Delft University of Technology
Software Engineering Research Group
Technical Report Series

# PIL: A Platform Independent Language for Retargetable DSLs

Z. Hemel, E. Visser

Report TUD-SERG-2009-025


TUD-SERG-2009-025

Published, produced and distributed by:

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

ISSN 1872-5392

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

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

This paper is a pre-print of:

Zef Hemel, Eelco Visser. PIL: A Platform Independent Language for Retargetable DSLs. In Mark G. J. van den Brand, Jeff Gray, editors, *Software Language Engineering, Second International Conference*, SLE 2009, Denver, USA, October, 2009. Lecture Notes in Computer Science, Springer, 2009.

```
@inproceedings{HemelVisser:2009,
  title = {{PIL}: A Platform Independent Language for Retargetable {DSLs}},
  author = {Zef Hemel and Eelco Visser},
  year = {2009},
  booktitle = {Software Language Engineering, Second International Conference,
               SLE 2009, Denver, USA, October, 2009. Revised Selected Papers},
  editor = {Mark van den Brand and Jeff Gray},
  series = {Lecture Notes in Computer Science},
  publisher = {Springer},
}
```

# PIL: A Platform Independent Language for Retargetable DSLs

Zef Hemel, Eelco Visser

Software Engineering Research Group, Delft University of Technology,
The Netherlands, Z.Hemel@tudelft.nl, visser@acm.org

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

## 1 Introduction

Intermediate languages have been used in compiler construction since the 1960s [24] to improve the retargetability of compilers. Rather than generating machine architecture specific instructions directly, compilers emit machine-independent instructions written in a low-level intermediate language, which is subsequently translated into machine-specific instructions by machine-specific compiler back-ends.

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

Generating source code instead of machine instructions poses a new retargetability challenge at the level of *software platforms*. A software platform consists of one or more programming languages with a set of libraries and frameworks, deployable on one or more operating systems. Dozens of software platforms compete and companies typically standardize on a single one (e.g. Sun's Java, Microsoft .NET or LAMP[1]). Consequently, DSL vendors have to choose whether to generate code for a single software platform, or multiple software platforms. Ideally, a DSL compiler targets many platforms, to maximize its potential customer base. Whereas encoding implementation knowledge of domain-specific concepts in a compiler enables the reuse of this knowledge between *applications*, there is little reuse between the different *back-ends* of such a compiler, due to language and library discrepancies between platforms. This lack of reuse causes significant maintenance problems. For instance, the ANTLR parser generator [19] has code generator back-ends for over a dozen platforms. However, many of them are not up-to-date with the latest ANTLR release. Similarly, WebDSL [30], a DSL for data-intensive web applications, has back-ends for Java and Python, but whenever a new feature is added to WebDSL, it needs to be implemented and maintained in each back-end individually, in practice leading to incompatible platform back-ends.

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

The lack of reuse between compiler back-ends could be circumvented by performing automatic language translation, e.g. translating generated Java code to Python, but this translation is expensive because of the complexity of the Java language and its libraries. Efforts to port dynamic languages, specifically Ruby and Python, to the CLR [1, 2] and JVM [3, 4] so that software written in these languages is portable to these platforms, are also very complex, often incomplete and have performance issues.

In this paper we introduce the intermediate language PIL, a Platform Independent Language providing a level of abstraction between DSL and software platforms, abstracting from discrepancies between platforms, thereby removing the need for platform-specific back-ends. In contrast to intermediate languages in traditional compiler construction, PIL operates on a higher level of abstraction and has a concrete syntax, based on a subset of Java, leveraging the productivity advantages of generating source code

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

over generating machine instructions. PIL is designed as a small intermediate language, capturing only essential object-oriented constructs and is therefore easier to port to multiple platforms than Java, for instance. In addition, the design of a language specifically targeted at code generators enables the development of code-generation specific language features. PIL/G, a thin layer of abstraction on top of PIL, provides some of such code generation such as partial classes, partial methods, identifier concatenation and expression blocks. In the future we also see opportunities to integrate DSL debugging support as part of PIL/G. We realized an implementation of the Java and Python back-ends of the WebDSL compiler using PIL, reducing the maintenance effort of these back-ends.

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

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

## 2 Code Generator Architecture

In this section we describe the general architecture of a code generator generating code for a *single* platform. We examine how to cater for *multiple* platforms in the next section. The initial single back-end generator architecture is depicted in Fig. 1. It is composed of two parts: the front-end, which parses, checks and desugars models described in the DSL, and the back-end, which generates code from the model. We first give a brief overview of the operation of the generator front-end, followed by a discussion of generator back-ends.

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

A set of model-to-model transformations, also known as *desugarings*, transform the AST to a simplified, core DSL model. Normalizing transformations perform model simplification, such as adding default values for omitted optional information. More complex transformations transform higher-level constructs to a reduced set of lower-level constructs. For example, in WebDSL, access control [14] and workflow [16] are implemented as abstractions on top of the user interface, data model and action sub-languages, implemented through a number of model-to-model transformations. The result of the front-end transformations is a fully checked, normalized model represented in a reduced set of *core DSL constructs*.

*The generator back-end* A generator back-end generates code from a core DSL model produced by the front-end. Intuitively, generating code that uses a high-level framework seems an attractive, productive option [23, 28]. However, in the long term, frameworks often become too restrictive when more control is required over the exact execution of the generated application [13]. Initially, the WebDSL compiler generated code for the JBoss Seam framework, a high-level Java framework utilizing Enterprise Java Beans (EJBs) for the business logic, Hibernate for models and JSF (Java Server Faces) for constructing views. As the WebDSL language evolved, JSF in particular, became too restrictive. The WebDSL view models no longer were a good match for JSF. Consequently, JSF and EJBs were replaced by plain Java servlets that contain `println` statements printing HTML code.

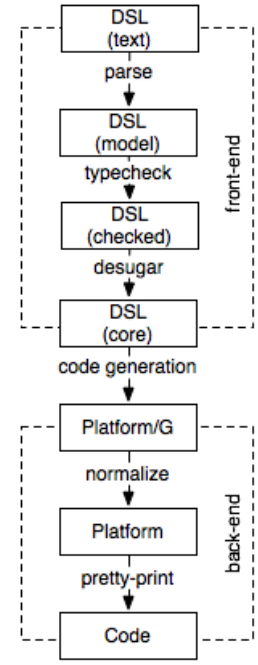


**Fig. 1.** Code generator architecture

There is mismatch in platform requirements between *developers* and *code generators*. Many modern software platforms (e.g. Java, .NET, Ruby, Python and PHP) are object-oriented at their core. Platforms typically try to differentiate by adding features on top of that core (Fig. 2), to improve the expressivity for *developers*, e.g. syntactic sugar and high-level frameworks. While improving developer productivity, these features limit flexibility, because they are only optimized for common use cases. By generating lower-level code, the execution of the generated application can be more effectively and flexibly controlled. In addition, because the object-oriented core of these platforms is similar, generating code at this level also significantly improves portability, which is discussed extensively in section 3. Although lower-level code is more verbose, the code is not intended to be read or modified, so this is not an issue.

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

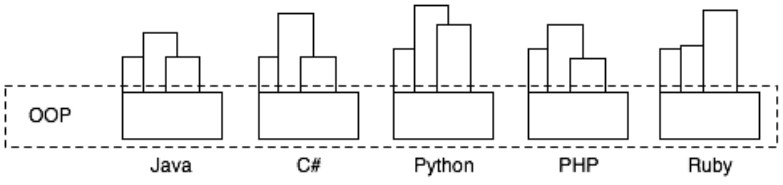


**Fig. 2.** Platforms and their features

and can be avoided by the use of code compositionality features, specifically partial classes and methods.

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

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

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

**Fig. 3.** Partial classes and methods added to Java

Partial classes are class fragments that are combined at compile time by merging their contents. Similarly, partial methods enable fragments of a method to be distributed over multiple partial classes. Partial methods are also merged at compile-time. Some languages support partial classes, e.g. Smalltalk, Objective-C, C# 2.0, Common LISP (CLOS) and Ruby, but many other languages do not support this feature, e.g. Java and PHP. Partial methods are less common. C# 3.0's partial method support is different than the partial methods just described; partial methods in C# are an optimization feature for providing hooks into generated code. Partial classes in C#, typically generated by a code generator, can declare the signature of a method as partial, meaning that if the method is implemented in another partial class with the same name, typically defined by a programmer, it operates as a regular method. However, if the method is not implemented in another partial class, all calls to the partial method are removed.

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

**Fig. 4.** A simple page definition in WebDSL

In previous work we presented the *code generation by model transformation* approach [15]. The key idea of this approach is to represent code as a model, enabling further transformation of generated code. Compositionality features such as partial classes and methods can be implemented by extending the target language. Fig. 3 shows an example of Java/G, Java extended with partial classes and methods, marked with `@Partial` annotations (in Fig. 1 the generalized form of this language is referred to as Platform/G). The compiler's transformation rules emit fragments of Java/G code, which are subsequently merged and written to files.

As an example of a code generation rule, we illustrate how Java code is generated from WebDSL page definitions using the Stratego/XT transformation toolset. WebDSL is a domain-specific language for data-intensive web applications [30]. It has sub-languages for the definition of data models, user interfaces, access control, workflow and business logic. The WebDSL page definition in Fig. 4 defines a page `blogEntry` with an argument of type `BlogEntry`. The view of the page defines a section, consisting of a header with the title of the blog entry and its content. Fig. 5 defines a Stratego/XT rewrite rule `page-to-java`, which transforms a WebDSL page to a Java class. When the left-hand side of the rule (before `->`) is matched its meta variables *x_page*, *farg**, *elem** are bound to their corresponding values. The right-hand side of the rule

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

**Fig. 5.** Rewrite rule that transforms pages to Java classes

defines the generated Java/G code. In the `where` condition, individual page elements are mapped to Java statements using the `elem-to-java` rule. The code pattern in the left and right-hand side of the rule use the *concrete syntax* [9] of the source and target languages, respectively. Code enclosed in `|[` and `]|` quotations is internally parsed by Stratego and turned into its AST representation. Consequently, the `page-to-java` rule matches an *AST representation* of a page definition and produces a Java/G AST, rather than textual code.

## 3 Retargeting a DSL Generator

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

### 3.1 Adding a Backend to a Generator

The most intuitive approach to support an additional target platform in a DSL implementation is copying an existing back-end and porting it to generate code for the new platform (Fig. 6). Generalizing this approach, supporting $N$ platforms requires $N$ back-end implementations. Fig. 7 shows how the `page-to-java` rule (Fig. 5) has been ported to generate Python code. A comparison of the Java and

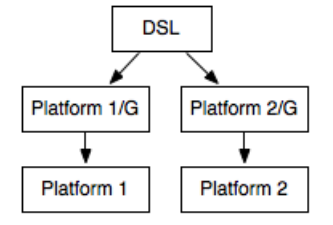


**Fig. 6.** Adding a backend to a generator

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

**Fig. 7.** A Python version of the `page-to-java` rules (Fig. 5)

Python back-ends suggest an additional advantage of generating low-level code: the syntax and low-level APIs do not differ that much between platforms. The main changes that have to be made to port a back-end are syntactic and relate to minor API differences. Although there is conceptual reuse between back-end generation rules, there is no code reuse between them, resulting in large-scale code duplication. In addition, code duplication also occurs in the reimplementation of code compositionality features for each back-end. Code duplication gives rise to maintenance problems. For instance, when the DSL is changed, modifications have to be propagated to all back-ends.

### 3.2 Language Translation

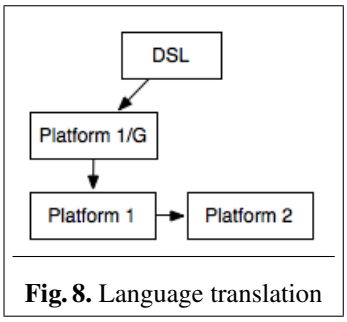


**Fig. 8.** Language translation

Efforts to translate dynamic languages, specifically Ruby and Python, to the CLR [1, 2] and JVM [4, 3] bytecode and Microsoft's *Java Language Conversion Assistant* to translate Java to C# code appear an attractive option to build retargetable software. Since only one transformation from the DSL to one of these platforms needs to be defined, this approach would solve the code duplication issue in generator back-ends. Fig. 8 depicts this scenario. Transformations that port code from one platform to another are reusable in multiple generators. In addition, code compositionality language extensions have to be implemented only once and need not be ported. However, language ports are problematic due to sheer language complexity, performance issues and the fact that these languages and their platform libraries are not designed to be portable across platforms. Consequently, this approach does not scale well.

### 3.3 High-Level Intermediate Languages

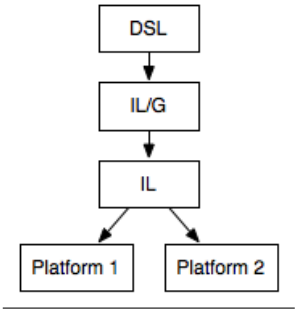


**Fig. 9.** High-level intermediate language

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

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

Typically, the most expensive transformation in a DSL compiler is the transformation from the DSL to target platform code. This transformation is expensive because of the large *semantic gap* between DSL and platform code. Thus, the number of times that this transformation needs to be implemented should be limited as much as possible. A well-known technique in compiler construction is the use of intermediate languages [24, 25]. By using an intermediate language, the maintenance of the compiler is much improved, since only one complex transformation from the DSL to the intermediate language needs to be implemented and maintained. Furthermore, the semantic gap between the intermediate language and the platform is very small, enabling implementations of the intermediate language for new platforms to be developed with little effort. Such intermediate language implementations are reusable in multiple DSL generators. Code compositionality features, as well as other features convenient for code generation can be implemented as an abstraction on top of the intermediate language, implemented as a transformation. The architecture of this scenario is depicted in Fig. 9.

### 3.4 Evaluation

Fig. 10 compares the costs of the three approaches to construct retargetable DSLs. As the transformation from the DSL to platform code is expensive, the first approach, where $N$ supported platforms require $N$ back-end implementations, is not a desirable solution. In the second approach, language translation, only one DSL to code transformation needs to be implemented. However, the language translation $C$, although reusable in multiple compilers, is very expensive to implement due to the high cost of implementing full language translations. Using an intermediate language requires

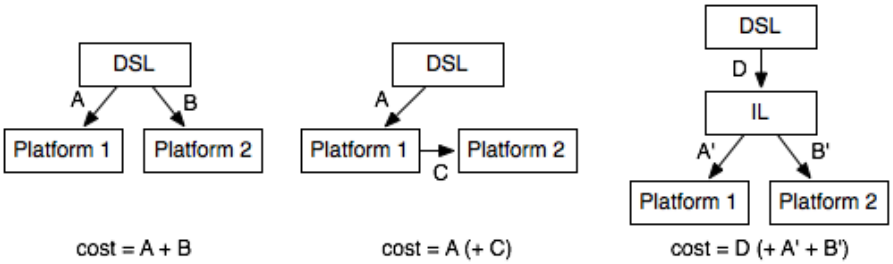


**Fig. 10.** The scenarios and costs of transformation options

only one transformation from the DSL to code written in the intermediate language. Implementing translations from the intermediate language to Platform 1 and 2 ($A'$ and $B'$) is cheap because of the small size of the intermediate language. In addition, these translations only need to be implemented once and are reusable in multiple compilers. Therefore, a future DSL compiler is only required to implement the transformation from the DSL to the intermediate language.

## 4 PIL: A Platform Independent Language

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

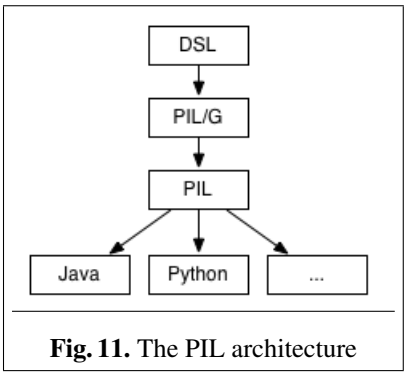


**Fig. 11.** The PIL architecture

By generating PIL code, rather than e.g. Java or Python code, the cost of targeting multiple platforms is greatly reduced. Any code generation toolset can be used to generate textual PIL code, which is subsequently translated to either Java or Python code by the PIL compiler (Fig. 11). PIL can also be linked to the generator directly as a library. Currently the PIL compiler library can be linked into Stratego/XT programs, but we are working to enable usage of the PIL library from other tools. The advantage of using PIL as a library is that the overhead of parsing, pretty-printing and I/O can be

eliminated by handing an AST to the PIL library rather than a textual representation of the PIL program. While the PIL compiler currently generates Java and Python code, more platforms can be added. Adding a new PIL target platform is cheap.

### 4.1 PIL: Object-Oriented Programming Essentials

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

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

From the Java language the following features are omitted:

- Visibility modifiers for classes, fields and methods (e.g. public, private, protected): information hiding features serve no purpose in generated code.
- Interface and abstract classes: can be replaced with classes with dummy implementations of interface methods.
- Inner and anonymous classes: can be implemented as regular classes.
- Imports: are syntactic sugar for the use of fully qualified class names.
- Checked exceptions: are not supported by most other platforms
- Distinction between primitive and object types: in PIL everything is an object. Nevertheless, the Java *implementation* of PIL does use primitive types and boxes and unboxes as required.
- Type coercion (e.g. from `int` to `long`): can be made explicit by a code generator.
- Enums: can be implemented using e.g. integers.
- Array syntax, e.g. `byte[] a` and `new byte[] { ... }`. In PIL an array is a regular generic type: `Array<Byte>` and can be instantiated with `new Array<Byte>(...)`.
- The one public class per file restriction. This feature is of no use the context of code generation.

The Java syntax had to be slightly adjusted to make the language context free. Java's `.` (dot) operator, which is used in package names, static member access, as well as instance member access requires type analysis to disambiguate. In PIL the dot operator is only used for instance member access. In the context of package names, PIL uses `::`. PIL has no static member support. Each language requires at least a minimal set of built-in data types, such as integers, strings, characters, arrays and maps. PIL implementations map each of these types to platform-specific implementations. Platform-specific APIs not part of the built-in data types can be accessed through *external class declarations*, which are further discussed in Section 5.

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

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

**Fig. 12.** Partial classes and methods

### 4.2 PIL/G: Compositionality of Code Generation

PIL is a small, easy to port language, but it lacks some features that greatly simplify code generation. In section 2 we discussed that most general purpose programming languages lack compositionality features such as partial classes and methods, which enable concise and modular code generation rules. PIL/G adds such compositionality features to PIL. Through a number of model-to-model transformations the PIL/G model is normalized to regular PIL and then mapped to platform code. In addition to partial classes and methods, PIL/G also adds identifier concatenation and expression blocks.

*Partial classes and methods* Partial classes and methods enable small code generation rules to emit pieces of code that together define a larger artefact. Fig. 12 shows two examples of PIL/G code that use partial classes and methods. Normalization of PIL/G to PIL results in a single `SomePage` class containing two fields (`inputCounter` and `pageTitle`) and one `init()` method in which `inputCounter` and `pageTitle` are set. The order in which partial methods' code bodies are concatenated is not defined.

*Identifier concatenation* A common pattern in transformations is composing two or more identifiers into one. For instance, generating getters and setters for a class property (Fig. 14) requires repetitive invocation of helper rules to render proper names for the getter and setter method. PIL/G adds a special # identifier concatenation operator to achieve the same result in a more concise manner, as demonstrated in Fig. 15. The operator adheres to the Java naming convention meaning that the concatenation of `get` and `name` results in `getName`.

```
var be : BlogEntry :=
  BlogEntry { blog := b }
```

⇓

```
BlogEntry be =
 {| BlogEntry e0 = new BlogEntry();
    e0.setBlog(b);
  | e0 |}
```

⇓

```
BlogEntry be = exprBlock0(b);
...
BlogEntry exprBlock0(Blog b) {
  BlogEntry e0 = new BlogEntry();
  e0.setBlog(b);
  return e0;
}
```

**Fig. 13.** Transformation from WebDSL entity constructor expressions to PIL implementation

*Expression blocks* DSL constructs typically have a higher expressivity than the target platform language that implements them. Therefore, it is common that an expression in the DSL requires multiple statements of implementation code. In WebDSL

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

**Fig. 14.** Transformation without identifier concatenation

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

**Fig. 15.** Transformation with identifier concatenation

this problem occurs while transforming entity constructor expressions to PIL expressions. Fig. 13 shows the transformation steps required to implement a simple example, in which an instance of the `BlogEntry` entity is created, initializing its `blog` property with blog `b`, assigning the result to a new variable `be`. The implementation of this example in PIL requires a variable declaration statement with an initialization expression derived from the entity constructor expression. In order to realize the entity construction, two PIL statements are required: one statement to create an instance of the `BlogEntry` class, and a second to set the `blog` property. Implementing such a transformation is complex, requiring statement lifting. To simplify this type of transformation, PIL/G provides expression block syntax [8]: `{| stat* | returnvalue |}`, as demonstrated in the second transformation step in Fig. 13. During the PIL/G to PIL normalization, expression blocks are lifted to separate methods, receiving closure variables as arguments.

### 4.3 Developing PIL Back-Ends

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

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

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

**Fig. 16.** Web request and response interface in PIL

```
package webdsl;

import javax.servlet.http.*;

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

**Fig. 17.** Part of Java wrapper of Request interface

# 5 PIL/Platform Interaction

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

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

## 5.1 API Wrapping

APIs such as I/O, threading and networking libraries are typically available and similar across platforms. For instance, the API to handle HTTP requests looks slightly different on each platform, but behaves the same. On each platform there is a method to retrieve a GET or POST parameter, get or set a cookie and get access to sessions. Thus, these APIs can be wrapped behind an interface, such as depicted in Fig. 16 which shows PIL `external class` declarations for a simple Web API. On the Java platform, this interface is implemented using the Java Servlet APIs (Fig. 17) and on the Python platform it is implemented wrapping its CGI module. The `external class` declaration as seen in Fig. 16 exposes these classes to PIL code. After the code is generated, the defined wrapper APIs and generated code are combined and compiled by the platform compiler, or interpreted by a platform interpreter.

### 5.2 Missing API on Some Platforms

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

Because we could not find a suitable object-relational mapper for Python that is compatible with Hibernate, and because Hibernate did not suit our needs entirely anyway, we implemented a simple ORM framework in PIL (Fig. 18). Although Hibernate's implementation is substantial, WebDSL only requires a fraction of its features. A significant part of Hibernate's implementation is dedicated to framework *usability*, such as its extensive configuration and annotation support. Such features are of little value when code is generated. Therefore, the ORM library we implemented in PIL, provides only the features and behavior that WebDSL requires. PIL makes the ORM framework very portable, because each platform only requires the wrapping of a low-level database API, enabling the execution of SQL queries (Fig. 19).

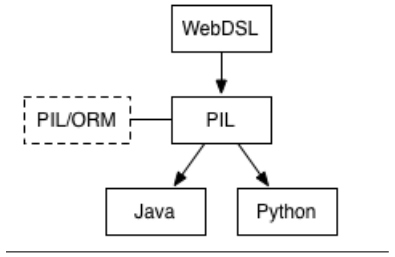


**Fig. 18.** ORM implemented in PIL.

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

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

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

**Fig. 19.** Low-level interface to database

### 5.3 Semantic Mismatches

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

If changing semantics in the wrapper is not feasible, adapting the framework could be an option. However, this requires a fork of the framework and the maintenance effort that comes with it. Another solution in this case is to reimplement the incompatible frameworks, either in PIL or in a platform-specific manner, as described in Section 5.2.

### 5.4 Platform-Specific Glue

Code generated from a DSL often does not implement the entire application. The canonical example of this are parser generators which only generate parsers that are subsequently invoked from code written specifically for the platform, or code generated from another DSL. Similarly, WebDSL generated code is not invoked directly either, but compiled in conjunction with web application glue code. WebDSL pages are translated to `Page` classes as illustrated in Fig. 5. Additional code is emitted that registers the page class in a global map during application initialization (Fig. 20). For each web application a singleton `WebApp` class is generated, whose `initPages` method is extended for each page. Glue code, specific for each platform, instantiates the generated `WebApp` class and retrieves the classes to instantiate based on the request. An example of such glue code for Java is shown in Fig. 21.

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

**Fig. 20.** Transformation that emits a partial function registering the page class

## 6 Discussion

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

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

**Fig. 21.** Instantiating PIL-generated Page objects from Java

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

### 6.1 Future Work

PIL has currently two platform back-ends: Java and Python. In the future we intend to add more, such as C#/.NET, PHP and Objective-C back-ends. We used PIL to implement back-ends for WebDSL, but in the future we intend to use it to implement back-ends for other DSLs as well. PIL has been developed using Stratego/XT, a DSL for program transformation, which we also want to port to other platforms. Currently there is a C and a partial Java back-end, PIL could greatly simplify maintaining such back-ends. We intend to also investigate if it is feasible to port the SGLR [29] parser implementation to PIL.

A problem with source code generation as implemented by many DSL compilers is that debugging is very difficult, because the structure and line numbers of the resulting source code are typically very different than the original DSL source. Techniques such as origin tracking [27] can be used to keep track of position information during transformations. Wu et al. [31] describe a technique in which position information mappings between source and target code are used by a wrapper around an already existing debugger for the target language. Another approach is by instrumenting generated code

that communicates with an external generic debugger. TIDE [26] takes this approach to simplify the process of defining debuggers for languages built using ASF+SDF. This approach also seems well-suited when implementing debugging support for multiple platforms, since a consistent debugging interface can be implemented for all platforms and no platform-supplied debugging support is required. Using the TIDE approach, code needs to be instrumented with `step` calls that send events to an external debugger with position information and the current environment. We see an opportunity for PIL/G to simplify adding debugging support in this manner, for instance by adding a `step` abstraction to the language that can easily be enabled or disabled.

### 6.2 Related Work

*Intermediate languages* Reusable intermediate languages for the purpose of retargetability are not a new idea. In compiler construction they appeared as early as 1960. UNCOL [24], the Universal Computer Oriented Language, was developed in response to the increasing number of programming languages required to target an increasing number of machine architectures. $M$ languages and $N$ machines require $M * N$ compilers, whereas with an UNCOL only $M + N$ generators and translators are required. Unfortunately, no truly universal UNCOL emerged to handle all languages and machines, likely due to the large size of the instruction set required to generate *efficient* machine code. Other proposed intermediate languages include BCPL's O-code [21], P-code [25] and `C--` [20]. `C--` is a more recent attempt to simplify machine code generation for multiple machine architectures. Rather than a byte-code representation of the intermediate language, `C--` has a concrete syntax similar to C. Similar to PIL, `C--` is designed as a code generation language. Its focus is much more low-level, however. It addresses a number of problematic areas of C, such as the lack of garbage collection and the difficulty of implementing tail calls. Retargetability is achieved by plugging in one or more machine code generation back-ends, e.g. gcc, VPO [6] or MLRISC [12]. The mentioned intermediate languages all operate at the abstraction level of machine architecture instructions. PIL by contrast is a much higher level language. It unifies object oriented programming languages and their platforms rather than machine architectures. What PIL adds on top of the mentioned approaches is *code generation specific features* such as partial classes, partial methods, identifier concatenation and expression blocks.

Union and intersection machines [11] represent fictional machines with features roughly equivalent to the *union* or the *intersection* of features offered by typical target machines. They are used as a basis from which intermediate language are derived. Union intermediate languages are good for generating efficient machine code, whereas the small size of intersection intermediate languages, such as PIL, are easier to port.

Basil is a high-level intermediate language with two use cases: (1) a target language for compilers for high-level languages and (2) a language to develop run-time libraries to be used by generated code [22]. PIL, too, is targeted at code generation and can be used to develop run-time libraries as well. Basil can translated to C. A subset of Basil, pure Basil, can be translated to LISP. In contrast to PIL, Basil is not an object oriented language and its purpose is not to simplify retargeting compilers. Instead, it is designed to make the semantic gap between source language and machine language smaller. In

addition, Basil allows its code to be annotated with position information, which can be used for debugging.

*ANTLR* Code generation back-ends for the ANTLR [19] parser generator are defined using StringTemplate [5], a template engine designed for code generation. For each supported platform (currently over a dozen) a number of code templates define the code to generate to implement ANTLR's features. As of version 3 of ANTLR, each back-end requires around 2800 lines of StringTemplate code. Whenever a new ANTLR version is released, the templates for each templates need to be adapted, resulting in a number of platform back-ends being out of sync with the current ANTLR version.

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

*The Model-Driven Architecture* OMG has defined the Model-Driven Architecture [17] in which platform-independent models (PIMs), through an *MDA mapping* are transformed to platform-specific models (PSMs). Whether this mapping should be performed manually or automatically is not specified. In our approach the WebDSL model is a PIM and the platform-specific code that is generated from that are PSMs. PIL acts as a thin layer in-between PIM and PSMs. The separation of platform-independent models from platform-specific models is essential when targeting multiple platforms. For instance, DSLs that contain escapes to the underlying platform are not platform-independent and can therefore not realistically be ported to other platforms. The fact that we implemented WebDSL back-ends for two different platforms (Java and Python), proves that WebDSL models are indeed platform-independent.

Bézivin et al. demonstrate how MDA can be used to automatically derive multiple PSMs from one PIM. They generate Java, web services and JWSDP from UML and EDOC PIM [7] models. Muller et al. [18] apply the MDA approach to generate web applications from visual UML-based models. Their code generator is written in Java and can generate either Java or PHP code that communicates with either an Oracle, MySQL or PostgreSQL database. The development of WebDSL so far has not focussed on supporting multiple database systems, but this is future research. We intend to investigate supporting not only SQL databases, but also alternative types of databases such as Google's BigTable [10]. Although PIL itself does not solve the problem of targeting different types of database systems, it can make the code to use these database systems portable across software platforms.

### 6.3 Conclusion

In this paper we explored a number of approaches to construct DSL compilers targeting multiple software platforms. The ability to retarget DSLs is enabled by the strict separation between platform-*independent* models and platform-*specific* models. It is common

practice to maintain separate compiler back-ends for each targeted platform. However, the maintenance of these back-ends is costly because DSL to target platform transformations are expensive to build and maintain. We argued that high-level intermediate languages can improve the retargetability of DSLs by only having to define one transformation from the DSL to the intermediate language. Subsequent mappings from the intermediate language to target platforms are cheap to develop and reusable in multiple DSL compilers.

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

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

*Acknowledgments* This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

## References

1. IronPython website. http://www.ironpython.com, 2009.
2. IronRuby website. http://www.ironruby.net, 2009.
3. JRuby website. http://jruby.codehaus.org, 2009.
4. Jython website. http://www.jython.org, 2009.
5. StringTemplate website. http://www.stringtemplate.org, 2009.
6. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. *SIGPLAN Not.*, 23(7):329–338, 1988.
7. J. Bezivin, S. Hammoudi, D. Lopes, and J. Jouault. Applying MDA approach for web service platform. In *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International*, pages 58–70, Washington, DC, USA, 2004. IEEE Computer Society.
8. M. Bravenboer, R. de Groot, and E. Visser. MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, volume 4143 of *LNCS*, pages 297–311, Braga, Portugal, 2006. Springer Verlag.
9. M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.
10. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
11. J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, 1984.

12. L. George. MLRISC: Customizable and reusable code generators. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1997.
13. D. M. Groenewegen, Z. Hemel, L. C. L. Kats, and E. Visser. When frameworks let you down. platform-imposed constraints on the design and evolution of domain-specific languages. In J. Gray et al., editors, *Domain Specific Modelling (DSM'08)*, pages 64–66, October 2008.
14. D. M. Groenewegen and E. Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In D. Schwabe and F. Curbera, editors, *Eighth International Conference on Web Engineering (ICWE 2008)*, pages 175–188. IEEE CS Press, July 2008.
15. Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *International Conference on Model Transformation (ICMT 2008)*, volume 5063 of *LNCS*, pages 183–198, Heidelberg, July 2008. Springer.
16. Z. Hemel, R. Verhaaf, and E. Visser. WebWorkFlow: An object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301 of *LNCS*, pages 113–127, Heidelberg, September 2008. Springer.
17. J. Miller and J. Mukerji. MDA guide version 1.0.1. 2003.
18. P.-A. Muller, P. Studer, and J. Bézivin. Platform independent web application modeling. In P. Stevens, J. Whittle, and G. Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 220–233. Springer, 2003.
19. T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.
20. S. Peyton Jones, N. Ramsey, and F. Reig. C–: A portable assembly language that supports garbage collection. In G. Nadathur, editor, *PPDP*, volume 1702 of *LNCS*, pages 1–28. Springer, 1999.
21. M. Richards. The portability of the BCPL compiler. *Software - Practice and Experience*, 1971.
22. L. Semenzato. The high-level intermediate language l. Technical Report UCB/CSD-93-760, EECS Department, University of California, Berkeley, Jul 1993.
23. T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
24. T. B. Steel, Jr. A first version of UNCOL. In *IRE-AIEE-ACM '61 (Western): Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 371–378, New York, NY, USA, 1961. ACM.
25. UCSD. *UCSD p-System and UCSD PASCAL Users Manual*. SofTech Microsystems, 1981.
26. M. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework - tool demonstration. *Electr. Notes Theor. Comput. Sci.*, 141(4):161–165, 2005.
27. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.
28. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.
29. E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.
30. E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *LNCS*, pages 291–373, Heidelberg, October 2008. Springer.
31. H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *Softw., Pract. Exper.*, 38(10):1073–1103, 2008.

TUD-SERG-2009-025
ISSN 1872-5392