SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:1253–1292
Published online 28 August 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.936

An approach for the systematic development of domain-specific languages

Mark Strembeck[1,*,†] and Uwe Zdun[2]

[1]*Institute of Information Systems, New Media Lab, Vienna University of Economics and Business (WU Vienna), Austria*
[2]*Distributed Systems Group, Information Systems Institute, Vienna University of Technology, Austria*

SUMMARY

Building tailored software systems for a particular application domain is a complex task. For this reason, *domain-specific languages* (DSLs) receive a constantly growing attention in recent years. So far the main focus of DSL research is on case studies and experience reports for the development of individual DSLs, design approaches and implementation techniques for DSLs, and the integration of DSLs with other software development approaches on a technical level. In this paper, we identify and describe the different activities that we conduct when engineering a DSL, and describe how these activities can be combined in order to define a tailored DSL engineering process. Our research results are based on the experiences we gained from multiple different DSL development projects and prototyping experiments. Copyright © 2009 John Wiley & Sons, Ltd.

Received 6 November 2008; Revised 5 June 2009; Accepted 13 July 2009

KEY WORDS: applied software engineering; model-driven software development; domain-specific languages; language engineering

1. INTRODUCTION

Software engineers have to deal with a wide variety of application domains originating from diverse economic and industrial sectors, such as airline companies, banks, car manufacturers,

*Correspondence to: Mark Strembeck, Institute of Information Systems, New Media Lab, Vienna University of Economics and Business (WU Vienna), Austria.
†E-mail: mark.strembeck@wu.ac.at

Contract/grant sponsor: European Union FP7 project COMPAS; contract/grant number: 215175

Copyright © 2009 John Wiley & Sons, Ltd.

energy suppliers, hospitals, newspaper publishers, or telecommunication companies. Moreover, such business domains are subdivided in organizational domains, such as accounting, customer support, human resources, procurement, research, or sales. In addition to business and organizational domains, software engineers have to deal with orthogonal, technical domains concerning different functions of a software system, such as access control, quality of service, system backup, or workflow definition. While these enumerations of domains are far from being complete, the examples illustrate the diversity of application domains existing on different abstraction layers.

In each of these domains, human users have to fulfill certain tasks. Throughout this paper we use the term *domain expert* to refer to a human user who is a professional in a particular domain, such as a stock analyst in the investment banking domain, a physician in the health-care domain, or a network administrator in the domain of computer networks. In addition, in many domains there is a fine-grained differentiation between domain experts specializing in different sub-domains. For example, in the banking domain there are different types of customer consultants (e.g. for ordinary customers, for corporate customers, or for private wealth management), different types of experts for investment banking (e.g. for mergers and acquisitions, financial risk management, or stock and derivative trading), and different types of management experts (such as business process experts or legal experts).

Building software systems that are tailored to the needs of a particular type of domain expert is already a complex task. This task becomes even more complex due to the continuing specialization of professional knowledge in all kinds of application domains. Therefore, the construction of software systems demands systematic development approaches that enable software engineers to cope with this situation. However, in spite of the advances in different software engineering disciplines, the problem of building tailored software systems for domain experts—in arbitrary domains—is far from being solved. For this reason, *domain-specific languages* (DSLs) receive a constantly growing attention in recent years (see, e.g. [1–6]). DSLs are specifically tailored for the needs of a particular problem or application domain. They promise that domain experts themselves can understand, validate, modify, test, and sometimes even develop DSL programs.

Unfortunately, the systematic development of DSLs is not fully tackled yet. Most existing approaches especially focus on technical facets of designing and implementing DSLs. That is, the main focus of DSL research is up to now on case studies and experience reports for the development of individual DSLs [5,7–9], design approaches and implementation techniques for DSLs [2,8,10–15], and the integration of DSLs with other software development approaches, such as programming languages for embedding DSLs [11,16], model-driven software development (MDSD) [1,3,17,18], or component-based software development [19]. However, the process of defining the DSL and the way this process is used in the context of a general-purpose software development approach, such as the Unified Process (UP) [20] or agile approaches like Extreme Programming (XP) [21], are seldom addressed.

In this paper, we identify and describe the different activities that we conduct when engineering a DSL. The activities documented in this paper are a result of the experiences we gained from numerous different DSL development projects and from multiple prototyping experiments. In particular, we performed an in-depth analysis of 14 DSL projects we participated in. In addition, we conducted several experiments with students and applied qualitative research methods such as observations, conversations, and interviews with professional developers to generalize and validate our results on a broad basis. Details about lessons learned from our experiences in various projects can be found in Section 8.

Our experiences in building DSLs especially result from projects in the MDSD context, projects where we built DSLs as extensions for dynamic programming languages, and projects where these two approaches overlap (see Sections 3 and 8). In addition, other approaches to building DSLs exist, such as the Grammarware approach [22], which is focusing on grammars and all software artifacts that directly depend on grammars. The work that is presented in this paper focuses on DSLs in an MDSD context and DSLs as extensions for dynamic programming languages; the Grammarware view on DSLs is not explicitly addressed in this paper.

Our analyses revealed the different activities that we conduct when engineering a DSL. These activities form a micro-process that can be tailored to various influencing factors of an actual DSL project. Examples of such influencing factors are the general-purpose software development approach used in a certain project (in which the micro-process for DSL development has to be embedded), the availability of domain experts, or the budget of the project. In particular, we found that one of the main challenges for software engineers is to increase the participation of domain experts in DSL design activities, in order to avoid misinterpretations of the domain and/or designing a DSL detached from the needs of the domain experts. In this paper, we describe both, the general activities that we perform to engineer a DSL, as well as activities we apply to tailor the DSL engineering process.

The remainder of this paper is structured as follows. In Section 2 we discuss related work to illustrate the state-of-the-art and how our approach contributes to the state-of-the-art. Next, in Section 3, we give essential background information on DSLs and introduce the different DSL concepts that we refer to in subsequent sections. In Section 4, we give an overview of our approach for the systematic development of DSLs. Subsequently, Section 5 presents the details of our approach by illustrating the individual process activities for DSL engineering using one specific variant of the DSL engineering process. Next, in Section 6, we describe how the DSL development process can be tailored in other ways than the prototypical tailoring presented in Section 5. To exemplify our approach, Section 7 provides an illustrative example that describes the engineering of a DSL for role-based access control (RBAC). Section 8 describes lessons learned from various projects, before Section 9 concludes the paper.

2. RELATED WORK

To the best of our knowledge, the work presented in this paper is the first approach to identify the development activities and describe a generic process for DSL development. However, a number of previous approaches also address specific process aspects of DSL development. In this section, we summarize them and compare them to our approach.

2.1. Related work on patterns, best practices, and lessons learned in DSL development

Several authors introduced patterns and pattern languages that can be applied in DSL development. This includes patterns for the design and implementation of DSLs [14] and patterns for evolving frameworks into DSLs [23,24]. A pattern is a time-proven solution to a recurring design problem. The patterns not only describe how a DSL is developed, but also why it is developed in a specific way. In this sense, they complement our approach and can be applied in the context of our approach. In addition, patterns in pattern languages also describe the potential sequences in which the patterns can be applied. This includes a process aspect, similar to our work. However, the named pattern languages rather focus on specific design issues, whereas our approach focuses on describing the overall micro-process for DSL development.

The patterns for the design and implementation of DSLs by Spinellis [14] have been used by Mernik *et al.* [2], who provide a survey of decision factors for the decision, analysis, design, and implementation phases of DSL development. These decision factors can be considered during the DSL development. For example, the decision factor *Notation* deals with the consideration whether the DSL should provide a new or existing domain notation. For a few decision factors, Mernik *et al.* suggest implementation guidelines, which describe a process aspect. This process aspect focuses on the general design decision and implementation issue, and not the general DSL development micro-process as in our approach.

In addition to our approach and the different patterns' approaches, some other contributions are also based on observations from practical DSL development. For example, Luoma *et al.* [8] conducted a study including 23 industrial projects for the definition of domain-specific (graphical) modeling languages (DSMLs). As in our approach, a number of DSLs are systematically compared. However, only one kind of DSL development process, the language model-driven process (introduced below), is used and it focuses on the discussion of factors that drive language construct identification. Our approach also covers the integration of DSL concepts with a target platform and addresses the tailoring of the DSL development process.

Similar to patterns, lessons learned have been used as a vehicle to convey best practices of DSL development. For example, Wile [5] reports on 12 lessons learned from three DSL experiments . For each lesson he introduces a respective rule of thumb and gives an overview of the experiences that are the origin of the corresponding rule. Our approach is based on a conceptual model of DSLs that incorporates the concepts identified in the above-mentioned approaches and adds a process-oriented perspective that can be combined with these approaches. However, in general the lessons learned reported by Wile can also be observed in our projects and are hence reflected in parts of our process models.

In summary, the related work on patterns, best practices, and lessons learned in DSL development have in common with our approach that they all derive from experiences and contain some process aspects, but none of them concentrates on describing the DSL development process explicitly. They all focus only on specific, mostly technical issues, whereas our approach provides a more holistic, process-centric view.

2.2. Related work on model-driven and dynamic languages DSLs

In this paper, we especially focus on DSLs in an MDSD context and embedded DSLs for dynamic languages, as our research method is based on the observation of our experiences and most of our own projects were conducted in these fields (see also Sections 3 and 8). A number of approaches in these fields confirm our observations and report on similar development activities as the ones we have observed in our own projects.

For example, supporting model-driven development using several DSLs implemented in Ruby [17] can be achieved by implicitly using a similar process of developing such DSLs as the one we observed in our projects. This has been demonstrated for four embedded DSLs for Ruby that support basic tasks in model-driven development: a DSL to create language models, a DSL to specify restrictions on models, a model-to-model transformation DSL, and a model-to-code language. This study independently confirms our observations for technical DSLs developed in dynamic programming languages.

Other examples that independently confirm our results are based on the idea to build DSLs from component building blocks that can be incrementally designed and composed [9,13,25–27]. This idea originates from approaches such as keyword-based programming [19], in which so-called ‘keywords’ serve as building blocks for DSLs. In particular, a number of (universal) keywords are suggested that are then glued together to compose DSLs. This approach was first envisioned in [28] and is akin to building embedded DSLs in dynamic languages (such as Ruby, Perl, Python, or Tcl, for example). These approaches independently confirm some of the key artifacts we observed in our projects, as well as the process activities related to those artifacts.

DSLs developed in dynamic programming languages are usually build by tailoring an existing general-purpose language (GPL) by adding or changing methods, operators, and default actions [29]. This approach inherently adds activities to the DSL engineering process for either adding methods to the language, or finding proper abstractions in the GPL and changing them in order to best suit the needs of the DSL. Our DSL engineering process can integrate these additional activities. In contrast to the implicit process for building a DSL described in [29], our development process adds a systematic approach for engineering not only the appearance of the DSL in the GPL, but also the models defining the DSL.

Similar to the language-oriented approaches, we also find many independent confirmations of our findings in some model-based or model-driven approaches. For example, similar processes can be used to define DSMLs (UML-based) [10]. In particular, that means to define DSMLs via UML profiles [30], an approach that is commonly used when defining DSMLs (see, e.g. [3]). This approach uses the same key artifacts that we observed when an external modeling DSL is build.

Similar approaches have also been presented for non-standard modeling tools, such as MetaCase’s MetaEdit+tool [1]. In this approach, the steps and considerations for developing DSMLs are pretty similar to the processes observed in our work. However, the processes described in [1] are based on experiences only with the MetaEdit+tool. As our approach is independent of a specific tool or modeling approach, it is more generic. That is, our approach can be tailored to develop DSMLs, and in addition it comprises other approaches for DSL development. However, concerning other facets of DSL development, the approach in [1] also addresses topics that are out of scope in our approach. For example, it also reports on experiences regarding organizational issues, such as running small proof of concept projects to convince executives or on criteria for creating a team that builds a certain DSM solution.

A number of other independent confirmations of our finding can be found in the model-driven literature. In [31], Kurtev *et al.* identify typical problems that may arise when developing a DSL. Moreover, they describe how these problems can be addressed with a model-based DSL framework called AMMA (ATLAS Model Management Architecture, see [32]). AMMA was built on top of the Eclipse Modeling Framework (EMF) [33] and provides tool support for the definition of DSLs. The AMMA core consists of the KM3 meta-model definition language, the TCS language for the definition of concrete syntaxes, and the ATL model transformation language. AMMA does not provide means for defining a DSL’s (behavioral) semantics, but Kurtev *et al.* are experimenting with abstract state machines as a foundation for modeling behavioral DSL semantics with AMMA. Other examples for tools that support DSL definition are the Generic Modeling Environment (GME, see [34,35]) and Microsoft’s DSL tools (see [36]). In [36], Cook *et al.* describe Microsoft’s approach for domain-specific development with Microsoft Visual Studio DSL tools. They suggest an approach for designing a DSL and provide an extensive tutorial for creating DSLs in Visual Studio. In [37], Balasubramanian *et al.* describe two DSMLs that were built using GME. The process aspects of these approaches are mainly implicitly reported, e.g. via examples. These examples confirm that the named approaches follow one of the process tailorings we propose. Our paper additionally provides a systematic approach to describe the DSL engineering process—including different ways to tailor it.

2.3. Related work on grammarware DSLs

Grammarware is defined as a term that comprises grammars (such as context-free grammars or XML schemas) and grammar-dependent software (like parsers, program converters, or XML document processors) [22]. The DSLs we analyzed for this paper are mostly built using dynamic languages or model-driven tools (see also Section 8). Hence, the grammarware approach has not been sufficiently reflected in our experiences, and it might be possible that the process models must be adapted to be used with the grammarware approach. However, as the approaches based on grammarware and model-driven development are quite close (for instance, many model-driven, textual DSL suites, such as XText of openArchitectureWare [38], use grammarware inside), we expect the necessary alterations to be rather limited.

2.4. Approaches dealing with DSL evolution and maintenance

One important aspect of our process models is that they consider DSL evolution and maintenance as an integral part of an incremental development process. In contrast, some other approaches specifically focus on evolution and maintenance as separate aspects. For example, Bierhoff *et al.* [7] investigate the incremental development of a DSL based on existing example applications for a target domain. This means, they chose an application and on top of that they (ex-post) develop a DSL that is expressive enough to describe the application's functionality. This approach represents an innovative version of the 'Extracting DSL from existing system' process variant that we identified in our work (see also Section 6). Van Deursen and Klint describe report on the experiences they gained with Risla [15], a DSL for financial engineering, and especially focus on DSL maintenance issues. In our approach, we have incorporated some of the process aspects of maintenance through feedback loops and iterative development. Wang and Gupta [39] present an approach to rapidly adapt the DSL development environment (consisting of components like compiler, interpreter, or debugger) to changes in the respective DSL. In particular, Wang and Gupta express DSL syntax and semantics using Horn logic. Because the corresponding horn logic expressions are executable, they automatically yield an interpreter for the respective DSL. Based on this DSL interpreter and an actual DSL program, Wang and Gupta then use partial evaluation [40] to automatically generate compiled code. Each time the DSL's syntax and semantics are changed, the corresponding interpreter can be automatically derived from the changed horn logic specification. Our approach deals with DSL evolution in terms of feedback loops and iterative development. Hence, the approach by Wang and Gupta can be seen as a specific concretization of some steps in our development process.

In our approach, in contrast to the above-mentioned approaches, maintenance and evolution of DSLs are not considered as separate process aspects, but rather as an integral aspect of incremental DSL development. Apart from this aspect, all named maintenance and evolution approaches can be integrated with our DSL engineering process.

3. DOMAIN-SPECIFIC LANGUAGES

A DSL is a tailor-made (computer) language for a specific problem domain. In this sense, DSLs differ from GPLs, such as C, C#, Java, Perl, Ruby, or Tcl, that can be applied to arbitrary problem domains. In particular, DSLs are often not Turing complete, and they only provide abstractions suitable for one particular problem domain. However, this specialization results in significant gains in expressiveness and ease of use in the DSL's application domain compared with a GPL (see, e.g. [2,8]): Owing to this expressiveness, a user typically needs to use less language instructions to achieve a certain result in a DSL, compared with achieving the same result in GPLs. A significantly smaller number of instructions lead to programs that are more easy to read and comprehend.

One of the very first papers describing the idea of defining tailored (programming) languages from a set of predefined primitives was presented by Landin [28]. In recent years, DSLs received a constantly growing attention, especially in the area of MDSD, see, e.g. [1,3,18,41,42]. The basic idea of DSLs, however, already has a long history (see, e.g. [43,44]). In the Unix context, for example, DSLs have a tradition as so-called 'little languages' or 'mini languages' (see [43,45]), and in the context of languages such as Common Lisp the development of 'embedded languages' is promoted (see [46]). Other popular examples of DSLs are LaTeX for Typesetting [47,48], HTML for hypertext web pages [49], the (extended) Backus–Naur Form (BNF) for syntax specification [50], or SQL for database queries and manipulation [51,52].

In general, DSLs can be designed and used on different abstraction layers, ranging from DSLs for technical tasks to DSLs for tasks on the business-level. Thus, DSLs can be defined for non-technical stakeholders, such as business analysts or biologists, for example. In general, a DSL makes domain knowledge explicit. That is, the DSL is built so that domain experts can understand and modify DSL code to phrase domain-specific statements that are understood by an information system. An important design principle of DSLs is, therefore, to keep the DSL as simple as possible while making it as powerful as needed.

For the purposes of this paper, we distinguish three main areas for the development of DSLs:

- DSLs in MDSD: here DSLs are often defined as DSMLs that include an infrastructure for the automated transformation of DSL-based models to source code artifacts (see, e.g. [1,3,18,38,41,42]).
- DSLs as extensions of (dynamic) programming languages: here DSLs are domain-specific extensions for GPLs (see, e.g. [4,17,29,53]). Such DSLs are defined as embedded languages that use the language infrastructure of their host language and extend the host language with DSL abstractions.
- DSLs in the grammarware context: here DSLs are based on grammar formalisms and grammar notations and are often specified for the definition of special-purpose document types, for example defined via XML schemas (see, e.g. [22]).

In actual engineering projects, these three areas are not disjunct but may have significant overlaps (for instance, many model-driven, textual DSL suites, such as XText of openArchitectureWare [38], use grammarware inside). As mentioned above, our experiences in developing DSLs result from projects that either applied MDSD techniques, defined on an embedded DSL as an extension for a dynamic programming language, or combined these two approaches (see also Section 8). Therefore, we do not explicitly address the grammarware view on DSLs, even if some of our findings may also be applicable to this field of research as well.

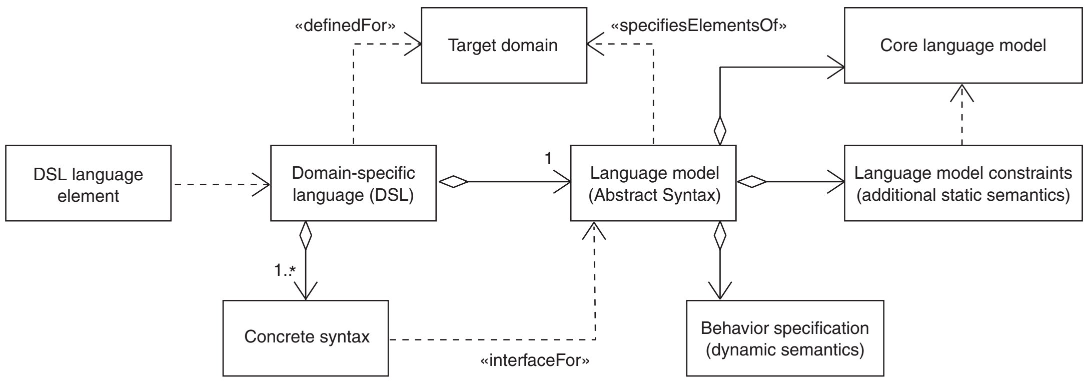

«definedFor»
Target domain
«specifiesElementsOf»
Core language model
DSL language element
Domain-specific language (DSL)
1
Language model (Abstract Syntax)
Language model constraints (additional static semantics)
1..*
Concrete syntax
«interfaceFor»
Behavior specification (dynamic semantics)

Figure 1. Domain-specific languages: artifacts.

Figure 1 depicts the main DSL artifacts and their relations[‡]. From a language user's point of view, the DSL consists of language elements. The definition of these language elements is provided by the language model. The DSL's *language model* specifies elements from the DSL's target domain. Some authors also call the language model the *abstract syntax* of the DSL (see, e.g. [18]). In essence, the language model is a composite model that consists of three sub-models (see Figure 1).

The *core language model* captures all relevant domain abstractions and specifies the relations between these abstractions. In particular, these abstractions refer to elements of the DSL's *target domain*. Examples of domain abstractions are account, bond, client, fund, stock, or stock order in the banking domain; pre-condition, post-condition, test case, or test result in the domain of software testing; role, subject, permission, or constraint in the domain of access control. The DSL's core language model thus formalizes domain-specific knowledge and must be validated by domain experts. The core language model can be defined using any suitable modeling language, such as the UML [30] and UML extensions. Depending on the type of DSL, the core language model can either be a meta-model or an ordinary model.

The *language model constraints*, also referred to as (additional) *static semantics*, are essentially a part of the DSL language model (see Figure 1). They express invariants on elements of the core language model and/or on relations between those elements and thereby define semantics that cannot be expressed directly in the (graphical) core language model. An example from the access control domain would be an invariant that defines that two mutual exclusive roles must never be assigned to the same user (see also Section 7). Usually, constraints are provided in a formal constraint language. If the core language model is defined using the UML, for example, such constraints can be specified using the Object Constraint Language (OCL) [54]. Please note that some language model constraints can also be given as informal text, for instance if it is impossible to express a certain constraint with a particular formal constraint language. The downside of this

[‡]Please note that these artifacts are relevant for any DSL. For many existing DSLs, however, not each of these elements is necessarily realized as a separate artifact. For instance, the DSL's core language model is often not explicitly specified, but only existing in the developers' minds and/or buried in the code. In our approach, we treat the central DSL artifacts as explicitly specified artifacts in the development process.

option is that we then cannot use software tools for an automatic handling and checking of such informal constraints.

The *DSL behavior specification*, sometimes also referred to as *dynamic semantics*, is a part of the language model and defines the (behavioral) effects that result from using a DSL language element. Moreover, it defines how the DSL language elements can interact at runtime. The behavior can be specified in many different ways, ranging from high-level control flow models, over detailed behavioral models, to a precise textual specification.

In addition to the artifacts specifying the abstract syntax of the DSL, each DSL needs a concrete syntax to use the DSL in a certain system environment. A *concrete syntax* represents the abstractions defined through the DSL's abstract syntax, and each DSL can have multiple concrete syntaxes, e.g. a graphical syntax and a textual syntax, or a syntax for human users and a syntax for interacting software components.

The concrete syntax serves as the DSL's interface. Thus, the definition of the concrete syntax(es) is especially important from the DSL user perspective. In case a DSL is primarily built for non-programmer human users, usability properties of this interface, such as being simple and intuitive, are of central importance. In an ideal case, the concrete syntax is therefore, both, convenient for human users and easy to process by software components. If a DSL is primarily built for technical experts and/or machine interaction, however, a more complex concrete syntax may be acceptable or useful.

Transformations are defined to transform DSL code written in a concrete syntax to another model representation or to (programming language) code that can be executed on a specific platform (see Figure 2). In general, a *transformation* is a directive that defines how one (model) format is to be transformed to another (model) format (see, e.g. [55,56]).

A *platform* consists of software building blocks that provide functions to implement the DSL's semantics in a specific system environment. The platform consists of generic platform artifacts, such as programming languages and frameworks (for example based on the Enterprise JavaBeans (EJB) technology or Microsoft .NET), as well as DSL-specific platform artifacts (i.e. parts of the software platform that need to be implemented or integrated into a generic platform just to support the DSL).

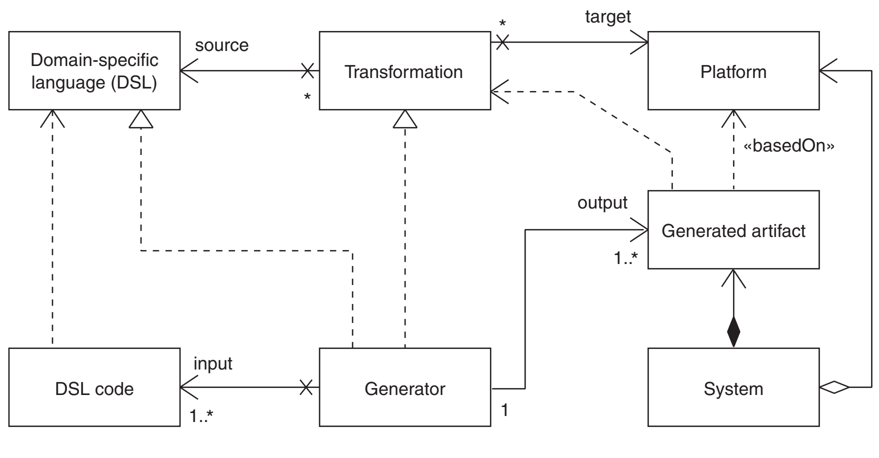


Figure 2. DSL transformations: artifacts (loosely based on [3]).

DSL-to-platform transformations are conducted by a *generator* component (see Figure 2) and result in *generated artifacts* (such as Java, C#, or Ruby code for example) based on the respective platform (see, e.g. [1,3,57]). Note that often repeated transformations to intermediate formats are conducted, before machine code is generated that can actually be executed on a certain processor hardware. However, after transforming DSL code written in a concrete syntax to platform-specific artifacts, the remaining transformations are typically executed by standard tools of the target platform, such as compilers or interpreters (examples are a *C* compiler or a Tcl interpreter).

Two different styles of DSLs can be distinguished with regard to the implementation approach of the DSL (see, e.g. [14,16]):

- An *embedded DSL* (also called *internal DSL*) is defined as an extension to an existing GPL and uses the syntactic elements of the underlying (host) language. In [2], the approach of building an embedded DSL is called 'language exploitation'. An embedded DSL can directly access all features of the host language including libraries, frameworks, or other platform-specific components. Moreover, the tools (e.g. editor, debugger, compiler, or interpreter) available for the host language (or platform) can directly be used when working with the DSL. Thus, it is often not necessary to define transformations or to build an extra generator for an embedded DSL, even though it is possible, of course. In particular, the compiler or interpreter of the host language serves as DSL generator or it is an essential part of the DSL generator. Typical examples of embedded DSLs are the DSLs provided in the Ruby on Rails framework, the Ruby DSLs described in [29], and the Haskel example presented in [11].
- An *external DSL* is defined in a different format than the intended target language(s) and can use all kinds of syntactical elements (independent of any other language). In [2], the approach of building an external DSL is called 'language invention'. External DSLs come with the advantage that DSL designers may define any possible syntax, be it textual or graphical, without considering the syntactical particularities of a given host language. Thus, an external DSL is not bound to a certain host language or platform but can be mapped to different target platforms via transformations. In an external DSL, only the language elements exposed by the concrete syntax are available to the DSL user. Hence, it is impossible to use a feature accidentally that is not part of the DSL (as it could happen in embedded DSLs). Typical examples of textual external DSLs are DSLs developed using openArchitectureWare's XText framework [38] or Microsofts' OSLO framework [58], and many examples of graphical DSLs such as the DSMLs presented in [1].

4. APPROACH OVERVIEW

In the previous section, we have discussed different facets of DSLs and essential DSL concepts. While these facets are more or less well understood, a major issue remains: What are the steps necessary to systematically create a DSL? We conducted many projects where we built different types of DSLs (see Section 8). From these experiences, we can say that the development process of our different DSLs usually consisted of more or less the same realization activities. Hence, our approach is to describe these activities and to identify their interdependencies. In particular, we identified the four main activities summarized in Table I (see also Figure 3). Details on these main activities and the corresponding sub-activities are described in Section 5.

Table I. Overview: main activities in our DSL development approach.

Main activity	Sub-activities	Typical input	Typical output
Defining the DSL's core language model	Identify domain abstraction, add domain abstraction to language model, define language model constraints, check language model	Depends on the actual process tailoring, see Sections 5 and 6	DSL core language model, DSL language model constraints
Defining the behavior of DSL language elements	Select language model element(s), define language model element(s) behavior, check DSL behavior	DSL core language model, DSL language model constraints, DSL concrete syntax	DSL behavior definition
Defining the DSL's concrete syntax(es)	Define symbols for language model elements, define DSL production/composition rules, define DSL concrete syntax	DSL language model	DSL concrete syntax
Integrating DSL artifacts with the platform/infrastructure	Map DSL artifacts to platform features, extend/adapt platform, define DSL-to-platform transformations, test DSL, check integrated DSL	DSL language model, DSL concrete syntax	Transformations, DSL test suite, platform extensions

In our experience, DSL development is an explorative, iterative process in most cases. Unfortunately, for different DSLs developed in different application contexts, the order in which these activities need to be performed, and the exact steps that must be executed to perform the activities, can change significantly (see Section 6). Influencing factors are, for example, the size or impact of the DSL, the size of the corresponding development project, the involvement of the various DSL stakeholders (especially the domain experts), and the type of development process that is used in the DSL project (see Section 8 for examples on how these factors have influenced the DSLs we have built in our projects). As DSLs are usually developed in the context of larger projects or existing software products, often these influencing factors are predetermined and cannot easily be changed or adapted for the DSL development. However, in any case a systematic process, including all required activities, is useful to guide DSL developers. For these reasons, our general approach to DSL development consists of two major tasks:

1. Tailor the DSL engineering process to the project's context.
2. Apply the tailored DSL engineering process to develop the DSL.

It often makes sense to further refine a tailored DSL engineering process during its application phase in order to improve the process and/or adapt it to the environment in which it is performed (see Section 6). Some prototypical examples of tailored DSL engineering processes that we identified in our projects are sketched below:

- *Language model driven*: In this type of DSL engineering process, the language model definition drives the DSL development. That is, first the core language model is defined to reflect all

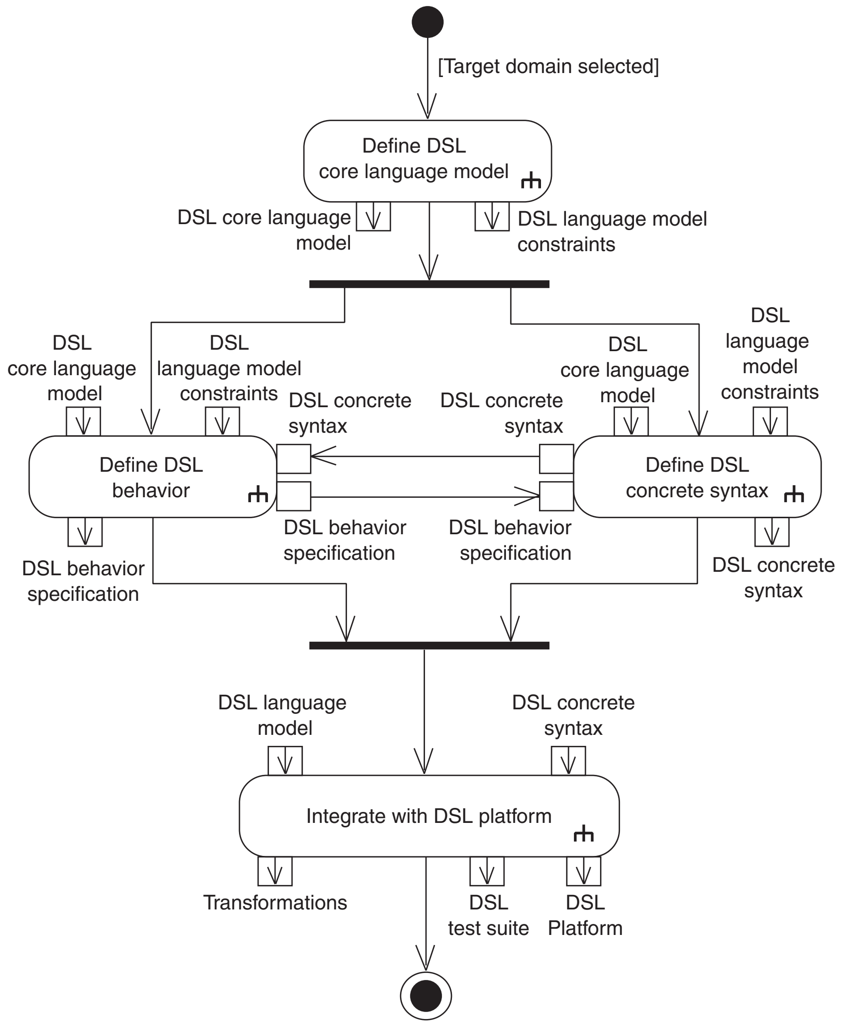

[Target domain selected]
Define DSL core language model
DSL core language model
DSL language model constraints
DSL core language model
DSL language model constraints
DSL concrete syntax
DSL concrete syntax
DSL core language model
DSL language model constraints
Define DSL behavior
Define DSL concrete syntax
DSL behavior specification
DSL behavior specification
DSL behavior specification
DSL concrete syntax
DSL language model
DSL concrete syntax
Integrate with DSL platform
Transformations
DSL test suite
DSL Platform

Figure 3. Language model-driven DSL engineering process (main control flow).

relevant domain abstractions, then the concrete syntax is defined along with the DSL's behavior, and finally the DSL is mapped to the platform/infrastructure on which the DSL runs. Like all other process variants, the language model-driven process is working incrementally. This means, incremental refinements, iterations, and feedback loops are applied whenever necessary, for instance to correct mistakes or to refine the results of prior activities. The language model-driven process variant can be further tailored by specifying how large the increment defined in an iteration is—ranging from larger language model chunks that are defined in one batch to an own iteration of the process for every single language model element.

- *Mockup language*: To raise the participation of domain experts, it is also possible to start DSL development with the concrete syntax design and then distill the abstract syntax and semantics from this concrete syntax. In particular, the concrete syntax is developed together with domain experts by utilizing the domain experts' domain knowledge. Next, a first prototype implementation of the resulting mockup language is developed. The prototype then serves as a means to perform acceptance testing activities, and to further refine and tailor the concrete syntax to the domain experts' needs.

- *Extract DSL from existing system*: Sometimes an existing software system should (ex post) be provided with a DSL. That is, the domain abstractions for the DSL can be derived directly from the existing system. In this case, it makes sense to first elicit the language model elements from the abstractions given in that system. If the architecture of the corresponding software system is documented (e.g. using a graphical modeling language such as the UML), this architecture description can be a valuable source for the elicitation of domain abstractions.

In our experience, the language model-driven process is well suited for explanatory purposes—it proceeds in a top–down fashion, which is easy to understand and follow in a documentation. Thus, even if the DSL is defined using some other tailored process variant, it makes sense to document or explain the DSL with the language model-driven process. In our experience, the language model-driven process is also well suited to be applied to small projects, where the DSL developers are domain experts themselves and in which the DSL is developed from scratch (for example, when developing technical DSLs for software developers).

In Section 5, we will use the language model-driven process variant to explain the different activities of DSL development. In addition, we describe interdependencies of these activities in terms of inputs required and outputs provided by the activities.

Our approach mainly aims at the engineering activities for developing a DSL—in particular, we describe the recurring development activities that we identified in our DSL projects. Nevertheless, other development activities must also be performed, such as defining/selecting the software development tools and techniques, or project management activities to involve and coordinate different stakeholders. However, in our experiences such aspects are often determined through the general purpose software development process that is applied in a particular project, such as the UP [20] or agile processes such as XP [21]. Therefore, such general-purpose development activities are not in focus of our engineering approach. That is, our activities for DSL engineering do not define a complete software development process, but rather a micro-process that can be plugged into these general-purpose processes. The (considerable) differences between general-purpose software development processes are an important reason why our micro-process must be tailored before it can be applied in a concrete project (see Section 6).

5. PROCESS ACTIVITIES FOR DSL ENGINEERING—ILLUSTRATED USING A LANGUAGE MODEL-DRIVEN PROCESS

5.1. Main activity flow

Figure 3 shows an example tailoring of the DSL engineering process. In particular, the process in Figure 3 shows the main control and object flow of the 'Language Model-Driven Process' variant (see Section 4). Note that it is of course possible to move back in the process at any time, for example, if an error or an incompleteness in one of the models is detected. For the sake of simplicity, however, we chose to omit the corresponding reverse links in Figure 3 and focus on the main flows. The purpose of this and the following figures in this section is two-fold:

- First, the figures specify each action (or sub-activity) in terms of the inputs required and the outputs provided by this particular action (or sub-activity).
- Second, the figures show one possible tailoring of the process as a control and object flow. The purpose is to show to the reader one possible way of how to combine the DSL engineering activities and traverse through a DSL process. In particular, we use the 'Language Model-Driven Process' for explanatory purposes. Other tailored processes can be defined by combining the process activities using a different control and object flow (see Sections 4 and 6).

The 'Language Model-Driven Process' is initiated by selecting the target domain of the DSL. The first activity is to define an (initial) core language model and corresponding language model constraints for this target domain. Subsequently, the process continues with two parallel activities: the definition of the DSL's behavior and the definition of at least one concrete syntax (see Figure 3). We suggest to conduct these activities in parallel because the behavior and the concrete syntax of a DSL may mutually affect each other. Performing the two activities in parallel is especially useful for embedded DSLs (see Section 3) because syntax and behavior of the host language have a considerable influence on the concrete syntax and behavior of the DSL. When realizing the DSL as an external DSL, these two activities may also be conducted in succession (i.e. the DSL development process can be tailored in this way). However, even if the DSL is build as an external DSL, conducting both activities in parallel is often useful because intermediate results can be exchanged to produce a concrete syntax and corresponding behavior definitions that are consistent, complete, and well integrated with each other.

After defining the DSL's behavior and concrete syntax, the fourth activity in the engineering process maps the different artifacts to the target platform (see Figure 3). This activity produces transformations, corresponding integration tests, and corresponding platform extensions for the DSL.

Note that an important facet of the engineering process is testing and each of the main activities includes sub-activities to check the respective DSL artifacts on different levels of abstraction. For example, whenever a running piece of code is produced, it should be unit tested. In general, there are many kinds of code artifacts that could be produced during a DSL engineering process, such as a language model element with an API, a concrete syntax element, a grammar, a transformation template, or a method for producing a part of an intermediary language. In addition to unit testing, DSL integration testing checks if different DSL artifacts interact as specified. In Section 5.5, we introduce integration tests as part of the DSL-to-platform integration activity. This is because integration tests can only be run successfully if all parts of the platform that are related to the respective integration test are available. Integration tests are especially derived from the DSL's behavior specification (see Section 5.3).

In the following sections, we describe each of the four main activities in detail.

5.2. Subprocess: define DSL core language model

In this section, we explain the refinement of the main activity 'Define DSL Core Language Model' (see Figure 3). The subprocess starts with the identification of domain abstractions (see Figure 4). For example, if we are engineering a DSL for the access control domain, corresponding domain

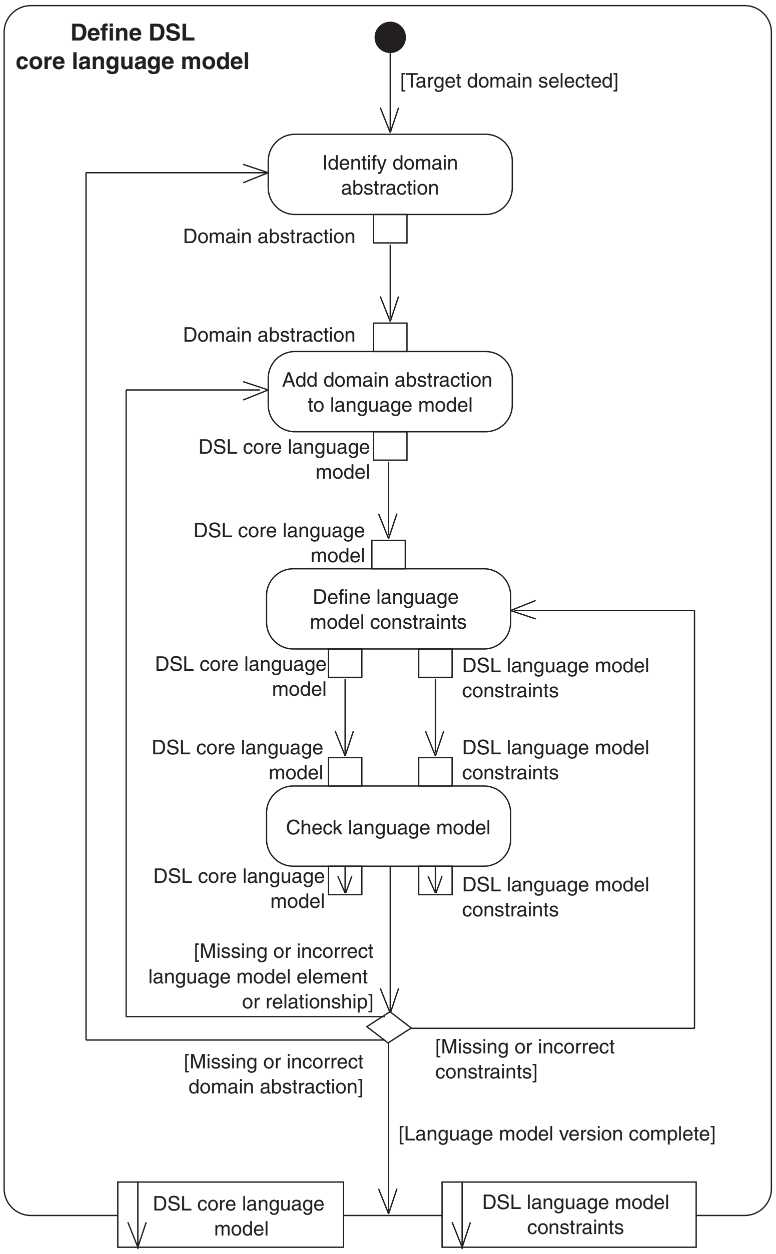

Define DSL core language model
[Target domain selected]
Identify domain abstraction
Domain abstraction
Domain abstraction
Add domain abstraction to language model
DSL core language model
DSL core language model
Define language model constraints
DSL core language model
DSL language model constraints
DSL core language model
DSL language model constraints
Check language model
DSL core language model
DSL language model constraints
[Missing or incorrect language model element or relationship]
[Missing or incorrect constraints]
[Missing or incorrect domain abstraction]
[Language model version complete]
DSL core language model
DSL language model constraints

Figure 4. Subprocess: define DSL core language model.

abstractions could be: subject, role, or permission; examples for domain abstractions in the banking domain are account, bond, stock, and stock order. For complex domains, the identification of domain abstractions is usually non-trivial. Hence, we generally recommend to follow a domain analysis method, such as domain-driven design for the identification of domain abstractions (see, e.g. [59–62]).

Subsequently, the domain abstractions are integrated with each other to build a version of the DSL core language model. Most often language models are defined using a modeling language, such as UML [30], or a modeling framework, such as EMF [33]. In the most simple case, adding a domain abstraction means to add an additional entity (e.g. adding a class to a class model) for the respective domain abstraction and then connect it to existing entities using relationships, such as associations or generalizations. However, adding domain abstractions can also mean to refactor the core language model. For instance, a new abstraction can also be added by modifying existing entities or relationships.

Next, it is checked whether we need to define additional language model constraints or modify existing ones. Subsequent to defining the core language model and its constraints, the language model is checked for completeness and correctness from a domain-oriented perspective (not in a formal computer science sense). As a thorough core language model is essential for engineering a DSL, the software engineers must check the core language model in collaboration with domain experts. If these checks reveal an incorrectness or a missing domain abstraction, the software engineers move back to the corresponding activity to adapt the model accordingly. This cycle is repeated until the language model is accepted by the software engineers and the domain experts (see Figure 4). We generally advise to follow the best practices described in [60] to stepwise bring the domain understanding and the technical understanding together.

5.3. Subprocess: define DSL behavior

Even though the language elements of a DSL are defined for a narrow target domain and often appear to be 'intuitively clear' to DSL designers and users, intuition may be misleading. In our experience, one of the most common sources of errors is that the software engineers believe that they understand the application domain, while this is not yet the case. Therefore, it is most often useful to explicitly define the behavior of a DSL.

The behavior definition of a DSL determines how the language elements of the DSL interact to produce the behavior intended by the DSL designers. This is not only important to the users (domain experts) of the DSL but also to the software engineers. In particular, explicitly specified behavior allows for a correct mapping of the (platform-independent) DSL specifications to a certain software platform that must behave exactly as specified.

The subprocess for the definition of DSL behavior (see Figure 5) starts with selecting the core language model elements for which we like to specify its behavior. In our experience, the selection is typically either a single element of the core language model or a number of related core language model elements. Only in rare cases the whole core language model is selected at once. Next, the behavior of the selected elements is defined. In general, the behavior can be specified in many different ways, ranging from high-level control flow models, over detailed behavioral models that are used for model-driven generation of the corresponding code, to a precise (executable) textual specification. Subsequently, the software engineers and domain experts check the behavior specification for domain correctness and for software-technical correctness. If an error is detected

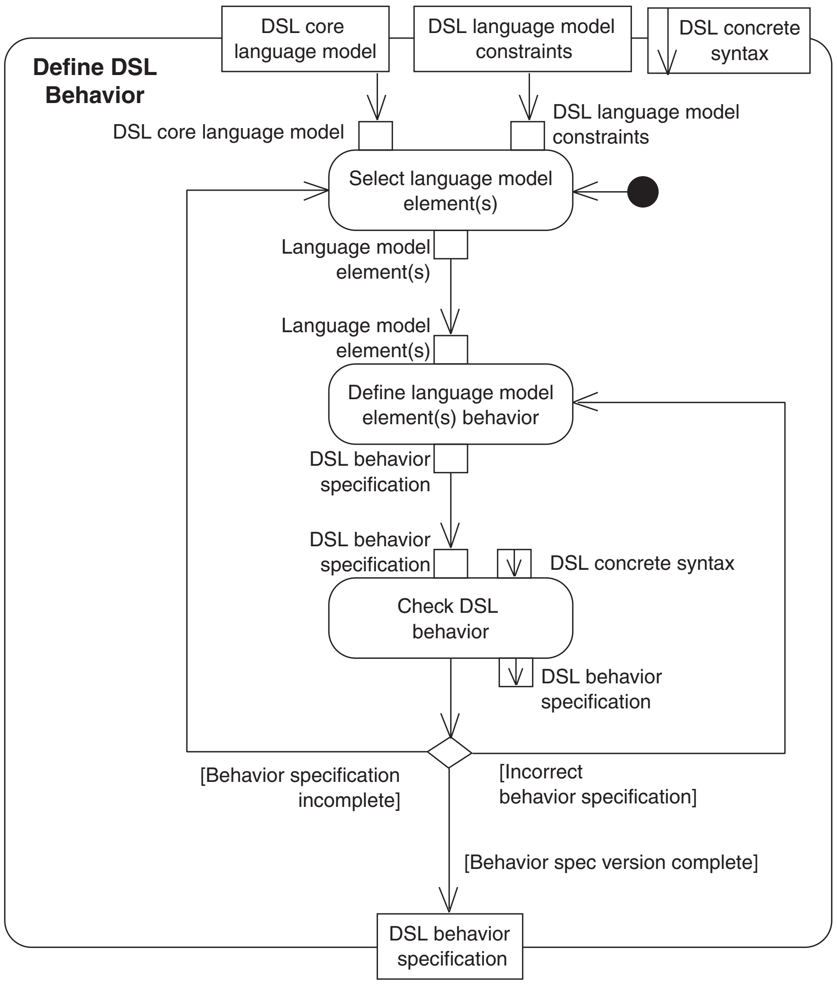

DSL core language model
DSL language model constraints
DSL concrete syntax
Define DSL Behavior
DSL core language model
DSL language model constraints
Select language model element(s)
Language model element(s)
Language model element(s)
Define language model element(s) behavior
DSL behavior specification
DSL behavior specification
DSL concrete syntax
Check DSL behavior
DSL behavior specification
[Behavior specification incomplete]
[Incorrect behavior specification]
[Behavior spec version complete]
DSL behavior specification

Figure 5. Subprocess: define DSL behavior.

or if the behavior definition is incomplete, the software engineers move to the respective activity to adapt the definitions accordingly.

5.4. Subprocess: define DSL concrete syntax

The definition of the concrete syntax(es) of a DSL is of central importance for DSL users because it represents the ‘user interface’ of the language. Hence, it should be comprehensible and convenient to use for domain experts—even if actual DSL code is often written by software engineers (see also Section 3).

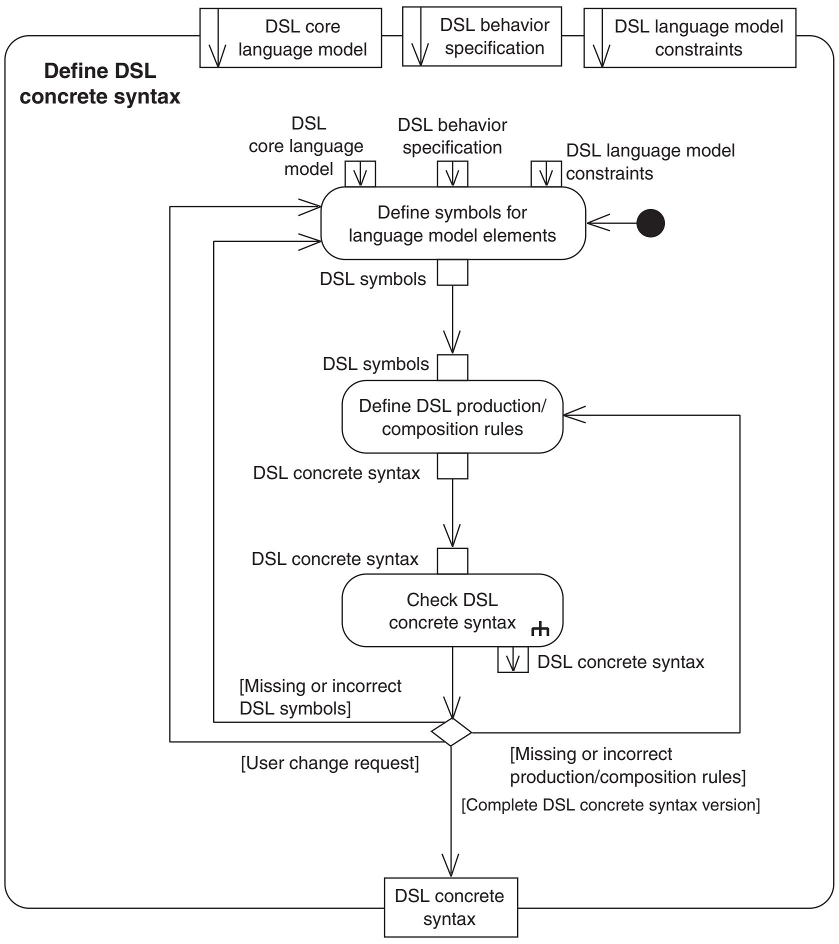

DSL core language model
DSL behavior specification
DSL language model constraints
Define DSL concrete syntax
DSL core language model
DSL behavior specification
DSL language model constraints
Define symbols for language model elements
DSL symbols
DSL symbols
Define DSL production/ composition rules
DSL concrete syntax
DSL concrete syntax
Check DSL concrete syntax
DSL concrete syntax
[Missing or incorrect DSL symbols]
[User change request]
[Missing or incorrect production/composition rules]
[Complete DSL concrete syntax version]
DSL concrete syntax

Figure 6. Subprocess: define DSL concrete syntax.

As discussed in Section 3, the concrete syntax of a DSL can be either graphical or textual. In addition, we further distinguish embedded DSLs and external DSLs. Moreover, some DSLs have only one concrete syntax, while others have more than one (e.g. a graphical and a textual syntax). Hence, the realization techniques for implementing concrete DSL syntaxes include implementing a GUI editor, implementing a grammar and a parser, or extending an interpreter, for example. Even though these realization alternatives are highly different on a technical level, we can find similar engineering activities that are necessary to define the DSL's concrete syntax.

The subprocess for defining the concrete DSL syntax (see Figure 6) starts by defining symbols for each element of the DSL. We use the generic term 'symbol' to refer to arbitrary concrete representations of DSL elements, for example graphical symbols or textual symbols of a grammar. With regard to these symbols, DSLs should, on the one hand, try to use standard programming language conventions where possible, for example to define how comments, strings, and numbers

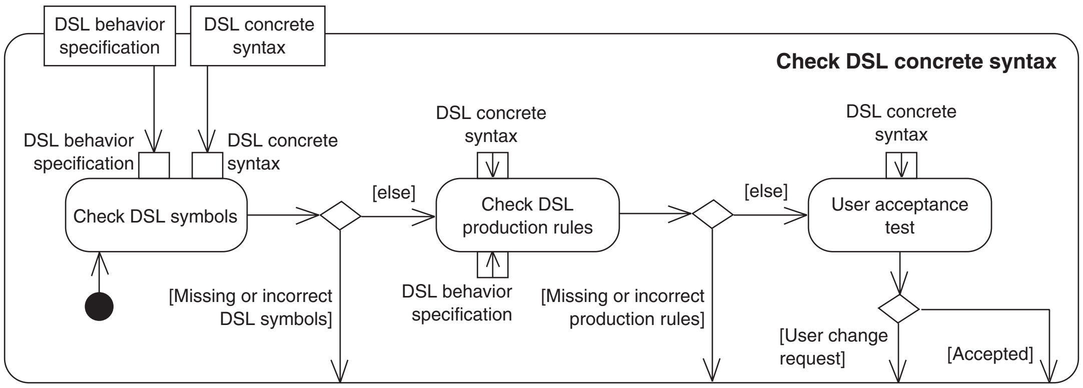

DSL behavior specification
DSL concrete syntax
Check DSL concrete syntax
DSL behavior specification
DSL concrete syntax
Check DSL symbols
[else]
[Missing or incorrect DSL symbols]
DSL concrete syntax
Check DSL production rules
DSL behavior specification
[else]
[Missing or incorrect production rules]
DSL concrete syntax
User acceptance test
[User change request]
[Accepted]

Figure 7. Checking concrete syntax artifacts.

are specified. On the other hand, symbols and terms familiar to the different DSL stakeholders should be used.

Next, the software engineers must define the composition rules or production rules of the syntax. These rules define how the symbols can be composed to phrase legal DSL expressions. In principle, all languages have a grammar and hence production rules. However, we also use the more general term 'composition rules' here, because in some cases the grammar is not formally specified in terms of formal production rules, but rather the composition rules are just implicitly given, for instance via the implementation of a graphical tool.

The next activity then checks the concrete syntax for correctness and completeness from the point of view of the domain experts (i.e. correctness and completeness are not meant in a 'formal' sense here). This is done by interviewing and discussing the domain experts, as well as using observations and studies on their work with the DSL prototype. Moreover, domain experts check if the concrete syntax is convenient to understand and use. Figure 7 shows this activity in detail. If the concrete syntax is found to be incomplete or incorrect, the technical experts move back to the corresponding preceding activity.

5.5. Subprocess: integrate with DSL platform

The target platform of the DSL consists of two main parts: generic platform artifacts, such as programming languages and frameworks, and DSL-specific platform artifacts. Examples for generic platform parts could be EJB or Microsoft's .NET infrastructure. The DSL artifacts, such as core language model, language model constraints, behavior definition, and concrete syntax must be *mapped* to the corresponding platform (see Section 3).

The first activity in the subprocess is to map the DSL artifacts to the features of the platform (see Figure 8). That is, we first have to decide which existing platform parts can be used to realize the respective DSL artifact. Often the platform needs to be extended or adapted because feature support is missing or incomplete. This is done as a next step, if necessary—the corresponding subprocess is shown in Figure 9. In case platform extensions must be defined, we also have to specify test cases

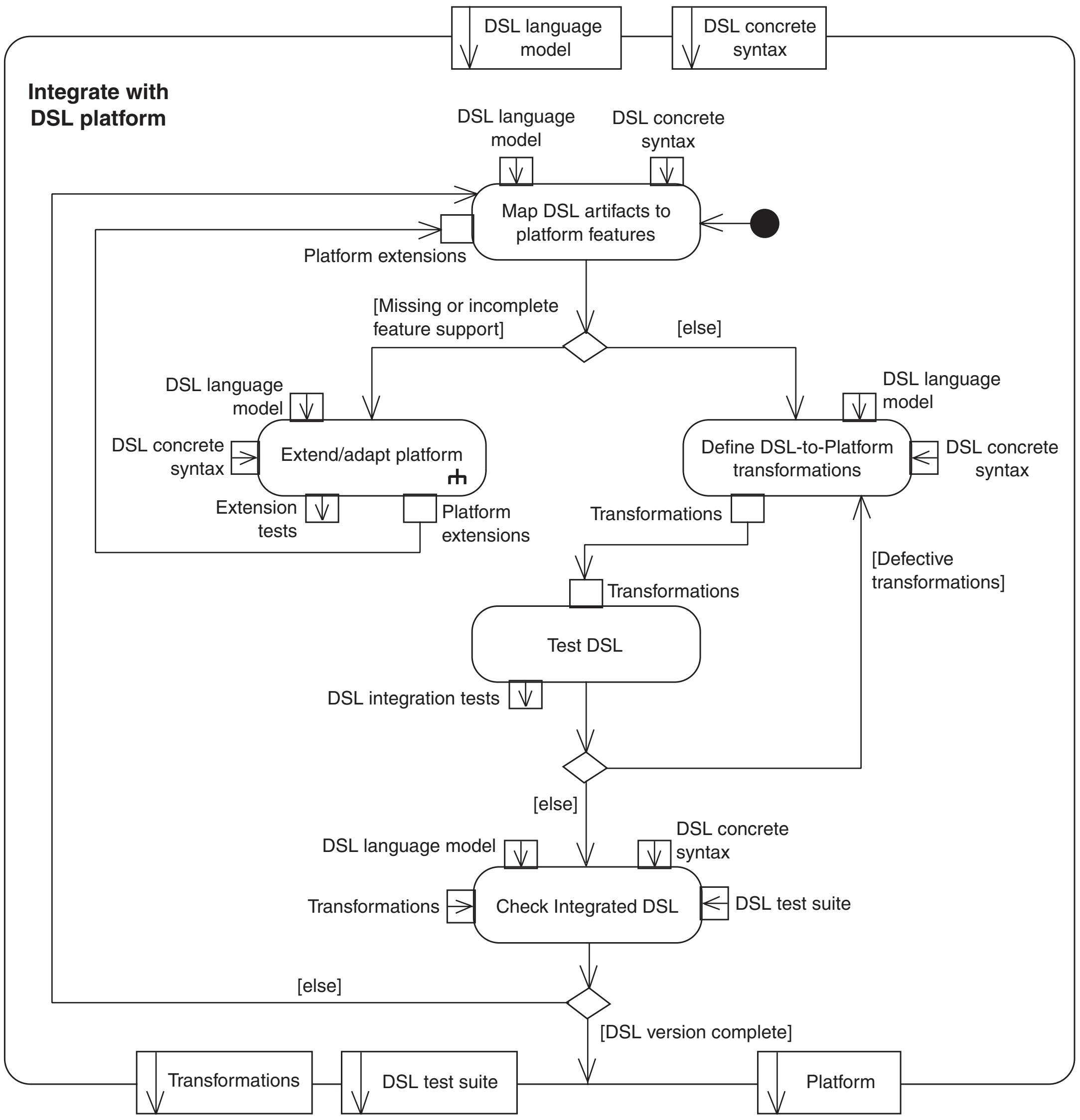


Figure 8. Subprocess: integrate with DSL platform.

for these platform extensions (see Figure 9). When testing of the platform extensions is completed, the platform extensions (and corresponding regression tests) are ready to use.

Now the software engineers check again if the target platform provides all features needed by the DSL. If the platform is still missing features, additional platform extensions are defined. In many cases it is best to define the platform's interface (and the corresponding extensions) through a clear API. In some cases, a low-level DSL can be used to specify the mapping between the API and the DSL under development.

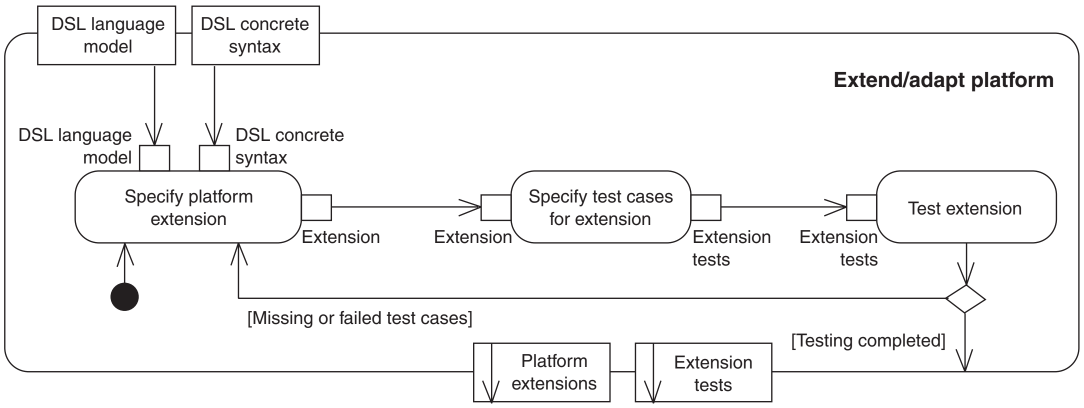


Figure 9. Subprocess: extend/adapt platform.

After all features needed for the platform are identified and implemented, DSL-to-platform transformations are defined. In case the language is defined as an embedded DSL, it is likely that this activity is very simple (see also Section 3). For external DSLs, the transformations can, for instance, be realized by defining transformation templates or transformation rules. Of course, both approaches can also be combined (for instance, transformation templates can also be used to generate code for an embedded DSL).

For external DSLs, the software engineers define transformations that convert models, which conform to the DSL's language model, to the platform (see also Section 3). These transformations should not directly rely on the concrete syntax. That is, if more than one concrete syntax is defined, the same (generic) transformations should work for all concrete syntaxes. For instance, the generator openArchitectureWare [38] requires all concrete syntaxes to be converted to EMF models, and its transformation language XPand allows developers to specify transformation templates that convert any legal EMF model to the target platform.

After defining the transformations, we specify corresponding transformation tests to assure that the generated artifacts resulting from the transformations actually behave as specified. Moreover, we define integration tests for the DSL to check that all DSL artifacts work together properly. The complete test suite of the DSL then includes various unit tests defined throughout the process, the integration tests defined in this activity, and (possibly) tests for platform extensions. In addition, the test suite may include the criteria for user acceptance tests of the concrete syntax (see Section 5.4).

Eventually, all DSL artifacts are subject to a final check conducted by technical experts and domain experts. This final check includes the repeated verification and validation of all DSL artifacts. If the language is found to be complete, the engineering process is finished and a version of the DSL is ready for use (see Figures 3 and 8).

6. TAILORING THE DSL DEVELOPMENT PROCESS

The micro-process for DSL development is generic in the sense that it can be used in the context of different types of general-purpose development approaches, such as the UP or agile development

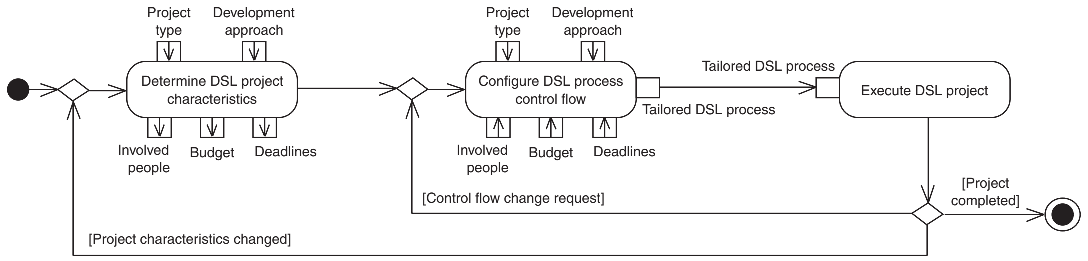


Figure 10. Project-specific tailoring of the DSL engineering process.

approaches. However, to apply the DSL engineering process in a particular project, it has to be tailored to fit into the corresponding organization's standard development approach or the specific approach of its project, and to meet the needs of the engineers conducting the DSL engineering process.

Figure 10 shows the activities that need to be conducted for a project-specific tailoring of the DSL engineering process. In general, we first determine the characteristics of a particular DSL project. Next, we define a tailored control flow for this particular project, before we can execute the tailored engineering process. It is also possible to dynamically adapt a tailored DSL engineering process, for example if the characteristics of a project change. The characteristics of a particular DSL project are determined by the project type, the development approach, the people (or roles) involved in the project, the (monetary) budget, and the deadlines that must be met. In the second tailoring activity, these factors then define the surrounding conditions of a particular DSL engineering project and influence the control flow of the corresponding tailored process.

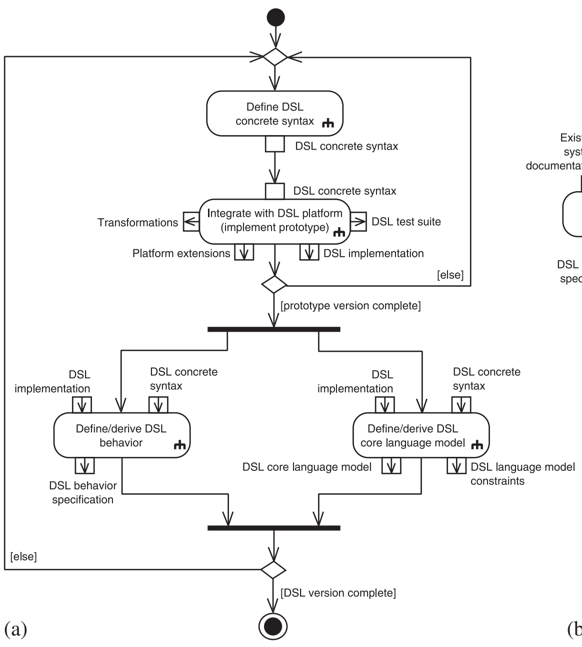
Figure 11 shows two examples of tailored DSL engineering processes. In particular, Figure 11(a) shows a process variant that starts with the engineering of a mockup language, and Figure 11(b) shows a process for the extraction of a DSL from an existing system (see also Section 4). Similar to the process variant shown in Figure 3, the tailored processes in Figure 11 show the main control and object flow of the corresponding variants. In addition to the links depicted in Figures 11(a) and (b) it is of course possible to move back in the process at any time. However, for the sake of simplicity we focus on the main flows.

In the following two sections, we provide more details on the two process tailoring examples.

6.1. Mockup language-driven DSL development

In case a domain is very complex and cannot be easily understood by the software engineers (e.g. air traffic control, definition of financial products, or power-plant control), DSL engineering requires the domain experts to be highly available for the development team. In such a project, it is sensible to raise the participation of the domain experts through the initial definition of a *mockup language*.

In particular, we start DSL development with the concrete syntax design and then distill the abstract syntax and semantics from this concrete syntax, as shown in Figure 11(a). This means, the concrete syntax is developed together with a domain expert by utilizing the domain expert's domain knowledge. Next, a first prototype implementation of the resulting mockup language is developed. The prototype then serves as a means to perform acceptance testing activities, to further refine and



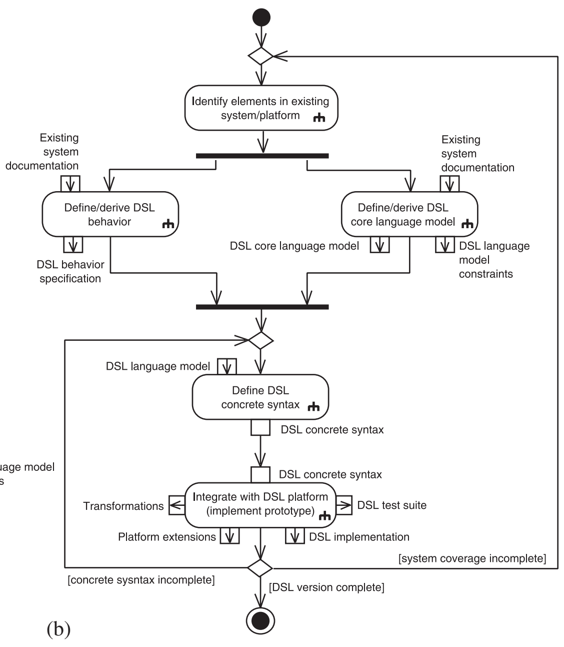

Identify elements in existing system/platform
Existing system documentation
Existing system documentation
Define/derive DSL behavior
Define/derive DSL core language model
DSL behavior specification
DSL core language model
DSL language model constraints
DSL language model
Define DSL concrete syntax
DSL concrete syntax
DSL concrete syntax
Transformations
Integrate with DSL platform (implement prototype)
DSL test suite
Platform extensions
DSL implementation
[system coverage incomplete]
[concrete sysntax incomplete]
[DSL version complete]
(b)

Figure 11. Examples of tailored process variants for DSL engineering.

tailor the concrete syntax to the domain expert's needs. This is done in multiple iterations to get closer to the domain abstractions in a stepwise manner.

Please note that this approach is not driven by minor syntactical considerations, such as which types of parentheses to use. Instead, the focus of the mockup language design is to identify the domain abstractions and terminology. Hence, it makes sense to combine the mockup language-driven process with a domain engineering approach, such as domain-driven design [60]. In domain-driven design, the abstraction level targeted by the mockup language-driven process is called the ubiquitous language of the domain.

After a version of the prototype is completed, the corresponding DSL core language model and DSL behavior definition that result from this version of the prototype should be documented. Subsequently, the process continues with the next iteration to further enhance the draft version of the DSL. This co-evolution of the different DSL artifacts continues until the domain experts and software engineers define the DSL as complete (see Figure 11(a)).

This process tailoring puts emphasis on checking the language model together with the domain experts. In particular, the subprocess for definition of the concrete syntax is guiding the whole DSL engineering process. This subprocess can either be performed iteratively or a syntax can be completely defined for a version of the language model. For instance, in case a second concrete syntax is added to a DSL, it often makes sense to specify the whole syntax at once. In our experience, a more iterative approach makes sense in case the language is used for incrementally illustrating and sketching domain concepts while the DSL is developed. This, of course, is dependent on the availability of the domain experts. Mockup language-driven DSL development is well suited for projects that follow an agile development approach [63,64], which advocate on-site domain experts (customers).

6.2. Extracting a DSL from an existing system

Sometimes the goal of an DSL development project is to define a DSL as an (additional) user interface for an existing software system. Therefore, a tailored engineering process for such a project adds an 'Identify elements in existing system/platform' analysis activity that identifies one or more elements of the system/platform (see Figure 11(b)). Examples for such elements are system components or interfaces that can be mapped to language model elements. That is, the domain abstractions for the DSL can directly be derived from the existing system. In seldom cases (e.g. for small systems or very well-understood domains), it makes sense to map the whole existing system/platform to the DSL at once. However, in most real-world cases this type of engineering process is also iterative (see Figure 11(b)).

If the architecture of the corresponding software system is documented (e.g. using a graphical modeling language such as the UML), this architecture description can be a valuable source for the elicitation of domain abstractions and DSL behavior. Here, domain experts either take part in the elicitation process or they review the resulting language model. The definition of the DSL's behavior and core language model can be further tailored, for example by how many DSL elements are specified in an iteration of the process or by defining how often the domain expert gets involved in the subprocess. This depends on the availability of the domain expert and how iterative the development process is. In less iterative processes, we usually specify the behavior of multiple elements at once. After defining the DSL core language model and behavior, the process proceeds with the definition of a concrete syntax and the integration with the platform. Integration with the platform here primarily means the definition of transformations from models (that conform to the language model) to code that is executable on the target platform.

7. AN ILLUSTRATIVE EXAMPLE: ENGINEERING A DSL FOR RBAC

In this section, we describe an example application of our DSL engineering process for the development of a RBAC DSL. In particular, we applied the language model-driven variant of our process (see Section 5) to develop this RBAC DSL. Below we present a selected, yet representative, subset of the RBAC DSL specification to demonstrate the engineering tasks described in Section 5. A complete definition of the RBAC DSL, including all formal models and specification details, would be beyond the scope of this paper.

7.1. Defining the DSL's core language model

We have chosen to use the language model-driven variant of our process because the RBAC DSL is placed in a domain well known to the developers (see, e.g. [4,65–69]). Therefore, it was sensible for us to start with a conceptual language model that can be incrementally refined as the DSL evolves.

Figure 12 depicts our core language model of the RBAC domain as an UML2 class model. In the RBAC context (see [70,71]), an access control *subject* is an active entity (e.g. a human user or a software agent) that is (or should be) able to access objects (e.g. a file like an XML document,

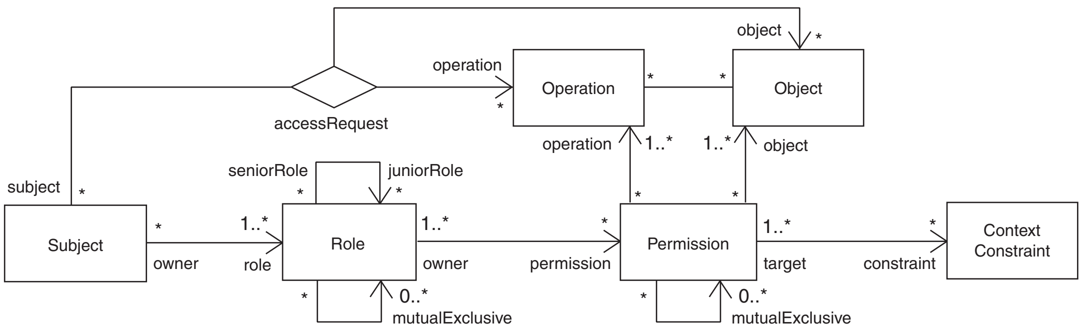


Figure 12. Core language model of the target domain: role-based access control.

or a hardware resource such as a printer) in a particular information system. Each subject receives its permissions through the roles assigned to this particular subject. A *role* is an abstract entity that represents a certain type of subject in terms of the permissions granted to this type of subject. When modeled to represent human users, roles most often reflect the work profiles of a certain organization (see [66,68]). In its basic form, a *permission* consists of an $\langle$operation, object$\rangle$ tuple. It represents the right to perform the respective operation on the corresponding object. Moreover, permissions can be associated with *context constraints*. Context constraints define predicates that must evaluate to true in order to grant a certain access request. They allow for the consideration of context information in access decisions and enable the definition of additional conditions on permissions, such as time constraints (for details see [69]).

An access request is defined through a ternary association and consists of a $\langle$subject, operation, object$\rangle$ triple (see Figure 12). Access requests are granted if the respective subject owns a corresponding permission.

In RBAC, the relations between subjects and roles as well as between roles and permissions are called *assignment* relations (see [70,71]). As both assignment relations are many-to-many relations, each role can be assigned to different subjects and each permission can be assigned to different roles.

The core language model includes two recursive associations on Role. The first one defines that a role may have multiple junior and/or senior-roles. In RBAC, these junior and senior-relations are used to build role hierarchies. In a *role hierarchy*, senior-roles are more powerful than junior-roles, which means that a senior-role inherits the permissions assigned to its junior-roles.

The second recursive association on Role defines that each role may be mutual exclusive to one or more other roles (see Figure 12). Mutual exclusion is used to enforce the ‘separation of duty’ concept (see, e.g. [67,70,72]), which directly affects role-to-subject and/or permission-to-role assignment. In access control, separation of duty constraints enforce conflict of interest policies. Conflict of interest arises as a result of the simultaneous assignment of two mutual exclusive roles (or permissions) to the same subject. Mutual exclusive roles (or permissions) result from the division of powerful rights or responsibilities to prevent fraud and abuse. An example is the common practice to separate the ‘controller’ role and the ‘chief buyer’ role in the medium-sized and large companies, or the separation of the ‘trader’ and ‘internal revision’ roles in investment banking.

In addition to the language model elements defined in Figure 12, there are a number of domain-specific constraints on core language model elements. Here, we use the OCL [54] for defining these constraints. Of course, one may also use other modeling languages than UML and OCL to define a DSL language model and corresponding constraints (see also Section 3).

The following OCL constraint defines that a subject can never own two mutual exclusive roles:

```
context Subject
inv: self.role->forAll(r1,r2 |
   r1.mutualExclusive->select(mer|mer.name=r2.name)->isEmpty()
   and
   r2.mutualExclusive->select(mer|mer.name=r1.name)->isEmpty() )
```

Furthermore, similar to permissions, separation of duty constraints is subject to inheritance (see, e.g. [67]). Thus, we define the OCL constraints below to express that no role can be mutual exclusive to itself nor to one of its junior or senior roles:

```
context Role
inv: self.mutualExclusive->select(mer|mer.name=self.name)->isEmpty()
inv: self.juniorRole->forAll(r|
    self.mutualExclusive->select(mer|mer.name=r.name)->isEmpty() )
inv: self.seniorRole->forAll(r|
    self.mutualExclusive->select(mer|mer.name=r.name)->isEmpty() )
```

Analogous to roles, mutual exclusion can be applied to individual permissions (see Figure 12). This means, that two mutual exclusive permissions may never be assigned to the same role and that no junior-role is allowed to possess a permission that is mutual exclusive to the permissions assigned to one of its senior-roles. This is defined by the following OCL constraints:

```
context Role
inv: self.permission->forAll(p1,p2|
   p1.mutualExclusive->select(mep|mep.name=p2.name)->isEmpty()
   and
   p2.mutualExclusive->select(mep|mep.name=p1.name)->isEmpty() )
inv: self.juniorRole->forAll(r|
   self.permission->forAll(p1|
    r.permission->forAll(p2|
     p1.mutualExclusive->select(mep|mep.name=p2.name)->isEmpty() )))
```

Finally, for the sake of completeness, we define that no permission can be mutual exclusive to itself:

```
context Permission
inv: self.mutualExclusive->select(mep|mep.name=self.name)->isEmpty()
```

The elements of the core language model (i.e. the classes and associations) determine the different language elements of our DSL. In addition, the core language model already determines and/or constrains the behavior of DSL elements to a certain degree. For example, the OCL constraints defined above specify invariants that must never be violated by an implementation (respectively the target platform) of the DSL. However, in addition to such invariants, a number of behavioral choices exist, which are not defined through the DSL's core language model. Therefore, we define the DSL's behavior as a next step.

7.2. Defining DSL behavior

Because of our extensive experiences in the RBAC domain, we were able to completely define the behavior for each feature of the DSL before it got implemented (respectively mapped to the target platform). When, during implementation or other process stages, we observed issues with the DSL behavior specifications, we incrementally refined them.

For the RBAC DSL, we especially used extended UML activity diagrams (see [73]) to model DSL behavior and the interaction of different concerns in the DSL. In a subsequent step, these activity models are further refined through UML interaction diagrams. The activity models then specify the high-level control flows that are independent of the implementation on concrete platforms, while the interaction models describe the platform-specific details (see also Section 7.4).

Figure 13 depicts an activity diagram that shows the primary control flow for authorization decisions. Furthermore, it shows the interdependencies of the Authorization concern, the Role concern, the Permission concern, and the ContextConstraint concern.

The `checkAccess` language element is derived from the 'accessRequest' association shown in Figure 12. The `checkAccess` element probably provides the most important functionality of the RBAC DSL and is applied to check if a certain access can be granted or must be denied, that is if a certain subject `s` is allowed to perform operation `op` on object `ob` (see Figures 12 and 13). After receiving a `checkAccess` message, the Authorization concern is responsible to make an access decision for the corresponding access request. In order to reach this decision, it has to interact with

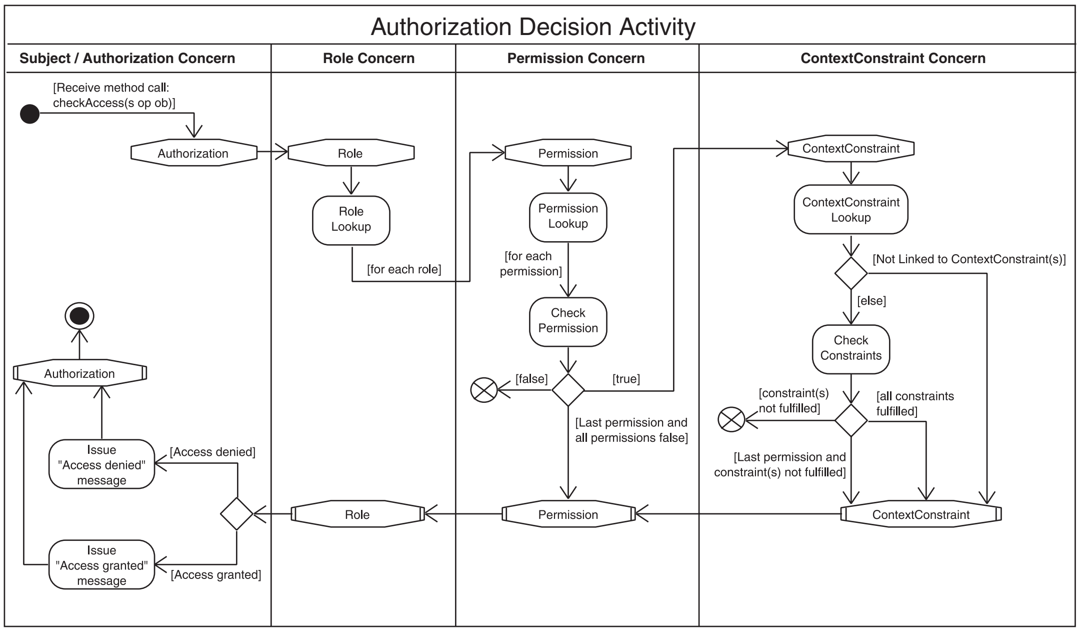

Authorization Decision Activity
Subject / Authorization Concern
Role Concern
Permission Concern
ContextConstraint Concern
[Receive method call: checkAccess(s op ob)]
Authorization
Role
Role Lookup
[for each role]
Permission
Permission Lookup
[for each permission]
Check Permission
[false]
[true]
[Last permission and all permissions false]
ContextConstraint
ContextConstraint Lookup
[Not Linked to ContextConstraint(s)]
[else]
Check Constraints
[constraint(s) not fulfilled]
[all constraints fulfilled]
[Last permission and constraint(s) not fulfilled]
ContextConstraint
Permission
Role
Authorization
Issue "Access denied" message
[Access denied]
Issue "Access granted" message
[Access granted]

Figure 13. Control flow of the authorization concern [73].

the Role concern. The Role concern then performs a role lookup procedure to determine the roles assigned to the respective subject. Subsequently, the Permission concern takes over to check the permissions that are assigned to the corresponding role objects.

However, as mentioned above, to grant a certain access request it is not sufficient to own the appropriate permission—all context constraints associated with the corresponding permission must be fulfilled at the same time. Thus, if a certain permission actually grants the access request (indicated by returning 'true') the ContextConstraint concern intercepts the control flow to check the constraints linked to this particular permission object. Depending on the result of the permission and constraint checking procedures an 'access granted' or an 'access denied' message is issued (see Figure 13).

7.3. Defining a concrete syntax

In the next step, we define a concrete syntax for our RBAC DSL. At this point we like to point out that we conducted two projects to build RBAC DSLs (see also Section 8). In the first project, we defined a declarative XML-based concrete syntax that was primarily intended as a transfer syntax for exchanging RBAC DSL models over network connections. In the second project, we defined an additional textual syntax for an embedded RBAC DSL that is based on the same language model (see Sections 7.1 and 7.2). This embedded DSL is intended to be used by software developers. Moreover, our target platform for the RBAC DSL (the xoRBAC platform) provides GUI tools to create and manage RBAC models (see [65,68,69]). This means, we can use the GUI tools of our target platform to manage the resulting RBAC models. In other words, with the xoRBAC target platform we get a graphical tool 'for free' and do not need to define an additional graphical syntax. Figure 14 shows the concrete syntaxes of our RBAC DSL. Below, we describe the development of the XML-based syntax in detail. In particular, we implemented a generator for the RBAC DSL that uses an XML parser to generate a DOM tree [74], which is again transformed to a valid runtime model for our target platform (see Section 7.4).

Figure 15 shows a tree view of the corresponding XML schema (actually it shows an excerpt for the most relevant elements). The schema defines structures to represent the different entities defined via the language model, such as a `subject` element, a `role` element, and so forth. Furthermore, to depict the role-to-subject assignment relation (see Section 7.1) the `subject` element includes a nested `role` element. However, to define a valid RBAC model with our DSL, the `role` element nested in `subject` must refer to a role defined via the top-level `role` element (see Figures 12 and 15). To enforce this constraint, we use the `key` and `keyref` elements of XML schema (for details see [75]). Other relations, such as permission-to-role assignment or mutual exclusion (separation of duties) are realized accordingly (see Figure 15).

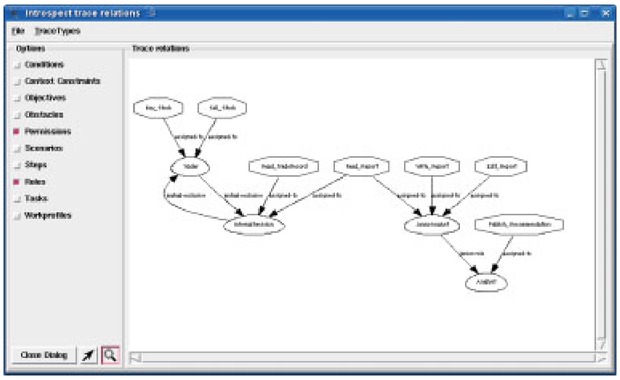
In the example from Figure 14, we have defined two subjects, `Sarah` and `Sophie`. Sarah owns the roles `Analyst` and `Trader`, while Sophie is assigned to the role `InternalRevision`. The Analyst role has a directly assigned permission `Publish_Recommendation` and inherits the permissions `Read_Report` and `Edit_Report` from its junior-role `Junior_Analyst`. The `Trader` role has two directly assigned permissions and is defined as mutual exclusive to the `InternalRevision` role (note that `ssd` is an abbreviation of 'static separation of duty', see also [67]). Finally, each permission consists of an $\langle \text{operation}, \text{object} \rangle$ pair—for example, the `Read_TradeRecord` permission defines the right to perform the operation `read` on objects of type `traderecord`.

```
<policySpec>
 <subject name="Sarah">
   <role>Analyst</role>
   <role>Trader</role>
 </subject>

 <subject name="Sophie">
   <role>InternalRevision</role>
 </subject>

 <role name="Analyst">
   <permission>Publish_Recommendation</permission>
   <junior>JuniorAnalyst</junior>
 </role>

 <role name="JuniorAnalyst">
   <permission>Read_Report</permission>
   <permission>Edit_Report</permission>
 </role>

 <role name="Trader">
   <permission>Buy_Stock</permission>
   <permission>Sell_Stock</permission>
   <ssd>InternalRevision</ssd>
 </role>

 <role name="InternalRevision">
   <permission>Read_TradeRecord</permission>
   <permission>Read_Report</permission>
   <ssd>Trader</ssd>
 </role>

 <permission name="Buy_Stock">
   <operation>buy</operation>
   <object>stock</object>
 </permission>

 <permission name="Read_TradeRecord">
   <operation>read</operation>
   <object>traderecord</object>
 </permission>

 ...

 <permission name="Write_Report">
   <operation>write</operation>
   <object>report</object>
 </permission>
</policySpec>
```

XML-based concrete syntax (external RBAC DSL)

```
Role create Analyst
Role create JuniorAnalyst
Analyst addJuniorRole JuniorAnalyst
Role create InternalRevision
Role create Trader
...
Permission create Read_Report -operation read -object report
Permission create Write_Report -operation write -object report
Permission create Edit_Report -operation edit -object report
...
JuniorAnalyst assignPermission Read_Report
JuniorAnalyst assignPermission Edit_Report
...
Subject create Sarah
Sarah assignRole Analyst
Sarah assignRole Trader

Subject create Sophie
Sophie assignRole InternalRevision
...
```

Textual concrete syntax for developers (embedded RBAC DSL)



Graphical representation of DSL models via platform specific tools

Figure 14. Concrete syntaxes of our RBAC DSL.

7.4. Platform integration

7.4.1. Mapping to the xoRBAC platform

As mentioned above, xoRBAC is used as the target platform for the DSL. xoRBAC provides a sophisticated RBAC service including a policy decision point and a policy repository for RBAC policies (for details on the xoRBAC platform, see [65,67,69,76]). To enable a thorough definition of DSL-to-platform-transformations, we thus need to map our DSL artifacts (including the behavior definition) to the functions provided by xoRBAC.

Figure 16 shows an excerpt of Figure 13 that includes ≪*concernSpec*≫ stereotypes for the ConcernStart nodes of the `Role` and the `Permission` concerns (see also [73]). In particular, these concernSpecs define that the behavior of the corresponding concerns is implemented through the `Role` and `Permission` classes of xoRBAC, respectively. Moreover, the `enterOperation` of both concerns is implemented via the `checkAccess` operation of the corresponding classes.

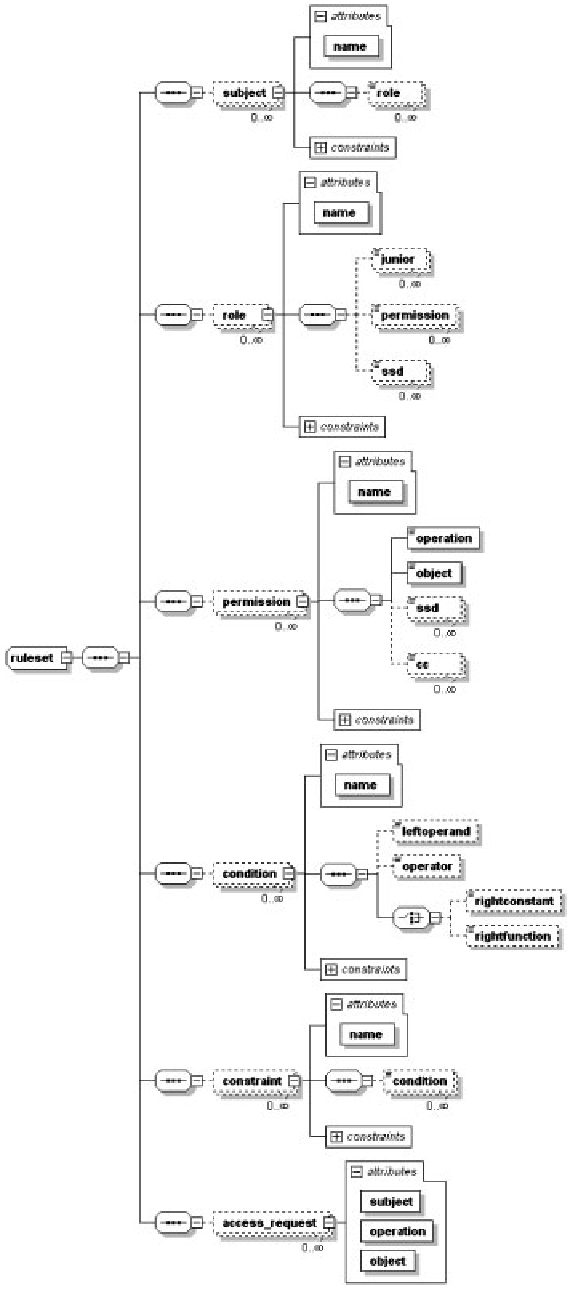


Figure 15. XML schema (excerpt) of a concrete syntax for the RBAC DSL.

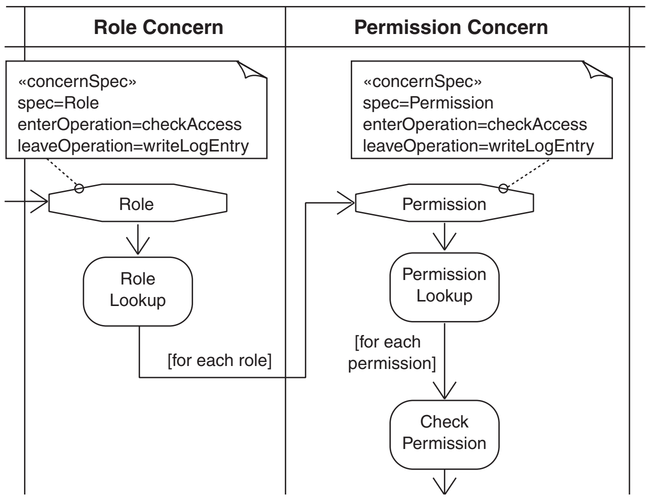


Figure 16. The role and permission concerns with concernSpec stereotypes [73].

In Figure 17, we see an interaction diagram that models the invocation sequence of the `Role` concern in xoRBAC. After receiving a `checkAccess` message, the corresponding `Role` object invokes the `checkAccess` method on the permission objects assigned to this particular role. If one of the permissions grants the access by returning 'true', the `Role` object immediately stops checking its permissions (see the 'break' InteractionOperator in Figure 17), writes a corresponding log entry, and returns the access decision.

7.4.2. Transformation of DSL language elements

After mapping the different DSL artifacts to our target platform, we define DSL-to-platform transformations that actually generate executable artifacts, which 'produce' the desired behavior on the target platform.

In our RBAC DSL, we used XML schema to define a concrete textual syntax (see Section 7.3). Now we have a number of different choices how to define transformations from XML-based DSL expressions to executable xoRBAC models. One option is to define a respective XSLT template. However, while XSLT is well suited to define transformations between relatively simple (and static) structures, the templates soon become very complicated for more complex and dynamic target domains. As a consequence such templates are hard to maintain in case the DSL and/or the target platform evolve.

Moreover, for the RBAC DSL it is not sufficient to simply transform a certain XML-based DSL expression to a corresponding executable model element of the xoRBAC platform. Instead, the transformations or, to be more precise, the generator applying the transformations must have a certain knowledge of the RBAC domain to produce executable models that conform to the semantics specified for the DSL (see also [4,69,73]).

For the above reasons, we decided to implement a generator component that reads the concrete syntax of our RBAC DSL and produces conforming executable xoRBAC models. Figure 18 shows

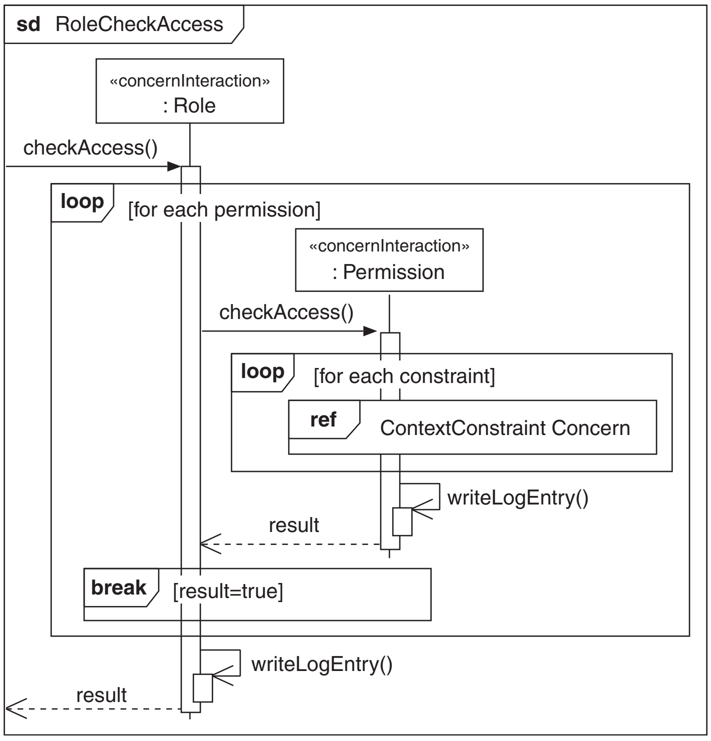


Figure 17. Interaction model for the Role concern [73].

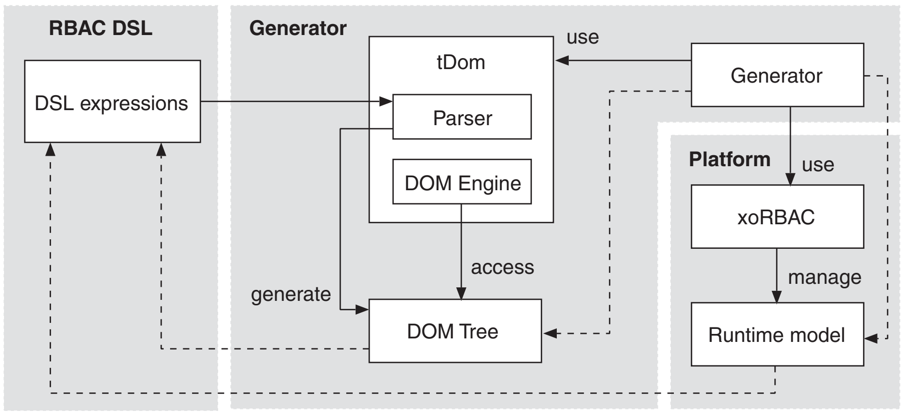


Figure 18. Transformation of DSL expressions: structural overview.

the conceptual structure of this generator. In particular, the generator uses tDOM [77] to parse XML-based DSL expressions and to access the resulting DOM tree [74]. The DOM tree provides a runtime representation of the XML input and allows to flexibly access the different DSL expressions.

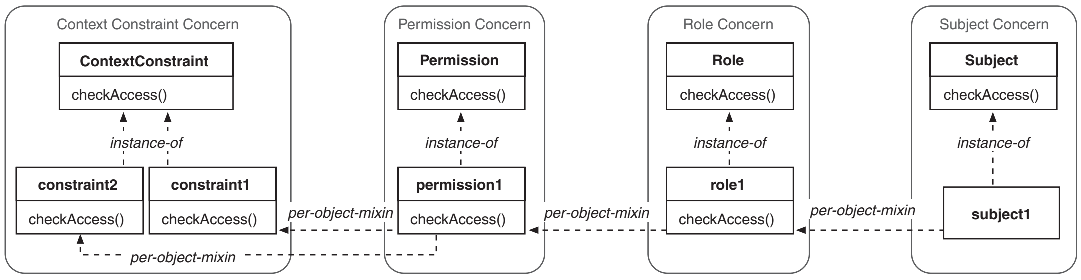


Figure 19. Example of an executable model generated from the RBAC DSL.

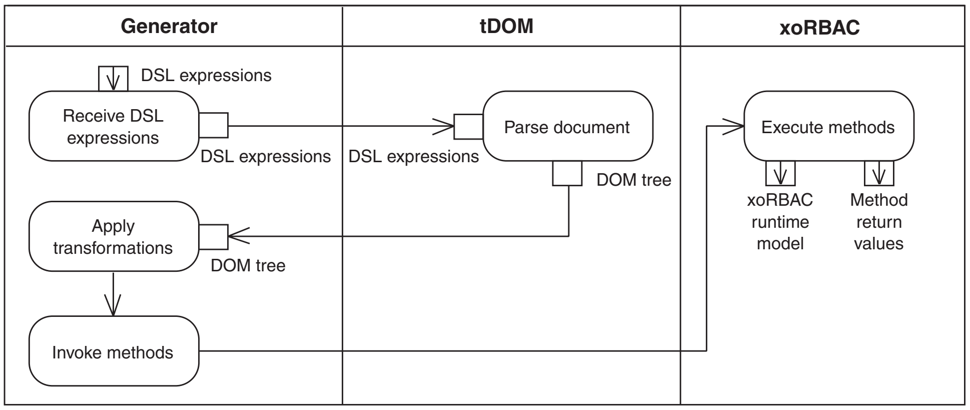


Figure 20. Transformation of DSL expressions: activity model.

The generator accesses the DOM tree via tDOM and generates a conforming executable xoRBAC model (see Figure 18). Figure 19 shows an example of such an xoRBAC runtime model.

Figure 20 shows an activity model that defines the corresponding action sequence in detail: After receiving a document with DSL expressions, the generator dispatches the corresponding XML document to tDOM and invokes the 'Parse document' action. The tDOM component parses the document, builds the respective DOM tree, and returns this DOM tree (respectively a reference to the DOM tree) back to the generator. The generator then applies the DSL-to-platform transformations and invokes the corresponding xoRBAC methods. As a result of these method calls, xoRBAC produces a conforming executable model and respective method return values that may be processed by the generator or other services.

8. LESSONS LEARNED FROM THE VARIOUS PROJECTS

The research presented in this paper is based on our experiences from various projects. In each of the corresponding projects we analyzed and verified many details of our DSL engineering process, and changed it, when mismatches occurred. In the course of the projects, at first, we started to construct an informal DSL engineering process guide and added all DSL development activities

Table II. Overview of DSL projects we analyzed in-depth.

Domain	Target platform/ technology	DSL type	Process variant	Project type	Our role(s)	Project duration
Bibliography management	Frag, HTML, Latex	Embedded	Mockup language driven	Research project	DSL Designer, Developer	3 month
BPMN process modeling	BPEL process engine, WSDL	External	Language model driven	Research project	Architect, Observer	1 year
Compliance metadata	Java, BPEL, HTML	External	Language model driven	Research project	Architect, Observer	6 month
DSL editor specification	Eclipse, Java, Frag	Embedded	Mockup language driven	Research project	Architect, Observer	1 year
Data backup	Tcl/XOTcl	Embedded	Mockup language driven	Case study	DSL Designer, Developer	6 weeks
Document archiving	C-based custom platform	External	Extracting DSL from existing system	Industry project prototype	DSL Designer, Developer, Observer	2 years
Multimedia home platform	Frag/Java MHP platform	Embedded	Mockup language driven	Industry project prototype	DSL Designer, Developer, Observer	1 year
Role-based access control	XML, xoRBAC	External	Language model driven	Research project	Domain Expert, DSL Designer, Developer	2 month
Role-based access control	Tcl/XOTcl	Embedded	Extracting DSL from existing system	Case study	Domain Expert, DSL Designer, Developer	4 weeks
SOA architectural views	BPEL process engine, WSDL, Java	External	Language model driven	Research project	Architect, Observer	2 years
Software testing	Tcl/XOTcl	Embedded	Mockup language driven	Research project	DSL Designer, Developer	3 month
Software architectural knowledge	Frag, HTML, Latex	Embedded	Mockup language driven	Research project	DSL Designer, Developer	3 month
Software documentation	Frag, HTML, Latex	Embedded	Mockup language driven	Research project	DSL Designer, Developer	3 month
Workflow definition	Tcl/XOTcl	Embedded	Mockup language driven	Case study	Domain Expert, DSL Designer, Developer	2 month

that we observed in our data. However, we then realized that this first (intermediate) result did not completely fit our observed data from the projects. Hence, we refined the result by explicitly describing the activities that we conduct when engineering a DSL, including their respective inputs and outputs. This evolved version was still unsatisfactory, as it neglected the implicit process aspects that we observed in our analysis. For this reason, in the third iteration, we added the tailoring step reported in Section 6 that better reflects the observed DSL projects and other available data. In a similar way, all details of our results were also refined in an iterative manner.

In addition to many other projects, we observed and analyzed 14 DSL projects in-depth, which are summarized in Table II[§]. To illustrate the main lessons learned in terms of the selection and tailoring of the DSL development process, as well as the different influencing factors, we summarize the main considerations for each of these projects:

- *Bibliography management*: This project was a small-scale project with only one developer in a University institute's Web site and information system administration setting. Test-driven development was used as the software development approach. The developer understood the domain and was a DSL user himself. A domain-model could rather easily be derived from the concrete syntax because the domain was well understood. Other potential users of the DSL where only initially involved—and hence a mockup language-driven micro-process was chosen because mainly the syntax needed to be agreed upon. Only one iteration of the mockup language-driven process was needed to develop an initial version of the DSL.
- *BPMN process modeling*: This project was a longer duration project embedded in a research project. Two people were involved in architecting the DSL. The development was mainly carried out by one developer, but access to the DSL users was very limited (only a few meetings). An agile process and test-driven development were chosen for the overall project. The main focus of the project was on the evolution of the models—the textual DSL syntax was mainly used as a convenient way to populate the models. A highly iterative version of the language model-driven micro-process was chosen, as it was the most efficient way to facilitate the communication of architects and developers between the project's regular releases.
- *Compliance metadata*: This project had a very similar setting as the BPMN process modeling and for the same reasons a language model-driven micro-process was chosen here.
- *DSL editor specification*: This project also had a very similar setting as the previous two projects, but there was one major difference: the syntax of the language played an important role, as user acceptance was regarded as a crucial success factor for the project. Hence, a mockup language-driven micro-process was chosen and executed in a highly iterative fashion. The mockup language design was frequently discussed with potential DSL users to get user input early on.
- *Data backup*: This project was a case study to develop a DSL for the specification of backup policies. The intention was to build an embedded DSL for XOTcl developers. Thus, a major requirement was the specification of a syntax that can easily be used by XOTcl developers. For this reason, we chose an agile development approach based on the mockup language-driven

[§]In the table, the projects are categorized as research projects, industry projects, or case studies. The *research project* category refers to DSLs that were developed as part of a larger research project. Likewise DSLs developed in an industry project are assigned to the *industry project* category. The *case study* category is used for projects whose sole purpose was the development of a DSL without a direct relationship to a larger research or industry project.

process variant. In particular, we used our own XOTcl expertise and collected feedback from two other domain experts (XOTcl developers) to iteratively adapt the syntax. The backup DSL was then implemented as an embedded XOTcl DSL that can be used in other XOTcl programs.

- *Document archiving*: This project was an industry project with the goal to reengineer a large-scale industry system. The system had a large set of developers. Little languages were used for various tasks in this system. The goal of the project was to architecturally improve the system. The existing little languages of the system were improved using a unified DSL concept. The domain concepts as implemented by the existing system were not in question. The main users of the DSL were the system developers and teams configuring the system for customers. For these reasons, it made sense to follow the 'extracting DSL from existing system' micro-process—and then gradually improve the design in frequent meetings.
- *Multimedia home platform*: This project was an industry project with the goal to develop a proof-of-concept prototype. The main development work was carried out by one PhD student. The design was carried out in a larger group, and also frequently discussed with 10 potential customers. However, over the course of this project these meetings were rather infrequent. The project was then taken over by the developers of the company. The mockup language-driven micro-process was chosen, as people on the customer sites were unfamiliar with modeling concepts such as the UML. The syntax-driven discussion of the DSL concepts, based on examples, helped to obtain as much as possible of the domain experience in a short time frame.
- *Role-based access control*: In this project we built an XML-based RBAC DSL. Because of our expertise in the RBAC domain, we chose the language-driven variant of the development process. Section 7 describes this project in detail.
- *Role-based access control*: This project was a follow-up case study of our first RBAC DSL project (see also Section 7). In particular, we aimed to build an embedded DSL for the xoRBAC platform. In this project, changing or extending the xoRBAC platform was not intended. Moreover, an API documentation of xoRBAC was available. Therefore, we chose to follow the 'extracting DSL from existing system' variant of our process. Because of our experiences from the first RBAC DSL project, and because of our detailed knowledge of the xoRBAC platform, we were able to conduct the project in only 4 weeks. No additional domain experts were involved in the project.
- *SOA architectural views*: This project was a research project and development was carried out by five PhD students. Domain knowledge is provided by regular architecting sessions, patterns on the domain, and infrequent discussions and reviews with industrial DSL users. Because the different developers work on separate areas and DSLs with some overlaps, we decided to follow the language model-driven micro-process, as it turns out to be efficient for the discussions. This was feasible because most of the domain experts were familiar with modeling abstractions.
- *Software testing*: This DSL project was part of a larger research project where we investigated the testing of dynamically changing runtime structures and dynamically changing runtime behavior of software programs. Our target platform was the dynamic object-oriented programming language XOTcl, and the intended users (i.e. domain experts) are XOTcl developers. Thus, the goal of this project was to build an embedded testing DSL in XOTcl. At the beginning of the project, we only had basic knowledge of the testing domain. Therefore, we decided to choose the mockup language-driven process variant and started with the definition of a concrete syntax. Two other XOTcl developers were consulted as domain experts to provide

feedback on the embedded DSL. Both domain experts were available on short notice. Based on the discussions, the interim versions of the resulting mockup language were iteratively refined.

- *Software architectural knowledge*: This project was a small-scale project that was conducted as part of the infrastructure work needed for a research project. As it has a very similar setting as the bibliography management project, we used the mockup language-driven micro-process for the same reasons.
- *Software documentation*: This project had a similar setting as the bibliography management project and hence we used the mockup language-driven micro-process for the same reasons.
- *Workflow definition*: This project was a case study to develop an embedded DSL for the specification of workflows on the source code level. In particular, the DSL should provide language abstractions for the definition of processes that consist of several tasks, which are performed sequentially and/or in parallel. Moreover, the tasks are assigned to roles and only owners of an adequate role are allowed to perform the corresponding tasks. From our experiences, we knew that it is often convenient to develop embedded DSLs using the mockup language-driven process variant. As always when using this process variant, the DSL was developed in an iterative fashion. In this project, we had one external domain expert who provided valuable insights into the process modeling domain. The domain expert was available on short notice, which allowed for rapid feedback loops.

9. CONCLUDING REMARKS

We introduced an experience-based approach for the systematic development of DSLs. The approach provides guidance for DSL developers and facilitates the communication between engineers and domain experts. Moreover, it provides an integrated view on the respective domain and the different models defining the corresponding DSL. Our research is based on experiences from numerous DSL projects and a substantial amount of studies and experiments in the field (see Section 8). In addition to our own experiences, we applied techniques like observations and interviews with other professional developers to shape the approach, and we successfully applied the approach in follow-up projects.

In particular, we identified the activities that we conduct when engineering a DSL. We described the definition process of DSL language models and explained the activities to define a behavior definition. Moreover, we described how we specify a concrete syntax that conforms to the DSL's abstract syntax, and the integration of the different DSL models with the respective target platform. Our engineering process is intended to be performed in a incremental fashion to allow for an evolutionary enhancement of DSLs, including the definition of different concrete syntaxes for the same DSL. Moreover, to achieve a high flexibility we described how the engineering process can be tailored to the specifics of actual projects. This means, our approach allows for the adaption of the development process, for example by adapting the control flow of the process or by adding or removing (sub)activities. Furthermore, the approach can be used with any suitable modeling technique and does not prescribe specific modeling languages. Therefore, DSL developers are free to chose the most appropriate tools and techniques when engineering a DSL.

In principle, our approach is applicable in each project that aims to develop DSLs in an MDSD context and/or DSLs as extensions for dynamic programming languages. In Section 6, we presented different tailored variants of our process that we applied in several projects. However, if completely different or new/innovative techniques are used for designing and implementing the DSLs, it is likely that the development process must be tailored in a different way. In general, tailoring allows to customize our DSL development process to project-specific requirements. Aside from defining a customized development process for a DSL project, the tailoring phase discloses possible differences or varieties of a certain DSL project. If such differences are found in more than one project, they are further investigated if they can or must be included in our overall approach to DSL development. In our future work, we therefore continue to validate and enhance the approach in an evolutionary fashion.

A benefit of using our approach is that DSL designers and developers can rely on proven micro-processes and can focus on their work task at hand, instead of specifying their own DSL processes. Compared with general-purpose processes, our approach has the benefit to focus on the specific activities and artifacts of DSL development. That is, it ensures that the specific tasks of DSL development are explicitly considered and not forgotten. The micro-processes can be adapted and used in most general-purpose SW engineering processes, but tailoring them into these general-purpose processes requires some effort. Thus, compared with rigid process models, explaining step by step how to develop a DSL, our approach has the benefit to offer more freedom and design choices. However, as a drawback some effort for tailoring is required.

ACKNOWLEDGEMENTS

This work was partially supported by the European Union FP7 project COMPAS, grant no. 215175.

REFERENCES

1. Kelly S, Tolvanen J. *Domain-specific Modeling*: *Enabling Full Code Generation*. Wiley: New York, 2008.
2. Mernik M, Heering J, Sloane A. When and how to develop domain-specific languages. *ACM Computing Surveys* 2005; **37**(4):316–344.
3. Stahl T, Völter M. *Model-driven Software Development*. Wiley: New York, 2006.
4. Strembeck M, Zdun U. Definition of an aspect-oriented DSL using a dynamic programming language. *Proceedings of the Workshop on Open and Dynamic Aspect Languages* (*ODAL*), Bonn, Germany, March 2006.
5. Wile D. Lessons learned from real DSL experiments. *Science of Computer Programming* 2004; **51**(3):265–290.
6. Zdun U. Concepts for model-driven design and evolution of domain-specific languages. *Proceedings of the International Workshop on Software Factories at OOPSLA*, San Diego, CA, 2005; 1–6.
7. Bierhoff K, Liongosari E, Swaminathan K. Incremental development of a domain-specific language that supports multiple application styles. *Proceedings of the Sixth OOPSLA Workshop on Domain-specific Modeling*, Portland, OR, October 2006.
8. Luoma J, Kelly S, Tolvanen J. Defining domain-specific modeling languages: Collected experiences. *Proceedings of the Fourth OOPSLA Workshop on Domain-specific Modeling*, Vancouver, BC, Canada, October 2004.
9. Thibault S, Marlet R, Consel C. Domain-specific languages: From design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering* (*TSE*) 1999; **25**(3):363–377.
10. Grant E, Narayanan K, Reza H. Rigorously defined domain modeling languages. *Proceedings of the Fourth OOPSLA Workshop on Domain-specific Modeling*, Vancouver, BC, Canada, October 2004.
11. Hudak P. Building domain-specific embedded languages. *ACM Computing Surveys* 1996; **28**.
12. Kosar T, Martinez-Lopez P, Barrientos P, Mernik M. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology* 2008; **50**(5):390–405.
13. Paul R. Designing and implementing a domain-specific language. *Linux Journal* 2005; **2005**(135).
14. Spinellis D. Notable design patterns for domain-specific languages. *Journal of Systems and Software* 2001; **56**(1):91–99.
15. van Deursen A, Klint P. Little languages: Little maintenance? *Journal of Software Maintenance* 1998; **10**(2):75–92.
16. Fowler M. Language workbenches: The Killer–App for domain specific languages? Available at: http://martinfowler.com/articles/languageWorkbench.html [31 October 2008].

17. Cuadrado JS, Molina JG. Building domain-specific languages for model-driven development. *IEEE Software* 2007; **24**(5):48–55.
18. Greenfield J, Short K, Cook S, Kent S. *Software Factories*: *Assembling Applications with Patterns*, *Frameworks*, *Models & Tools*. Wiley: New York, 2004.
19. Cleenewerck T. Component-based DSL development. *Proceedings of the Second International Conference on Generative Programming and Component Engineering* (*GPCE*). Springer: Berlin, September 2003; 245–264.
20. Jacobson I, Booch G, Rumbaugh J. *The Unified Software Development Process*. Addison-Wesley: Reading, MA, 1999.
21. Beck K, Fowler M. *Planning Extreme Programming*. Addison-Wesley: Reading, MA, 2000.
22. Klint P, Lämmel R, Verhoef C. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*) 2005; **14**(3):331–380.
23. Roberts D, Johnson R. Evolve frameworks into domain-specific languages. *Proceedings of the Pattern Languages of Programs Conference* (*PLoP*), Monticello, IL, September 1996.
24. Roberts D, Johnson R. Patterns for evolving frameworks. *Pattern Languages of Program Design 3*. Addison-Wesley: Reading, MA, 1997; 471–486.
25. Allen N, Shaffer C, Watson L. Building modeling tools that support verification, validation, and testing for the domain expert. *Proceedings of the 37th Winter Simulation Conference* (*WSC*), Orlando, FL, December 2005; 419–426.
26. Logix: Multi-language programming homepage. Available at: http://www.livelogix.net/logix/ [31 October 2008].
27. Valentin E, Verbraeck A. Guidelines for designing simulation building blocks. *Proceedings of the 34th Winter Simulation Conference* (*WSC*), San Diego, CA, December 2002; 563–571.
28. Landin P. The next 700 programming languages. *Communications of the ACM* (*CACM*) 1966; **9**(3):157–166.
29. Freeze J. Creating DSLs with Ruby. *Ruby Code & Style*. Artima, Inc.: Mountain View, CA, 2006. Published online: http://www.artima.com/rubycs/articles/ruby_as_dsl.html.
30. The Object Management Group. OMG Unified Modeling Language (OMG UML): Superstructure, Version 2.1.2, formal/2007-11-02. Available at: http://www.omg.org/technology/documents/formal/uml.htm [31 October 2008].
31. Kurtev I, Bézivin J, Jouault F, Valduriez P. Model-based DSL frameworks. *Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems*, *Languages*, *and Applications* (*OOPSLA*), Portland, OR, October 2006; 602–616.
32. The ATLAS model management architecture (AMMA) homepage. Available at: http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/ [31 October 2008].
33. Eclipse modeling framework project (EMF)—Homepage. Available at: http://www.eclipse.org/modeling/emf/ [31 October 2008].
34. The generic modeling environment homepage. Available at: http://w3.isis.vanderbilt.edu/Projects/gme/ [31 October 2008].
35. Ledeczi A, Maroti M, Bakay A, Karsai B. The generic modeling environment. *Proceedings of the IEEE International Workshop on Intelligent Signal Processing* (*WISP*), Budapest, Hungary, May 2001.
36. Cook S, Jones G, Kent S, Wills A. *Domain-specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional: Reading, MA, 2007.
37. Balasubramanian K, Gokhale A, Karsai G, Sztipanovits J, Neema S. Developing applications using model-driven design environments. *IEEE Computer* 2006; **39**(2):33–40.
38. Open Architecture Ware. openArchitectureWare. Available at: http://www.openarchitectureware.org/ [31 October 2008].
39. Wang Q, Gupta G. Rapidly prototyping implementation infrastructure of domain specific languages: A semantics-based approach. *Proceedings of the ACM Symposium on Applied Computing* (*SAC*), Santa Fe, NM, March 2005; 1419–1426.
40. Jones N. An introduction to partial evaluation. *ACM Computing Surveys* 1996; **28**(3):480–503.
41. Schmidt DC. Model-driven engineering—Guest Editor's introduction. *Computer* 2006; **39**(2):25–31.
42. Selic B. The pragmatics of model-driven development. *IEEE Software* 2003; **20**(5):19–25.
43. Bentley J. Programming Pearls—Little languages. *Communications of the ACM* 1986; **29**(8):711–721.
44. Herndon R, Berzins V. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering* (*TSE*) 1988; **14**(6):803–809.
45. Raymond E. *The Art of Unix Programming*. Addison-Wesley: Reading, MA, 2003.
46. Graham P. *On LISP—Advanced Techniques for Common LISP*. Prentice-Hall: Englewood Cliffs, NJ, 1993.
47. Lamport L. *LaTeX*: *A Document Preparation System* (2nd edn). Addison-Wesley: Reading, MA, 1994.
48. LaTeX project homepage. Available at: http://www.latex-project.org [31 October 2008].
49. Pemberton S, Austin D, Celik T, Dominiak D, Elenbaas H, Epperson B, Ishikawa M, Matsui S, McCarron S, Navarro A, Peruvemba S, Relyea R, Schnitzenbaumer S, Stark P. XHTML 1.0 The Extensible HyperText Markup Language (2nd edn)—A reformulation of HTML 4 in XML 1.0. Available at: http://www.w3.org/TR/xhtml1 [31 October 2008].
50. Information technology—Syntactic metalanguage—Extended BNF—ISO/IEC 14977:1996. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tcashcatalogue_detail.htm?csnumber=26153 [31 October 2008].
51. Chamberlin D, Boyce R. Sequel: A structured english query language. *Proceedings of the ACM SIGFIDET* (*now SIGMOD*) *Workshop on Data Description*, *Access and Control*, Ann Arbor, MI, 1974; 249–264.
52. Information technology—Database languages—SQL—Part 1: Framework (SQL/framework)—ISO/IEC 9075-1: 2003. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tcashcatalogue_detail.htm?csnumber=34132 [31 October 2008].

53. Zdun U. Tailorable language for behavioral composition and configuration of software components. *Computer Languages, Systems and Structures: An International Journal* 2006; **32**(1):56–82.
54. The Object Management Group. OCL 2.0 specification, Version 2.0, formal/06-05-01. Available at: http://www.omg.org/technology/documents/formal/uml.htm [31 October 2008].
55. Mens T, Gorp PV. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science* 2006; **152**:125–142.
56. Sendall S, Kozaczynski W. Model transformation: The heart and soul of model-driven software development. *IEEE Software* 2003; **20**(5):42–45.
57. Czarnecki K, Eisenecker U. *Generative Programming—Methods, Tools, and Applications*. Addison-Wesley: Reading, MA, 2000.
58. Microsoft. Microsoft modeling platform (code named Oslo). Available at: http://msdn.microsoft.com/en-us/library/cc709420.aspx [31 October 2008].
59. Clements P, Northrop L. *Software Product Lines—Patterns and Practices*. Addison-Wesley: Reading, MA, 2002.
60. Evans E. *Domain-driven Design—Tackling Complexity in the Heart of Software*. Addison-Wesley: Reading, MA, 2004.
61. Fowler M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley: Reading, MA, 1996.
62. Weiss D, Lai C. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley, Longman Publishing: Reading, MA, New York, 1999.
63. Highsmith J, Cockburn A. Agile software development: The business of Innovation. *IEEE Computer* 2001; **34**(9):120–122.
64. Highsmith J, Cockburn A. Agile software development: The people factor. *IEEE Computer* 2001; **34**(11):131–133.
65. Neumann G, Strembeck M. Design and implementation of a flexible RBAC-service in an object-oriented scripting language. *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS)*, Philadelphia, PA, November 2001; 58–67.
66. Neumann G, Strembeck M. A scenario-driven role engineering process for functional RBAC roles. *Proceedings of Seventh ACM Symposium on Access Control Models and Technologies (SACMAT)*, Monterey, CA, June 2002; 33–42.
67. Strembeck M. Conflict checking of separation of duty constraints in RBAC—Implementation experiences. *Proceedings of the Conference on Software Engineering (SE 2004)*, Innsbruck, Austria, February 2004; 224–229.
68. Strembeck M. A role engineering tool for role-based access control. *Proceedings of the Third Symposium on Requirements Engineering for Information Security (SREIS)*, Paris, France, August 2005.
69. Strembeck M, Neumann G. An integrated approach to engineer and enforce context constraints in RBAC environments. *ACM Transactions on Information and System Security (TISSEC)* 2004; **7**(3):392–427.
70. Ferraiolo D, Kuhn R, Chandramouli R. *Role-based Access Control* (2nd edn). Artech House: Boston, 2007.
71. Sandhu R, Coyne E, Feinstein H, Youman C. Role-based access control models. *IEEE Computer* 1996; **29**(2):38–47.
72. Ferraiolo D, Barkley J, Kuhn D. A role-based access control model and reference implementation within a corporate Intranet. *ACM Transactions on Information and System Security (TISSEC)* 1999; **2**(1):34–64.
73. Strembeck M, Zdun U. Modeling interdependent concern behavior using extended activity models. *Journal of Object Technology (JOT)* 2008; **7**(6):143–166.
74. Hors AL, Hegaret PL, Wood L, Nicol G, Robie J, Champion M, Byrne S. Document object model (DOM) level 3 core specification, Version 1.0. W3 Consortium Recommendation. Available at: http://www.w3.org/TR/DOM-Level-3-Core [31 October 2008].
75. Thompson H, Beech D, Maloney M, Mendelsohn N. XML schema Part 1: Structures (2nd edn). W3C Recommendation. Available at: http://www.w3.org/TR/xmlschema-1 [31 October 2008].
76. Zdun U, Strembeck M, Neumann G. Object-based and class-based composition of transitive mixins. *Information and Software Technology* 2007; **49**(8):871–891.
77. tDOM homepage. Available at: http://www.tdom.org [31 October 2008].

Copyright © 2009 John Wiley & Sons, Ltd.