Automating Language Evolution

Markus Pizka, Elmar Jürgens
Technische Universität München
Institut für Informatik
Germany – 85748 Garching
{pizka,juergens}@in.tum.de

Abstract

The design and implementation of complex software systems inherently spans multiple levels of abstractions. The concepts of each level of abstractions and their interplay are represented by formal languages that are either implicitly known or explicitly defined. Achieving high productivity in software development and maintenance is thus strongly connected with ruling the complexity of multi-level language design and evolution. This paper explains the necessity for automating multi-level language evolution, discusses its challenges and proposes concepts as well as a prototypical tool that support the incremental co-evolution of a staged language and program generation architecture. This approach reduces the cost of language maintenance and paves the ground for an incremental and bottom-up oriented way of developing domain specific languages.

1 Increasing Productivity Through DSLs?

Albeit three decades of intense research and significant progress in research and practice, the development and maintenance of software systems still constitutes a time-consuming, costly and risky endeavor. The reduction of software development and maintenance costs thus remains a research topic of paramount importance to software engineering. A basic idea behind many approaches that attempt to reduce these costs is to increase the productivity of software developers. One approach to raise productivity that has received increased attention in recent years, are domain specific languages.

1.1 Productivity Through Specialization

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [40]. The key characteristic of domain specific languages is their specialization to a problem domain. This specialization allows them to offer language constructs and abstractions tailored to the class of problems from this domain. DSLs typically allow these problems to be described very directly and concisely, requiring less developer effort than general purpose programming languages. DSLs thus have the potential to increase productivity and decrease costs of software development. Particularly prominent examples for the benefits of DSLs are found in the compiler construction field with specialized languages and generators amongst others for hashing (e. g. gperf), scanners [17], and parsers [9]. Tools for the interactive design of graphical user interfaces, the query language SQL, interface definition languages like WSDL [44], object-relational mapping tools like Hibernate [14], or modeling languages like MATLAB/Simulink add further examples for the benefits of DSLs, i. e. purpose-built languages with powerful generators.

Besides reduced development time and cost, DSLs are a promising mean to decrease the maintenance cost of software because of the reduced code size and increased comprehensibility, both due to the higher expressiveness of DSLs compared to general-purpose languages. The decrease of maintenance costs is particularly important since 80% of the costs of software are usually not devoted to development but to maintenance [24, 34, 31].

However, DSLs only help to reduce overall maintenance costs, as long as the costs for development and maintenance of the DSL itself can be amortized. If unanticipated changes to a domain require changes to the DSL definition, maintenance costs can be high, as noted in [6].

1.2 Limitations

However, the success of DSLs is clearly still limited to few niches. To the extent of our own practical experiences and that of our commercial partners, the bulk of code that gets written world wide and day by day, whether information systems in financial institutions or flight control systems for commercial aircrafts, makes little use of DSLs if any. Languages like Risla for Financial Products [6] are rare exceptions to the rule.

We state that there are at least three major reasons for this which we will explain more precisely in the following paragraphs. First are the costs of designing and implementing DSLs. Second, the limited capabilities of one-step generation and third, the constant need to evolve DSLs.

The multi-level language evolution concept and tool-support presented in this paper largely increases the applicability of DSLs by overcoming exactly these three current core difficulties in DSL design and implementation.

2 DSL Challenges

The goal of the concept and tool-set presented in this paper is to increase the long-term applicability of DSLs by supporting an

evolutionary and bottom-up oriented style for DSL design and implementation. The rationale for this approach is the need to overcome three major obstacles that we detail in the following paragraphs.

2.1 DSLs Are Expensive to Build

Though DSLs promise substantial gains in productivity the development of DSLs itself is expensive and troublesome. [26] summarizes the difficulties of building DSLs adequately as follows:

> DSL development is hard, requiring both domain knowledge and language development expertise. Few people have both. Not surprisingly, the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the application library stage.

Clearly, one of the primary contributions of DSLs is enabling reuse, i. e. reuse of abstractions and reuse of the knowledge about how to implement these abstraction in different contexts. As such, DSLs must cope with the same economic challenges like any other reuse oriented approach [35]. Building reusable components requires a costly analysis of the domain and its variability followed by an even more expensive implementation of reusable components[1]. The costs of planning and building such generalized components are usually a multiple of the costs of building a concrete solution to a particular problem [35, 3]. Hence, building DSLs only pays-off after repeated successful use of the DSL. However, due to the constant change of requirements and the execution environment [23], the future use of a DSL is uncertain and building DSLs is economically risky. Note, that this risk increases with the degree of specialization respectively the potential benefit.

Requirement 1 ***(Stepwise Bottom-Up Generalization)*** *To reduce the uncertainty of the benefit of DSL design and implementation, DSLs should be built in an incremental and bottom-up oriented manner instead of the currently predominant top-down and big-bang like approach. To support this style of DSL development, means for the step-wise generalization of existing concepts and solutions as needed are required.*

If DSL can be built by gradually abstracting and flexibilizing existing solutions (including DSLs) as needed, then the cost of building a DSL will never exceed the costs of developing the desired new solution from scratch significantly but often provide immediate pay-offs. No effort has to be put into speculation about future requirements and there is no need for risky in advance investment into flexibility that could possibly be needed in the future.

2.2 Generators are no Oracles

A DSL usually requires a generator that reads a word of the domain specific language and produces a word in the desired target language. Obviously, program generators for DSLs are nothing but program transformation systems providing a translation from a higher to a lower level language, which is also called *synthesis* or *compilation* [41]. Hence, program generators are subject to the same inherent limitations like conventional compilers that translate from C to Assembler or Java to Byte-code though they usually operate on a higher level of abstraction.

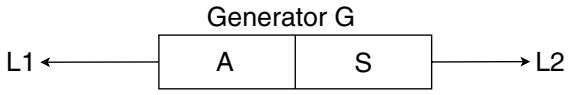

Figure 1. Generator basics

Figure 1 illustrates the basic structure of a program generator G. G reads words $v \in L_1$ and produces words $w \in L_2$ by first performing an analysis A of v and then synthesizing (S) result w. The whole benefit of this strategy corresponds to the distance between the level of abstractions of the input and the output languages that is $A(L_1)$ and $A(L_2)$. Basically[2], there are three different possibilities:

$A(L_1) = A(L_2)$: The DSL L_1 and its generator G are useless from a productivity perspective. G does not contribute any decisions to the implementation. All details of the output w are already specified in the input v. G only rephrases v which might increase readability of v compared to w but not reduce its complexity.

$A(L_1) > A(L_2)$: This means that some details of w are not described in v but G decides on the implementation of these details. Examples are the allocation of memory for local variables in C compilers or the optimization of an SQL query. Here, the gains for the user of L_1 are obvious. By leaving some decisions on *how to* implement w up to G, v becomes shorter and more declarative, by describing G *what to* implement.

$A(L_1) >> A(L_2)$: Unfortunately, the possibility to stretch the distance between $A(L_1)$ and $A(L_2)$ is very limited because of computational complexity. Even basic decisions, like the allocation of registers, turn out as being NP-hard [4]. Though not proved, yet, it can be assumed that mapping higher level descriptions such as a financial service specification to Java classes and objects will have to cope with similar complexity issues. Usually this complexity is circumvented by accepting suboptimal decisions and thereby reduced quality of the output. The wider the gap between $A(L_1)$ and $A(L_2)$ gets, the less information will be available to the decision maker G resulting in a weaker result w in terms of performance, reliability, usability and so on. Clearly, reduced quality is counterproductive for reuse. McIlroy stated in 1968 *"No user of a particular member of a family should pay a penalty in unwanted generality"* [25]. E. g. current Object-Relational mapping tools suffer from this trade-off.

This leads to the following contradictory observation:

1. The benefit that can be gained from a single generator is inherently and severely limited. We are convinced that there

[1] In case of DSLs represented through the domain language and a code generator.

[2] Ignoring reverse engineering, where $A(L_1) < A(L_2)$, since it cannot increase productivity

will not be a single flexible DSL for some high-level business domain with a generator that maps it to high-quality Java code!

2. To gain a significant advantage from a DSL the gap between the level of abstractions of the input and the output has to be wide (see $A(L_1) > A(L_2)$).

The concept and tool-set presented in this paper aims at solving this concept by fulfilling the following requirement.

Requirement 2 ***(DSL Layering)***
The design and implementation of a DSL should not be limited to a one-step compilation but support layers of DSLs and a staged generation process with additional user input at each stage.

Note, that staging further increases the complexity and costs of building and maintaining the DSL as discussed in 2.1 because changes on one stage might affect other stages, too. Again, ruling this complexity requires a tool-set that aids in gradually adapting the DSL hierarchy as needed (see requirement 1).

2.3 Language and Word Evolution

While building a DSL is costly, building layered DSLs is even more expensive and maintaining a single or even layered DSL is probably costliest simply because nothing is more constant than change entailing a constant need for evolution [23].

The design and implementation of a DSL trivially depends on the requirements of the domain. With the exception of DSLs that model a technical domain, such as regular expressions or SQL, the requirements are directly connected with the business processes in this domain. Unfortunately, nothing is more volatile then business processes [29] simply because business process agility is the mean to achieve competitive advantages. Due to the Frame-Problem [27], most of the future changes can not be anticipated in advance.

This poses a serious difficulty for DSLs. On the one hand, a DSL should be high-level or in other words as close to the business processes as possible to provide increased productivity compared to a general purpose language. On the other hand, the tighter the DSL is connected with the business processes, the more fragile it gets and the more often it will have to be changed which in turn reduces the benefits of possible reuse.

A non-trivial change to an existing DSL L leading to a new DSL version L' requires the following three major steps:

1. Change of the definition of L – its syntax and semantics.

2. Adaption of all tools processing L; at least the corresponding compiler or generator but maybe also syntax aware editors (e. g. highlighting), debugger, etc.

3. Transformation of all already existing words (programs) $w \in L$ into language L'.

As an alternative to step 3 one could also maintain older versions of DSLs so that words in older versions of the language could still be used and changed independently of newer versions of the language. However, this would create a complicated configuration management problem and in addition to this, users of older versions of the language could not benefit from any advantages of newer versions and new tools. In practice, this drawbacks forces users to migrate their words to the new version.

Hence, most DSLs will have to evolve over time including the tools that process these DSLs and words written in these languages. Without adequate tool-support, DSL evolution is a complex, time-consuming, and error-prone task that severely hampers the long-term success of a DSL.

Requirement 3 ***(Automated Co-Evolution)*** *DSL maintenance is inevitable for most realistic domains and requires adequate tool support. The transformation of existing words and the adaptation of language processing tools according to changes of the language has to be automated as far as possible.*

Note that this requirement complements requirement 1 because the tool-supported co-evolution of language, tools and words is a contribution to stepwise bottom-up generalization as formulated in requirement 1.

3 Related Work

The work presented in this paper combines DSLs [40] and generative programming [7] with elements of program transformation [41] and compiler construction, as well as software evolution [23]. Within this general context, the evolution concept has strong relations with Grammar Engineering, document transformation and language evolution as described in the following paragraphs.

3.1 Grammar Engineering

In [18], Klint, Lämmel and Verhoef argue that although grammars and related formalisms play a pervasive role in software systems, their engineering is insufficiently understood. They propose an agenda that is meant to promote research on Grammarware and state research challenges that need to be addressed in order to improve the development of grammars and dependent software.

One of these challenges is the development of a framework for grammar transformations and the co-evolution of grammar-dependent software. The Grammar Evolution Language proposed in this paper offers such grammar transformation operations and the automatic generation of compilers from DSL definitions with static validation of path expressions aims at the desired co-evolution of one important instance of grammar-dependent software, namely the compiler.

In [20], Lämmel proposes a comprehensive suite of grammar transformation operations for the incremental adaptation of context free grammars. The proposed operations are based on sound, formal preservation properties that allow to reason about the relationship between grammars before and after transformation. [22] and [19] present systems that implemented these evolution operations to incrementally transform LLL and SDF grammars.

Lämmel's grammar adaptation operations inspired the design of the Grammar Evolution Language used in our approach as a mean to automate language evolution. However, this paper focuses primarily on the coupled evolution of grammars and words of the language described by these grammars. Compared to the operations suggested by Lämmel, the Grammar Evolution Language sacrifices the formal basis to allow for simpler

coupled evolution operations. It would be desirable to combine the coupled evolution capabilities proposed in this paper with the formal preservation properties proposed by Lämmel in future versions of our tool Lever.

3.2 Document Structure Transformations

If the structure of a class of documents changes, all document instances have to be transformed to conform to the new structure. Document structure transformations aim at automating the transformation of document instances according to changes of the document structure which is similar to language and word evolution as described in requirement 3 above.

Document structure transformations for XML documents based on operators for DTD transformation and induced compensating transformations for documents instances are described in [21]. Su et al. propose a taxonomy for XML evolution operations. They suggest a complete, minimal and sound set of evolution primitives for DTDs and XML documents and show that they preserve validity and well-formedness of DTDs and XML document instances [36].

Both approaches perform coupled evolutions of XML schemes (in the form of DTSs) and documents and show completeness and soundness of the proposed primitives. However, both approaches are only complete in the sense that they allow arbitrary document structure transformations. They do not allow for arbitrary transformations of XML instance documents in order to compensate changes to the DTDs. Thus, information contained in documents may be lost as a consequence of DTD evolution.

The design of the language evolution operations in this paper refrains from achieving minimality in order to allow for lossless word transformations.

3.3 Evolution of Language Specifications

TransformGen is a system that generates converters that adapt programs according to changes of the language specification [33, 11]. While TransformGen automatically produces converters for local[3] changes, non-local transformations must be specified manually. Furthermore, non-local transformations cannot be reused between recurring evolution operations.

TransformGen only targets the adaptation of words but does not take language processing tools into account. The tool Lever presented in this paper goes one step further by semi-automating the adaptation of compilers, too. Moreover, Lever supports reuse of coupled transformations.

3.4 Object Oriented Language Specifications

In [12], Hedin describes an object-oriented notation for attribute grammars that heavily inspired the object oriented attribute grammar system implemented in Lever. Hedin et al. present further, more sophisticated object oriented attribute grammar formalisms in [13] and [8], which go beyond the attribute grammar system implemented in Lever. Future versions of Lever could incorporate inheritance between nonterminal grammar symbols and rewriting of syntax tree nodes, as proposed in [12] and [8].

[3]Local transformations are restricted to the boundary of a grammar production.

3.5 Schema Evolution in OO Databases

Regarding a data base schema as a language and the information contained in a data base as the words of this language allows to relate schema evolution with program transformation. Clearly, co-evolution of language and words is of predominant importance to this field and studied in various works.

In [2], Banerjee proposes a methodology for the development of schema evolution frameworks for object oriented databases (OODB) that was used in the ORION OODB system. The methodology suggests invariants for consistent database schemes and evolution primitives for incremental changes to the database. The evolution primitives perform coupled updates of both the schema and the objects in the database. Similar schema invariants and update primitives were proposed in [30] for GemStone OODB. The DSL Dictionary invariants that we use in our approach were inspired by these ideas.

4 Language Evolution Concept

The core concepts of the proposed approach to construct multi-level DSLs (see 4.1) incrementally are a grammar, word, and language evolution languages (4.3), and a generator architecture that is built around DSL histories (4.4).

4.1 Divide and Conquer

According to requirement 2 of section 2, layering DSLs is a key design principle for building powerful DSLs that map high level specifications to their implementation. Figure 2 illustrates the difference between one-step generation and the layering proposed in this paper.

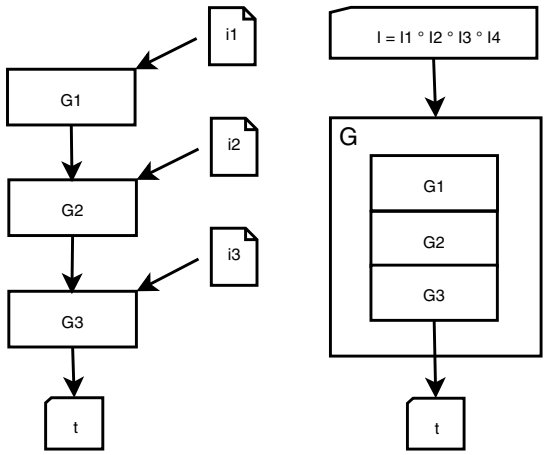

Figure 2. Staged versus one-step generation

On a theoretical level, staging the compilation process as shown on the left into three generators G_1, G_2, and G_3 that produce the output t in a sequence, seems identical with one-step generation of a composed generator G as shown on the right hand side; with the technical exception that inputs i_1, i_2 and i_3 are not fed into the generation process at once but at the beginning of each stage. In fact, the concatenation of the various inputs $i = i_1 \circ i_2 \circ i_3$ could be regarded as a word of the language

I that results from concatenating the input languages I_1, I_2, and I_3.

However, there are strong differences between the staged or the one-step generation model when it comes to the implementation of the DSL. Note, that not only the input fragments i_2 and i_3 depend on i_1 respectively $i_2 \circ i_1$ but also every language I_n depends on all inputs previous to stage n. Technically speaking, I_n corresponds with the information needed by G_n to further drive with the generation process in the situation created by $i_0, \ldots, i_{n_1}$. Now, specifying the unified language I of all possible input sequences would theoretically be possible but technically impractical. It would yield a undesirable DSL with numerous semantical conditions and exceptions allowing and restricting the use of language elements within a word of the language depending on arbitrary prefixes of the word.

In addition to the improved structuring of DSL, the staged model also indicates a feasible way of implementing complex generation process by dividing the task into separate steps with individual inputs at those points where it is needed. Though this might seem surprising for DSL design and implementation, this is exactly the strategy that system level software uses successfully to map high-level applications to system-level representations for execution. E. g. the C++ source code i_1 gets compiled with the C++ compiler G_1. The link-loader G_2 further sets the memory layout according to whether the user wants to execute the code as a stand-alone application or a shared library as specified in i_2. The operating system kernel G_3 then maps the results of these steps to main memory pages, CPU cycles, and so on according to the priorities of the user (i_3). Imagine the same process without staging. It would surely be possible but either hard to comprehend or less flexible.

Therefore, we clearly favor to divide and conquer the generation process.

4.2 Generator Transformation

Obviously, maintaining such a sequence of dependent generators is highly complex by itself and only practical with adequate tool-support. For example, if the top level DSL I_1 must be changed to accommodate a new feature, there is a high probability that the output of G_1 changes too, entailing the need for changing I_2, G_2, and so on. All of these changes of languages (I_n) and programs (G_n) can themselves be treated as language and program transformations. Hence, the evident tool to maintain a staged DSL is itself a DSL for the domain of DSL manipulation.

Figure 3 depicts this interrelation. A software architect or domain analysts uses the language evolution language DSL of the meta-level generator H to specify transformations of the DSLs $I_1, \ldots, I_n$. An application developer uses the resulting language and generator sequence $G1, \ldots, G_n$ to produce solutions on target platform t

The crucial element of this overall architecture is the top-level language evolution language and the meta-level generator H. Our tool called Lever[4] presented in section 5 implements significant parts of such a meta-level generator based on a grammar and word evolution input language.

[4]Language evolver.

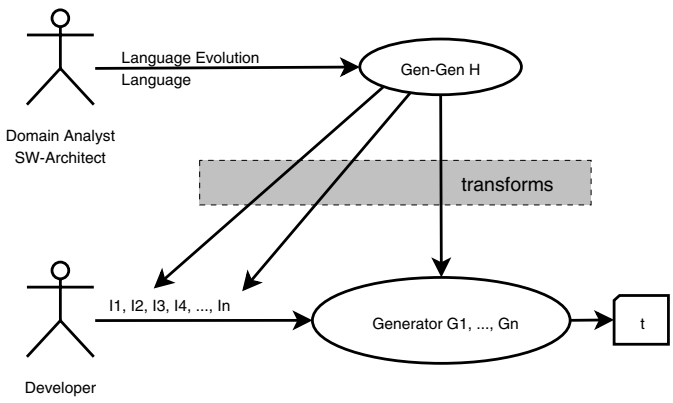

Figure 3. Generator-generator

4.3 Language Evolution Operations

The three proposed evolution languages for the manipulation of DSL specifications are displayed in Figure 4.

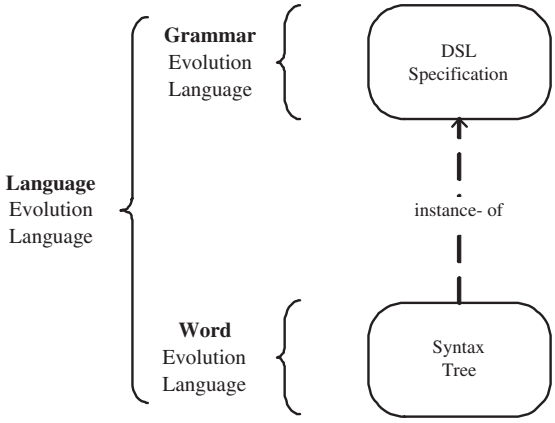

Figure 4. Evolution languages

Grammar Evolution Language (GEL) transforms the syntax and static and translational semantics of a DSL. GEL operations can be used for both creating the initial version as well as modifying it in order to yield subsequent versions of a DSL.

The GEL is complete in the sense that its statements can be used to transform any DSL syntax (and semantics) into any other DSL syntax (and semantics).

Word Evolution Language (WEL) statements work on the syntax trees of DSL words. During language evolution, they are used to perform syntax tree transformations to compensate changes of the underlying grammar.

WEL is complete in the sense that its statements can be used to transform any syntax tree into any other syntax tree and thus to compensate arbitrary changes to the DSL specification.

From the point of view of expressiveness, the combination of these two evolution languages allows the specification of all

possible transformations that might arise during the evolutionary development of a DSL.

However, from the point of view of usability, a third evolution language is desirable: the grammar and word evolution languages merely provide a low level of abstraction. Even simple coupled evolution operations, such as renaming a keyword in the syntax and all existing words, require at least two evolution operations – one from each language. Furthermore, coupled transformation knowledge cannot be reused to simplify recurring evolution operations. This gap is filled by the third evolution language.

Language Evolution Language (LEL) statements perform coupled evolution of both the grammar and the syntax tree. They provide a higher level of abstraction to users and enable reuse of coupled transformation knowledge. LEL builds on the GEL and WEL to implement its transformations.

LEL can be conceived as a procedure mechanism that uses GEL and WEL statements in the bodies of LEL procedures.

4.4 Evolution Architecture

Figure 5 shows the central components of the architecture of our language evolution tool Lever.

All evolution operations applied during the construction and evolution of a DSL are stored in the DSL History. The DSL History thus contains transformation information that specifies the delta between consecutive versions of a DSL. This transformation information is used to automatically adapt both the DSL compiler and existing DSL words to conform to the latest language version.

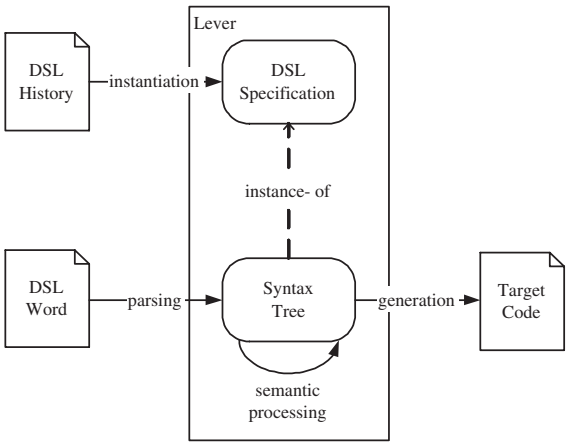

Figure 5. Lever architecture

DSL History contains evolution operations that define specifications for all versions of a DSL: The first evolution operations create the DSL specification for the initial version of a DSL. Subsequent evolution operations transform the DSL specification to yield later DSL versions.

DSL Specification is a comprehensive, declarative specification of the syntax and static and translational semantics of a single version of the DSL. It is explicitly available at runtime and drives the compilation process.

Syntax Tree is the in-memory representation of DSL words. It is an abstract syntax tree that is decorated with concrete syntax and semantic attributes.

DSL Word is the input for the compilation process. DSL words are versioned to allow the identification of the DSL version in which the word was written.

Target Code is the result of the compilation process.

4.4.1 Compiling Words of Arbitrary Versions

The information contained in the DSL history allows to translate DSL words written in any version of the DSL. During the compilation process, the following steps are performed:

1. Identification of the DSL word's language version.
2. Execution of the evolution operations from the DSL history in order to create a DSL specification in the corresponding language version.
3. Generation of a parser from the information in the DSL specification. The parser is then used to instantiate the syntax tree from the DSL word.
4. Transformation of DSL specification and syntax tree to the latest language version. Versions of the DSL dictionary and the syntax tree are compared with this latest DSL version. If needed the DSL history is used to transform both the DSL specification and the syntax tree to the latest version.
5. Semantic processing: according to the DSL semantics contained in the DSL specification, target code for the syntax tree is computed and written to the output.

5 Implementation: Lever

The proposed evolution operations and evolution architecture is implemented prototypically in our tool Lever.

5.1 DSL Specification Formalism

Lever uses an object oriented interpretation of attribute grammars[5] [28] as specification formalism for both syntax and semantics of a DSL. In Lever, DSL specifications are called *DSL Dictionaries*, since they define the syntax and semantics of every word a language comprises.

In DSL dictionaries, semantic rules specify how target code gets generated from the data contained in the syntax tree. In order to cleanly separate target code fragments, code generation logic and syntax tree access from one another, DSL dictionaries use code generation templates as semantic rules. Access to the syntax tree from within code generation templates runs via XPath [43] expressions.

Every access to the syntax tree from within a semantic rule introduces a dependency between the rule and the syntax tree.

[5] Context free grammars extended with semantic attributes and rules for their computation.

Language evolution operations may change the shape of the syntax tree and thus potentially break these dependencies. In order to support DSL architects, Lever can statically validate all XPath expressions against the DSL dictionary and thus detect broken dependencies during language evolution.

5.2 Evolution Operations in Lever

The Grammar Evolution Language (GEL) comprises statements to declare nonterminals, to create, rename and delete productions, to add, modify and remove (literal, terminal or nonterminal) production components and (inherited or synthesized) attribute declarations, to set semantic rules, to change the order of production components and to influence priorities and associativity of productions. Every GEL statement operates on a single DSL Dictionary element.

The Word Evolution Language (WEL) has been inspired by XUpdate [39], a language for updating XML documents. It comprises statements that use XPath expressions to select, insert, update and remove nodes from the syntax tree. Furthermore, it contains statements to declaratively construct syntax tree fragments and change the dictionary element a syntax tree node instantiates.

The Language Evolution Language (LEL) comprises statements for recurring coupled evolution operations, such as the introduction or removal of literal or terminal symbols, the encapsulation or in-lining of production components or the renaming of productions or literals (i.e. keywords).

Example

Figure 6 depicts a DSL dictionary for a simple expression language that translates sums to stack machine code, and a syntax tree for the expression *1+2*. The corresponding syntax tree is shown in 7. Some of the GEL statements that create the DSL dictionary are shown in listing 1

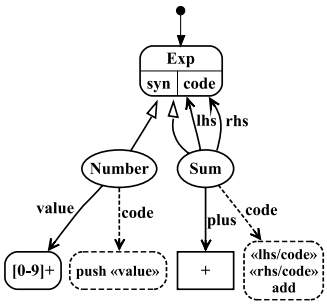

Figure 6. DSL dictionary in infix notation

Listing 2 shows the evolution operations that transform the expression language from infix to a postfix notation that surrounds every subexpression with angular brackets. (The expression *1+2* gets transformed to *<1 2 +>*.) It shows the higher level of abstraction that LEL statements provide over the use of separate GEL and WEL statements: lines 32-36 insert the opening < bracket into every sum. Line 38 performs the same operation for the closing > bracket, using only a single LEL statement.

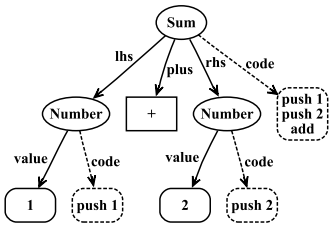

Figure 7. Syntax tree in infix notation

```
17 # Create prod. Sum: "lhs:Exp '+' rhs:Exp ->
       Exp"
18 g_create_production("Sum", "Exp");
19 g_append_nonterminal("lhs", "Exp");
20 g_append_literal("plus", "+");
21 g_append_nonterminal("rhs", "Exp");
22 g_set_associativity("left");
23 g_set_semantic_rule("code", "...");
```

Listing 1. Simple GEL statement example

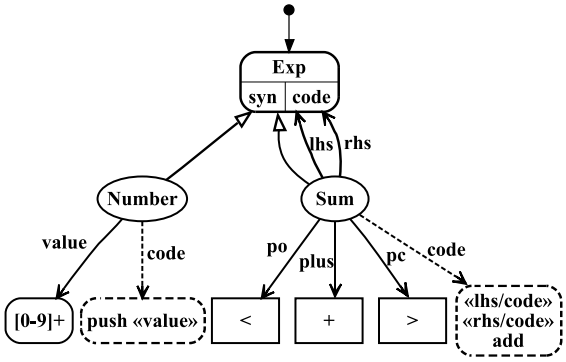

Figure 8. DSL dictionary after transformation to postfix notation

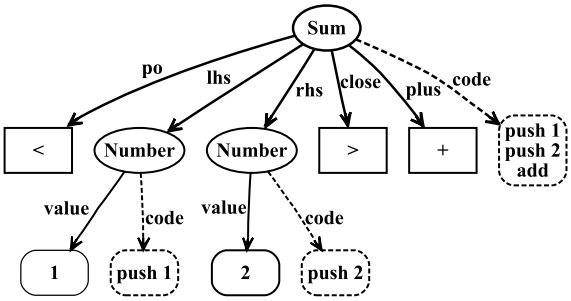

Figure 9. Syntax tree after transformation to postfix notation

```
29 g_move_symbol_before("rhs", "plus", "Sum");

31 # Insert "<" - GEL and WEL statements
32 g_append_literal("po", "<", prod="Sum");
33 g_move_symbol_before("po", "lhs", "Sum");
34 open = construct("<", "Sum.po");
35 insert("//.[DictionaryElem='Sum']", "po", open)
       ;

37 # Insert ">" - LEL statement
38 l_insert_lit_behind("close", ">", "rhs", "Sum")
       ;
```

Listing 2. Transformation from infix to postfix

5.3 Technical Implementation

Lever is implemented in Java. Figure 10 shows its components.

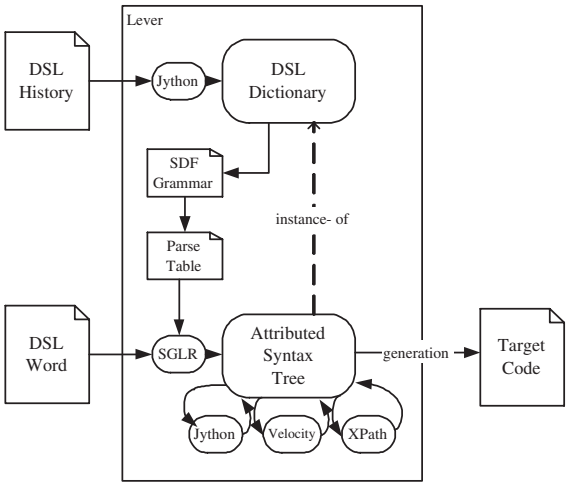

Figure 10. Lever implementation

Evolution Operations are implemented as a set of *Jython* procedures[6]. This way, the Jython interpreter can be used to execute the evolution operations from the DSL history. [7]

Parsing is done by the *Scannerless Generalized LR* parser (SGLR) [5], which recognizes the entire class of context free languages. This allows concrete and abstract syntax to be integrated into a single grammar, since no grammar modifications are necessary in order to make the grammar recognizable by conventional parsers using LL or LR parsing techniques. The SDF grammar that drives the SGLR parser is generated automatically from DSL dictionaries.

[6]Jython [16] is a 100% Java implementation of the Python [32] scripting language with a very tight integration into Java.

[7]Note that the evolution languages themselves can be conceived as domain specific languages for DSL construction, and as such profit from a bottom-up development using Lever. In the current version, this approach has not been taken in order to avoid the bootstrapping problem, which arises when Lever is used to develop a part of itself. We intend to develop them using Lever in the future, though.

Semantic Processing uses a dynamic attribute evaluation algorithm [15] in order to compute the semantic attributes that contain the target code. Semantic rules for static semantics are written in Jython. Semantic rules for translational semantics use the *Velocity Template Language*. Lever uses the *Velocity Template Engine* [37], the *Jython interpreter* [16] and the *JXPath* component from the Jakarta Apache project [38] to evaluate semantic rules.

Visualization uses *dot* from the graph drawing package *GraphViz* [1] to generate visual representations of DSL dictionaries and syntax trees to support developers during DSL development.

5.4 Limitations

The current version of Lever only automates the adaptation of the DSL compiler. Additional tools, such as a debugger, pretty printer or syntax aware editor still have to be maintained manually.

Furthermore, Lever currently only targets *textual* DSLs. However, it is our conviction, that the stated problems also hold for *visual* DSLs and we believe that the concepts this paper proposes can also be applied to them.

6 Case Study: Catalog Description Language

As a proof of concept, Lever was applied to develop a specification language for product catalog management systems in an evolutionary way. The results show the feasibility of the proposed approach to DSL development.

Due to space constraints, this case study only demonstrates language evolution in a single stage scenario. On a conceptual level, this can be justified, since the conceptual distance between the DSL and the target code framework is small enough to allow for generation of high-quality Java code. It should be evident that this simple example could be extended to a multi-level scenario by adding for example layout specifications along with an only partially compatible list of options for the desired GUI framework.

6.1 Domain

Product catalogs are collections of structured product documents. Each document belongs to a product family. Typically, all documents within a product family share the same structure, whereas different families have different document structures.

Catalog management systems are used to create, manage and publish product catalogs. This comprises the creation, manipulation and deletion of documents by users, and the persistence and export of catalog data to different media. Catalog management systems are data-centric. Thus, solution domain artifacts that implement editors, display forms, persistence and data export depend on the structure of the documents the implemented catalog comprises. Implementing each artifact by hand—for every single document structure contained in a catalog—is tedious, error prone and costly.

The goal of the *Catalog Description Language (CDL)* is to provide a declarative specification language for product catalogs, from which these structure-dependent artifacts can be gen-

erated. This increases the level of abstraction of catalog management system development, by using generation to replace stereotype implementation activities.

6.1.1 Target System

The Catalog management systems generated from CDL specifications comprise two types of code: generic framework code, which implements functionality common to all catalog management systems, and catalog specific code, which gets generated from CDL specifications.

As suggested by the Generation Gap pattern [42], inheritance is used to separate generic framework code (which resides in base classes) from catalog specific, generated code (which resides in generated subclasses).

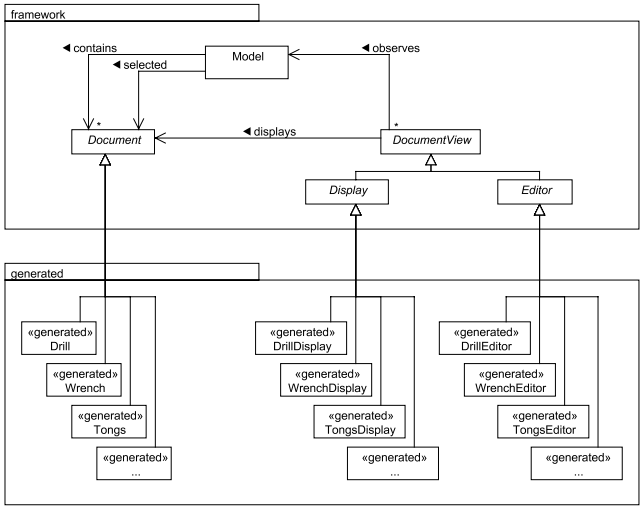

Figure 11. Framework architecture

The catalog management systems use a simple Model View Controller [10] architecture (compare Figure 11): A central *Model* stores all documents of a catalog. *DocumentViewers* (that serve both as viewers and controllers) are used to display and edit documents. Common functionality resides in the abstract base classes *Document*, *Display* and *Editor* in the *framework* package.

The document structure specific code resides in classes in the *generated* package, which derive from the abstract base classes. For each document family specified in a CDL document, a document class, a display class and an editor class are generated.[8]

Figure 12 shows the different conceptual layers of the catalog management system ordered by their level of abstraction. The higher an artifact appears in the figure, the higher its specialization and potential fitness to solve a domain problem and thus the lower its reusability to other problems in the domain.

6.1.2 Initial Language Version

Listing 3 shows an exemplary specification[9] for a tool catalog written in the initial version of CDL. The file specifies document structures for two product families: *Wrenches* consist of a

[8]The current version uses serialization as a generic persistence mechanism and does not support data export. In a future version, persistence and export code will also be generated from document structure specifications.

[9]simplified due to space constraints

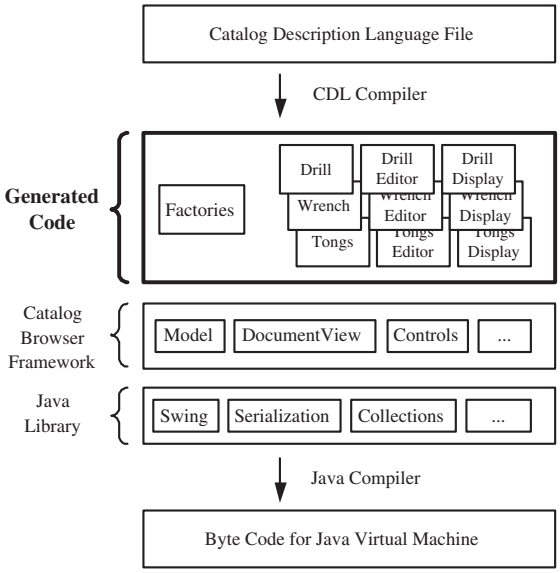

Figure 12. CDL stack

single multi-line text field *Description*, whereas *Drills* comprise one single-line text field *Headline* and two multi-line text fields *Description* and *Shipment*. The captions depict field labels displayed in editor forms.

Due to space constraints, the complete specification of CDL and the evolution operations applied during its evolution have been omitted. Refer to [*Removed due to anonymization*] for a complete reference of the implementation and evolution of CDL.

```
1  version 1
2  Wrench {
3    multiline  Descript   caption "Description"; }
4
5  Drill {
6    singleline Headline   caption "Family";
7    multiline  Descript   caption "Description";
8    multiline  Shipment   caption "Shipment Info"; }
```

Listing 3. CDL file in version 1

6.2 Evolving the Language

As is typical for incremental development, the first language version only comprises a small set of core language elements. Instead of designing the complete language up-front, we will grow it in small steps. This saves us from the effort and cost of performing a domain and variability analysis for our DSL. Furthermore, as we employ the first version of CDL to create catalog specifications, our understanding of the domain grows and we get feedback on our language design. Based on this feedback, we can make founded decisions on how to evolve the language.

In the following, we present two exemplary evolution steps: A relatively simple transformation that changes the concrete syntax, and a more complex transformation that restructures the language in a non-local way.

6.2.1 Local transformation

As a first change, we decide to make the concrete syntax of CDL more expressive, by adding the keywords *catalog*, *document* and *field* and encapsulating the documents of a catalog in curly braces. Since this change only affects the concrete syntax of our language and leaves its abstract syntax unchanged, no semantic rules have to be updated.

Listing 4 depicts the required evolution operations. Line 2 contains a Language Evolution Language statement that inserts the keyword *catalog* into the production *Cat* in the DSL Dictionary. *lbl* is the label of the new catalog keyword, *docs* is the label of the dictionary element before which the new keyword gets inserted. [10] The statements in lines 3-10 behave accordingly for the braces and remaining keywords.

These evolution statements offer a high level of abstraction to the DSL developer, since they transform both the DSL Dictionary and the syntax tree. Listing 5 shows the CDL file after transformation. The new keywords introduced by the evolution operations are depicted in bold font.

```
1  # Add catalog keyword and brackets
2  insert_lit_before("lbl", "catalog","docs","Cat"
       );
3  insert_lit_behind("open", "{", "label", "Cat");
4  insert_lit_behind("close", "}", "docs", "Cat");

6  #Add document keyword
7  insert_lit_before("label", "doc", "name", "Doc"
       );

9  #Add field keyword
10 insert_lit_behind("label", "fld","type","Field"
       );
```

Listing 4. Evolution operations for version 2

```
1  version 2
2  catalog {
3    document Wrench {
4      multiline  field Descript caption "Description"; }

6    document Drill {
7      singleline field Headline   caption "Family";
8      multiline  field Descript   caption "Description";
9      multiline  field Shipment   caption "Shipment Info"; }
10 }
```

Listing 5. CDL file in version 2: local change of concrete syntax

6.2.2 Non-local transformation

At this stage of development, we receive the requirement that a catalog management system must support users that speak different languages. As a consequence, catalog descriptions must be extended to support field labels in multiple languages. We decide to extract the field captions from the field definitions in order to preserve readability in the presence of many languages.

Listing 6 shows the CDL file after transformation. Lines 11-19 have been created by the evolution operations. Now that the labels have been extracted into a captions region, further captions regions can be added for additional languages.

This evolution scenario is an example for a non-local restructuring. It cannot be specified completely using high-level Language Evolution Language Statements alone. Rather, statements from the low level grammar and word evolution languages have been used to perform this evolution step. [11]

```
1  version 2
2  catalog {
3    document Wrench {
4      multiline  field Descript; }

6    document Drill {
7      singleline field Headline;
8      multiline  field Descript;
9      multiline  field Shipment; }

11   captions english {
12     Wrench {
13       Descript "Description"; }

15     Drill {
16       Headline "Family";
17       Descript "Description";
18       Shipment "Shipment Info"; }
19 }}
```

Listing 6. CDL file in version 3: non-local restructuring

7 Conclusion

DSLs are a promising approach to increase the productivity of software development through raising the level of abstraction and providing powerful generative techniques. However, DSLs are very expensive to build and even more expensive to maintain. The concepts and implementation techniques presented in this paper allow a new style of DSL development and maintenance by incremental step-wise evolution. This strategy renders the critical task of domain analysis less time-consuming and critical and greatly reduces the costs of changing a DSL by

1. automatically transforming all existing words in previous versions of the DSL and

2. providing tool-support for the adaptation of the DSL compiler.

As shown with the sample product catalog language, DSL architects are enabled to introduce flexibility into the DSL as needed at any time. The key to this flexibility is the transformation tool Lever (language evolver) that implements itself a powerful DSL for grammar, word, and coupled transformation for the consistent manipulation of DSLs.

As shown in this paper, a tool like Lever contributes to the construction of more powerful DSLs that span several levels of

[10] In Lever, every part of a DSL Dictionary is labeled—language evolution operations can thus refer to the DSL Dictionary elements they work on by their names.

[11] The evolution script comprises about 20 evolution operations and has been left out of this paper for brevity.

abstractions because this can only be done realistically by structuring the compilation process into a sequence of generation steps with a corresponding set of DSLs. Building and maintaining such a sequence of DSLs without tool supported and coupled transformation of grammars and words seems highly impractical.

Clearly, this work leaves room for interesting future work. One open question is how to further automate the adaption of the compiler and other language processing tools according to chances of the language. Another question that we will further investigate in the future is the actual construction of realistically applicable multi-level DSLs with the tool Level implemented as part of the work presented in this paper.

References

[1] ATT Research. Graphviz - Graph Vizualization Software. `http://www.graphviz.org/`, 2006.

[2] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD '87*, San Francisco, 1987. ACM Press.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[4] G. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, 2004.

[5] CWI. SGLR. `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/SGLR`, Jan. 2006.

[6] A. Deursen and P. Klint. Little languages: little maintenance? Technical report, Amsterdam, The Netherlands, 1997.

[7] U. W. Eisenecker and K. Czarnecki. *Generative Programmierung*. Addison-Wesley, München, 2002.

[8] T. Ekman and G. Hedin. Rewritable reference attributed grammars. volume 3086, pages 147–171, January 2004.

[9] Free Software Foundation. GNU Bison. `http://www.gnu.org/software/bison/`, Oct. 2005.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design patterns : elements of reusable object-oriented software*. Addison Wesley, 1995.

[11] D. Garlan, C. W. Krueger, and B. S. Lerner. Transformgen: automating the maintenance of structure-oriented environments. *ACM Trans. Program. Lang. Syst.*, 16(3), 1994.

[12] G. Hedin. An object-oriented notation for attribute grammars. In *ECOOP '89*, BCS Workshop Series, pages 329–345, Nottingham, U.K., 1989. Cambridge University Press.

[13] G. Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *WAGA'99*, pages 153–172, Amsterdam, The Netherlands, March 1999. INRIA rocquencourt.

[14] J. Inc. Hibernate – relational persistence for java and .net. World Wide Web, May 2006. http://www.hibernate.org/.

[15] F. Jalili. A general linear-time evaluator for attribute grammars. *SIGPLAN Not.*, 18(9):35–44, 1983.

[16] Jython Home Page. `http://www.jython.org/`, Jan. 2003.

[17] G. Klein. JFlex - The Fast Scanner Generator for Java. `http://jflex.de/`, July 2005.

[18] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[19] J. Kort and R. Lämmel. The grammar deployment kit - system demonstration. In *Proc. of the 2nd Workshop on Language Descriptions, Tools and Applications*. Elsevier Science, 2002.

[20] R. Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe 2001*, LNCS. Springer-Verlag, 2001.

[21] R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th Intern. Conf. on Reverse Engineering for Inform. Sys. (RETIS 2001)*, books@ocg.at. OCG, 2001.

[22] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In *LDTA-01*. Elsevier Science, Apr. 2001.

[23] M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Tursky. Metrics and laws of software evolution - the nineties view, 1997.

[24] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton. *Software Maitenance Management*. Addison Wesley, Reading, 1980.

[25] M. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Garmisch, Germany, Oct. 1968. NATO Science Committee.

[26] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. Technical Report SEN-E0517, CWI, Dec. 2005.

[27] S. E. of Philosophy. The frame problem. World Wide Web, 2004 Feb.

[28] J. Paakki. Attribute grammar paradigms a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[29] T. Panas, W. Löwe, and U. Aßmann. Towards the unified recovery architecture for reverse engineering. In *SERP'03*, volume 1, pages 854–860, Las Vegas, NV, June 2003. CSREA Press.

[30] D. J. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. In *OOPSLA '87*, pages 111–117, New York, NY, USA, 1987. ACM Press.

[31] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[32] Python Software Foundation. Python Programming Language. `http://www.python.org/`, 2006.

[33] B. J. Staudt, C. W. Krueger, and D. Garlan. A structural approach to the maintenance of structure-oriented environments. In *SDE 2*, Palo Alto, CA, 1987. ACM Press.

[34] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, Mar. 1997.

[35] R. Stützle. *Wiederverwendung ohne Mythos*. PhD thesis.

[36] H. Su, D. Kramer, L. Chen, K. T. Claypool, and E. A. Rundensteiner. XEM: Managing the evolution of XML documents. In *RIDE-DM*, pages 103–110, 2001.

[37] The Apache Software Foundation. Velocity. `http://jakarta.apache.org/velocity/`, 2005.

[38] The Apache Software Foundation. JXPath - JXPath Home. `http://jakarta.apache.org/commons/jxpath/index.html`, 2006.

[39] The XML:DB Initiative. XUpdate - Xml Update Language. `http://xmldb-org.sourceforge.net/xupdate/`, 2003.

[40] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[41] E. Visser. A survey of strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[42] J. Vlissides. Generation Gap. `http://www.research.ibm.com/designpatterns/pubs/gg.html`, Dec. 1996.

[43] W3C. XSL Transformations (XSLT). `http://www.w3.org/TR/xpath`, Nov. 1999.

[44] W3C. Web services description language (wsdl) 1.1. World Wide Web, Mar. 2001. http://www.w3.org/TR/wsdl.