure for realizing this scenario. Using a functional language, deployment configurations from source builds to service activation can be described [38]. Using this approach a first experimental setup has been created for deploying WebDSL applications, which is being used to deploy the `webdsl.org` website.

Extensibility A language should be designed for growth [88] in order to accommodate future requirements. Therefore, the implementation of a language should be easily extensible with new basic types, new constructs, new abstractions, and new sub-languages. Systems such as Silver [109], JastAdd [44], and Stratego/XT [103] (used in this paper), provide source level extensibility. That is, a language definition can be separated into modules and new features can be implemented by providing new modules. However, the new combination needs to be compiled from source *as a whole*. True extensibility would entail that *users*

of the language can combine extensions provided by different producers for a particular application without recompiling the generator. This requires separate binary extensibility of language definitions and generators.

Evolution The introduction of domain-specific languages can greatly improve the evolution of software by drastically reducing the amount of source code needed for systems. Paradoxically, reliance on DSLs also introduces a new software evolution problem. The number of languages in which software is written increases, requiring developers with knowledge of multiple languages [91]. Furthermore, while software applications may become easier to maintain, the implementations of the languages need to be maintained as well [98]. A problem that is seen as one of the factors for the failure of fourth generation languages. The next paragraphs discuss a number of challenges for evolution of domain-specific languages.

Data Migration Evolving applications based on DSLs should become easier. The size of an application is an order of magnitude smaller than before, which should make understanding and modifying programs easier. Complete code generation ensures that a complete system can be generated after modifying the DSL program. However, the data models that are implemented as the database schemas of deployed applications may have changed, requiring the database to be migrated to the new data model. To ease the evolution of applications, it is necessary to automate data migration as much as possible. At least there should be a language for specification of the migration between two data models at the level of the data model language (abstracting from implementation details of the database schema). Furthermore, the mapping between two data models could be inferred to some extent by considering the two versions.

Model Migration The problem of data migration also plays a role on a level higher-up in the modeling hierarchy. Changing the definition of a DSL requires adapting existing DSL models. To increase the acceptability of DSL evolution, it is desirable to support language changes with automatic conversion tools. First of all, that requires the definition of a transformation from models in the old language to models in that new language such that the new models have the same semantics as the old models. Supporting such semantics preserving transformations, requires the new language to at least support the functionality of the old language, which imposes some constraints on evolution. As in the case of data migration it would be desirable if the migration of models can be derived from the evolution of the grammar. In practice, language designers take great care to design language changes to be backwards compatible. Better migration solutions will enable language designers to make more drastic (re)design decisions, which are sometimes needed when insight in the domain grows.

An important practical consideration in the migration of programs is the treatment of white-space and comments (layout). Developers do not appreciate the look of their programs to be drastically changed by automatic transformations. As a result, a semantics preserving transformation on the abstract syntax

structure of a program is not sufficient. One solution direction is to support transformation with layout preservation. However, true layout preservation is not a solvable problem, since comments in programs do not have a formal relation to the surrounding code. Instead it would be a good idea to reduce the role of layout in languages. First, by making comments part of the syntactic structure, it can be treated like any other structures in transformations. Next, domain-specific languages should be designed to support self documenting code. After all, one of the ideas of DSLs is that they should express high-level application concerns, not implementation details. Finally, introducing enforceable coding standards (for layout) can eliminate the problem of re-formatting. (Note that these issues hold for visual (diagrammatic) languages as much as they do for textual languages.)

Abstraction Evolution A particular variant of DSL evolution is the addition of new abstractions to the language. In that case it may be worthwhile to transform existing DSL models to use the new abstractions. This requires recognizing the use of the implementation patterns that the new abstraction mechanism abstracts from. Semi-automatic support for pattern recognition and subsequent transformation would be useful to support developers in migrating to the higher-level abstractions.

Harvesting from Legacy Code Finally, after having developed a new DSL, it may be necessary to migrate existing legacy applications to the new DSL, which requires recognizing implementation patterns in legacy code. Even while a DSL design may be based on the abstraction of implementation patterns, these patterns may not be used exactly in an existing code base. As a concrete case, consider transforming legacy EJB applications to WebDSL programs, where JSF pages are translated to page definitions, entity classes to entity declarations, and session beans to page actions.

13 Conclusion

This paper has presented a case study in domain-specific language engineering. Based on this experience let's make an attempt at answering the questions 'when and how to develop a domain-specific language?'

When to develop a DSL? Starting to develop a DSL should only be done when there is a good understanding of the application domain and there exists a considerable code base for systems in the domain. That code base should exhibit clear signs of inadequate abstraction facilities in the form of boilerplate code in large quantities, even if best practices are being applied. Another sign is that mechanisms that have been introduced to raise the level of abstraction elude the verification facilities of the implementation language. Typical examples are XML configuration files, interpreter literal strings (SQL queries), and dependency injection annotations.

How to develop a DSL? Choose a high-level target technology; the DSL should not readdress problems that have already been solved by existing technology. Start with considering relatively large chunks of programs as candidate patterns. Study and understand the technology and recognize common patterns. Set up a basic generator early on. That makes it easy to experiment with alternative implementation strategies in the target architecture without having to write a lot of code. Do not overspecialize syntax. For example, a separate syntactic construct for each page element such as `section`, `header`, `list` in WebDSL, would lead to hard wiring in such constructs and a much larger implementation. Do not overgeneralize syntax either. Ending up with a completely generic syntax such as XML does not lead to readable programs. A core language that captures the essential operations of the domain is essential for achieving good coverage. But do not try to identify a core language from the start. The result may be too close to the target target technology. For example, a modeling language that covers all EJB concepts provides 100% coverage, but is too low-level. Extend the core language with syntactic abstractions that allow concise expression. Include facilities to build a library, such as modules for organization of the code base and parametric abstraction over DSL fragments.

Acknowledgments

In August 2006 Ralf Lämmel and Joost Visser invited me to give a tutorial at the GTTSE summer school to be held in July 2007. This invitation provided a perfect target and outlet for the rather uncertain sabbatical project that I had conceived to build a domain-specific language for web applications. Along the way I had many inspiring discussions about various aspects of this enterprise and received feedback on drafts of this paper. I would like to thank the following people for their input (in more or less chronological order of appearance): Martin Bravenboer, Jos Warmer, Sander Mak, William Cook, Anneke Kleppe, Jonathan Joubert, Rob Schellhorn, Danny Groenewegen, Zef Hemel, Paul Klint, Jan Heering, Ron Kersic, Nicolae Vintae, Charles Consel, and the GTTSE'07 reviewers. The research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. Eclipse IDE Meta-tooling Platform (IMP). <http://www.eclipse.org/proposals/imp/>.
2. Echo web framework. <http://echo.nextapp.com/site/echo2>, July 2007.
3. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools*. Addison Wesley, Reading, Massachusetts, 1986.
4. D. L. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, 1999.
5. A. Augustejn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1993.

6. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
7. J. W. Backus. Automatic programming: properties and performance of FORTRAN systems I and II. In *Proceedings of the Symposium on the Mechanisation of Thought Processes*, Teddington, Middlesex, England, November 1958. The National Physical Laboratory.
8. J. W. Backus et al. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.
9. V. Balat. Ocsigen: typing web interaction with objective Caml. In A. Kennedy and F. Pottier, editors, *Proceedings of the ACM Workshop on ML*, pages 84–94, Portland, Oregon, USA, September 2006. ACM.
10. C. Bauer and G. King. *Java Persistence with Hibernate*. Manning, Greenwich, NY, USA, 2007.
11. K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.
12. J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.
13. C. Brabrand, A. Möller, M. Ricky, and M. I. Schwartzbach. PowerForms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4):205–314, December 2000. Kluwer.
14. C. Brabrand, A. Möller, and M. I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.
15. M. Bravenboer. Connecting XML processing and term rewriting with tree grammars. Master’s thesis, Utrecht University, Utrecht, The Netherlands, November 2003.
16. M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In J. Lawall, editor, *Generative Programming and Component Engineering (GPCE 2007)*, pages 3–12, New York, NY, USA, October 2007. ACM.
17. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. Components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM 2006)*, pages 95–99, Charleston, South Carolina, January 2006. ACM SIGPLAN.
18. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. *Stratego/XT Tutorial, Examples, and Reference Manual (latest)*. Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 2006. <http://www.strategoxt.org>.
19. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. Special issue on Experimental Systems and Tools.
20. M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.
21. M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.
22. M. Bravenboer and E. Visser. Designing syntax embeddings and assimilations for language libraries. In *MoDELS 2007 Satellite Events Proceedings*, Lecture Notes in Computer Science, Nashville, USA, 2008. Springer. Selected paper from the Workshop on Language Engineering (ATEM 2007).

23. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.
24. D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured english query language. In R. Rustin, editor, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control*, pages 249–264, Ann Arbor, Michigan, May 1974. ACM.
25. A. S. Christensen, A. Möller, and M. I. Schwartzbach. Extending Java for high-level web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.
26. W. Clinger. Macros in scheme. *SIGPLAN Lisp Pointers*, 4(4):17–23, 1991.
27. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
28. C. Consel. From a program family to a domain-specific language. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey, pages 19–29. Springer-Verlag, 2004.
29. C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Pisa, Italy, Sept. 1998.
30. S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison Wesley, 2007.
31. J. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
32. K. Czarnecki. Overview of generative software development. In J.-P. Bantre et al., editors, *Unconventional Programming Paradigms (UPP 2004)*, volume 3566 of *Lecture Notes in Computer Science*, pages 313–328, Mont Saint-Michel, France, 2005.
33. K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, New York, NY, USA, 2000.
34. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
35. M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000)*. University of Wollongong, Australia, 2000.
36. S. Dmitriev. Language Oriented Programming: The next programming paradigm. <http://www.onboard.jetbrains.com/articles/04/10/lop/>, 2004.
37. E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Utrecht University, Utrecht, The Netherlands, January 2006.
38. E. Dolstra, M. Bravenboer, and E. Visser. Service configuration management. In J. E. James Whitehead and A. P. Dahlqvist, editors, *12th International Workshop on Software Configuration Management (SCM-12)*, pages 83–98, Lisbon, Portugal, September 2005. ACM.
39. E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In J. Estublier and D. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004)*, pages 583–592, Edinburgh, Scotland, May 2004. IEEE Computer Society.

40. S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, pages 56–63, September/October 2007.
41. J. Earley. *An Efficient Context-free Parsing Algorithm*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1968. (See also [42]).
42. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
43. S. Efftinge, P. Friese, A. Haase, C. Kadura, B. Kolb, D. Moroff, K. Thoms, and M. Völter. *openArchitectureWare User Guide. Version 4.2*. www.openarchitectureware.org, 2007.
44. T. Ekman and G. Hedin. Rewritable reference attributed grammars. In M. Oder-sky, editor, *18th European Conference Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169, Oslo, Norway, July 2004. Springer.
45. T. Ekman and G. Hedin. The jastadd extensible java compiler. *SIGPLAN Notices*, 42(10):1–18, 2007.
46. M. Fowler. Inversion of control containers and the dependency injection pattern. <http://www.martinfowler.com/articles/injection.html>, January 2004.
47. M. Fowler. Language workbenches: the killer-app for domain specific languages? <http://www.martinfowler.com/articles/languageWorkbench.html>, 2005.
48. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
49. R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35:121–131, feb 1992.
50. R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: a complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, 1992.
51. J. Greenfield and K. Short. *Software Factories. Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
52. D. Groenewegen and E. Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In D. Schwabe and F. Curbera, editors, *International Conference on Web Engineering (ICWE 2008)*. IEEE CS Press, July 2008.
53. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
54. Z. Hemel, L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *International Conference on Model Transformation (ICMT 2008)*, *Lecture Notes in Computer Science*. Springer, June 2008.
55. P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28:196, 1996.
56. JBoss Seam. *Seam - Contextual Components. A Framework for Java EE 5*, 1.2.1.ga edition, 2007. <http://www.jboss.com/products/seam>.
57. S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.
58. L. Kats, M. Bravenboer, and E. Visser. Mixing source and bytecode. A case for compilation by normalization. In G. Kiczales, editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, Nashville, Tennessee, USA, October 2008. ACM Press.

59. L. C. L. Kats, K. T. Kalleberg, and E. Visser. Generating editors for embedded languages. integrating SGLR into IMP. In A. Johnstone and J. Vinju, editors, *Proceedings of the Eighth Workshop on Language Descriptions, Tools, and Applications (LDTA 2008)*, Budapest, Hungary, April 2008.
60. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling. Enabling Full Code Generation*. John Wiley & Sons, Inc., 2008.
61. S. Kent. Model driven engineering. In M. Butler, L. Petre, and K. Sere, editors, *Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer-Verlag, May 2002.
62. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.
63. D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, December 1964.
64. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.
65. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction in: *Mathematical Systems Theory* 5(1), pp. 95–96, Springer-Verlag, 1971.
66. D. E. Knuth. *The T_EXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts, 1984.
67. S. Krishnamurthi, P. W. Hopkins, J. A. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the plt scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.
68. R. Lämmel and J. Visser. Typed combinators for generic traversal. In *Proceedings Practical Aspects of Declarative Programming (PADL 2002)*, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag, January 2002.
69. L. Lamport. *LaTeX : A Documentation Preparation System*. Addison-Wesley, Reading, Massachusetts, 1986.
70. D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages (DSL 1999)*, pages 109–122, New York, NY, USA, 1999. ACM Press.
71. S. Mak. Developing interacting domain specific languages. Master’s thesis, Utrecht University, Utrecht, The Netherlands, November 2007. INF/SCR-07-20.
72. K. D. Mann. *JavaServer Faces in Action*. Manning, Greenwich, NY, USA, 2005.
73. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
74. J. F. Nusairat. *Beginning JBoss Seam*. Apress, New York, USA, 2007.
75. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, April 2003.
76. OMG Architecture Board ORMSC. Model driven architecture. OMG document number ormsc/2001-07-01, July 2001. Available from www.omg.org.
77. T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages by Terence Parr*. The Pragmatic Programmers, 2007.
78. T. J. Parr. 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, 2004. ACM.

79. S. Peyton Jones, editor. *Haskell98 Language and Libraries. The Revised Report*. Cambridge University Press, 2003.
80. C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *International Conference on Functional Programming (ICFP 2000)*, pages 23–33. ACM, 2000.
81. T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, NY, USA, 1988.
82. D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, February 2006.
83. T. Sheard and S. L. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16, October 2002.
84. C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006)*, pages 451–464, New York, NY, USA, 2006. ACM.
85. D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.
86. D. Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *USENIX Conference on Domain-Specific Languages*, pages 67–76, Berkeley, CA, Oct. 1997. USENIX Association.
87. T. Stahl and M. Völter. *Model-Driven Software Development*. Wiley, 2005.
88. G. L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999. (Text of invited talk at OOPSLA 1998).
89. T. Sturm, J. von Voss, and M. Boger. Generating code from uml with velocity templates. In *Proceedings of the 5th International Conference on The Unified Modeling Language (UML 2002)*, pages 150–161, London, UK, 2002. Springer-Verlag.
90. Sun Microsystems. *JSR 220: Enterprise JavaBeansTM, Version 3.0. Java Persistence API*, May 2 2006.
91. A. L. Tharp. The impact of fourth generation programming languages. *SIGCSE Bull.*, 16(2):37–44, 1984.
92. P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL 2002)*, volume 2257 of *Lecture Notes in Computer Science*, pages 192–208, Portland, OR, USA, January 2002. Springer.
93. D. Thomas and D. H. Hansson. *Agile Web Development with Rails*. The Pragmatic Bookshelf, 2005.
94. M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
95. L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems*, 2009. (To appear).
96. M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
97. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.
98. A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

99. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.
100. J. van Wijngaarden. Code generation from a domain specific language. Designing and implementing complex program transformations. Master’s thesis, Utrecht University, Utrecht, The Netherlands, July 2003. INF/SCR-03-29.
101. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
102. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
103. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.
104. E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.
105. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. 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, September 1998.
106. W3C. *Cascading Style Sheets, level 2. CSS2 Specification*, May 1998. <http://www.w3.org/TR/REC-CSS2/>.
107. J. Warmer. A model driven software factory using domain specific languages. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Model Driven Architecture — Foundations and Applications, Third European Conference (ECMDA-FA 2007)*, volume 4530 of *Lecture Notes in Computer Science*, pages 194–203. Springer, June 2007.
108. B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In M. Felleisen, editor, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 199–210, Nice, France, January 2007. ACM.
109. E. V. Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In E. Ernst, editor, *21st European Conference on Object-Oriented Programming (ECOOP 2007)*, volume 4609 of *Lecture Notes in Computer Science*, pages 575–599, Berlin, Germany, July 2007. Springer.

TUD-SERG-2008-023
ISSN 1872-5392

