Abstract

This paper describes the Simpl toolkit for implementing domain-specific languages (DSLs). Simpl aims to be a comprehensive toolkit that covers most of the aspects of developing a DSL. DSL implementations created with Simpl can be embedded into a larger system because no assumptions about architecture of the system are made. Simpl is based on existing tools and adds to them a grammar description language, automatic generation of abstract syntax tree classes, convenient API for creating language IDEs, and integration layer to tie together the base components.

Two tests were performed to evaluate the Simpl toolkit. Firstly, a fairly complex DSL, namely Waebric, was implemented and code metrics were compared to implementations based on other DSL tools. Simpl implementation contained the fewest lines of code and the code complexity was lower compared to the other tools. Secondly, subjects implemented a DSL using Simpl and using the ANTLR parser generator. The results showed that implementing the task with Simpl took less time than with ANTLR.

1 Introduction

Van Deursen et al. define a Domain-specific language (DSL) as "a programming language or an executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [15]. Classical examples of DSLs are Unix Makefiles (build scripts), regular expressions (specifying text patterns), HTML (describing text layout), and GraphViz (describing graphs).

Studies have shown that the use of DSLs in the software development process improves development productivity, maintainability, and understandability [15]. The main reason behind this is high abstraction level of a DSL programs and closeness of mapping between DSL concepts and domain concepts. However, the benefits of using a DSL must be balanced with the costs of creating and maintaining the DSL and its implementation. The decision to use a DSL depends on the domain and the size of the application. In this respect tool support for DSLs plays a critical role as it can reduce the implementation cost and therefore make the DSL-oriented approach usable in a wider range of projects.

When using a DSL-based approach for developing a larger system (for example, enterprise information system) one must also take into account restrictions imposed by the architecture of the larger system. Usually the DSL program is not the top-level program, but the system is built on a framework that dictates the flow of control and calls the domain-specific logic. Also, the system may contain more than one DSL, thus necessitating that the DSL implementations must be able to coexist or even to cooperate. The same considerations apply for DSL tools that can be embedded into build system or even into running system itself.

Freudenthal [4] identifies requirements that should be fulfilled by DSL tools that are intended to be embedded as part of a larger system. The focus is on DSL toolkits – collections of DSL tools that solve a range of tasks related to a DSL implementation, such as parsing, program checking, code generation and providing an IDE for the DSL. In practice, these requirements can be satisfied by following two guidelines. Firstly, a DSL toolkit should consist of two separate parts. One, "non-visual" part contains the core of the DSL implementation: parser, program checker, code generator, etc. that can be embedded into a larger system. The other, "visual" part contains (possibly integrated) environment for editing
and managing DSL programs. Secondly, the non-visual part of the DSL implementation must be easily embeddable into a larger system without making any assumptions on how the system is implemented. In particular, the non-visual part must not have dependencies to visual part and must not assume that the DSL implementation is a top-level program or function in the system.

The contributions of this research are the following: detailed description of a tool (Simpl) that addresses some key shortcomings of existing DSL tools, and a usability testing of this tool. In addition to comparing Simpl against a popular baseline (namely ANTLR [10]), this empirical evaluation provides rough estimates of how long does it take to learn a tool and to develop a moderately complex DSL.

The following section gives an overview of related work on DSL tools and usability evaluations of DSL tools. Section 3 introduces Simpl DSL toolkit. Sections 4 and 5 present two studies that evaluate usability of Simpl. The first experiment compared implementation of a benchmark DSL with implementations done using different DSL tools. The second experiment compared time spent on implementing a DSL using Simpl and using a comparison tool.

2 Related Work

DSL Tools

This section presents a brief overview of some existing DSL tools with respect to the requirements outlined in the previous section. A detailed evaluation can be found in [4], where DSL tools were divided into the following categories based on their architecture.

- Standalone tools (ANTLR, Stratego/Xt\(^1\), JastAdd\(^2\), various parser generators and attribute grammar tools/libraries) are usually a single-purpose. For example, ANTLR is a parser generator, JastAdd is an attribute grammar system. These can often be embedded into a larger system, but building a complete DSL implementation usually requires either considerable manual work or it requires one to integrate different tools.

- Eclipse/EMF-based MDE tools (Xtext, EMFText/JaMoPP, etc.) offer no clear separation between visual and non-visual parts. Typically, abstract representation of the DSL program is integrated into the IDE. DSL implementations created using these tools usually cannot be embedded into larger systems.

- Language workbenches (MPS\(^3\), Intentional workbench\(^4\)) have architecture where all the functionality of the tool is only accessible through the IDE. Note that this is design feature of current tools, not a fundamental necessity.

Based on this categorization, a DSL developer has generally two options. On the one hand, he can choose a DSL toolkit that covers most of the aspects of a DSL implementation but is mostly architected around an IDE toolkit and thus very difficult to integrate into a larger system. On the other hand, he can choose DSL tools that can be easily integrated into larger systems but that only cover one aspect of creating a DSL implementation (thus forcing the developer to manually integrate several incompatible tools). Simpl remedies this problem by offering a DSL toolkit that covers most aspects of a DSL implementation while using an architecture that allows embedding the created DSL implementation into a larger system.

---

\(^1\)See http://strategoxt.org/
\(^2\)See http://jastadd.org/
\(^3\)See http://www.jetbrains.com/mps/
\(^4\)See http://www.intentsoft.com/technology/overview.html
Usability Evaluations

Although there is no shortage of DSL tools, there are less studies on usability evaluations of DSL tools. This subsection presents few available examples of such evaluations.

There exist several qualitative evaluations of programming tools based on the cognitive dimensions framework [5]. For example, an evaluation of C# using this framework is given in [1]. However, this and the other similar evaluations are based on subjective assessments of the evaluator and do not involve experimenting with the tool.

Davies et al. [2] proposed the Technology Acceptance Model (TAM) that is meant to evaluate “how users come to accept and use a given technology.” In particular, TAM can be used to evaluate modeling and development tools. Evans et al. [3] present results of usability evaluation of a particular graphical DSML created using the TAM framework. Kärnä et al. [7] report on a controlled experiment to evaluate a DSM solution created using MetaCase DSM tool. They let the subjects implement a simple application manually and compared it with a manual implementation. They also report on time it took to build the DSM solution, thus indirectly evaluating usability of the DSM tool. Klint et al. [8] implement the same DSL using specialized DSL tools and also without tools (using general-purpose programming languages). They compare code metrics of the implementations and conclude that DSL tools offer better maintenance than manually implementing DSLs. The evaluation reported in this paper combines a controlled experiment together with a questionnaire designed based on the TAM framework.

3 Description of Simpl

3.1 Overall Architecture

The architecture of Simpl follows from the requirements discussed in the introductory section and the following design goals.

- Simple things must be simple. For example, when one has implemented a non-visual part of the DSL (parser), then one should have a basic IDE without writing any lines of code nor configuration files. Of course, in order to get more detailed and feature-rich IDE, one has to write code.

- Complex things must be possible. It must be possible to implement complicated DSLs with static type checking, non-trivial code generation, etc. The DSL processing code must be statically checked.

- The DSL toolkit should be built from existing tools in order to minimize the effort needed and also to minimize the amount of new tools that the user must learn. For example, creating and maintaining new Turing-complete programming language for DSL processing is not acceptable.

Simpl builds on ANTLR parser generator, Eclipse IDE platform, IDE Meta-Tooling Platform (IMP), and Scala programming language. The main rationale for selecting these particular tools is that they are mature, have good quality and are distributed under open source licences. Tools that make up the non-visual part of the DSL implementation have few dependencies and can be easily embedded (and can coexist with other DSL tools). From the integration point of view, the main restricting choice is using Eclipse as the IDE platform – if the DSL user wants to use IDE developed via Simpl, she must install Eclipse. In this case, we chose the most popular platform.

Figure 1 shows the main components of a DSL implementation created with Simpl. The first part is the non-visual language implementation that can be embedded into a bigger system. Development of a new DSL starts with grammar description that specifies both the context-free grammar of the DSL and...
Figure 1: Architecture of a DSL implemented with Simpl. Components with captions in italic are automatically generated.

the classes for representing the abstract syntax tree (AST) of a DSL program. The Simpl parser generator takes the grammar description as input and produces a parser and the AST classes. The (optional) program transformation component takes as input an AST and checks or transforms it. The code generator converts the preprocessed AST to text. The second part of the DSL implementation is the language IDE. It builds on Simpl IDE framework and the non-visual part of the language implementation.

3.2 Grammar Description

Simpl provides a language for describing grammar of the DSL programs. Unlike many parser generators, Simpl does not focus on grammar actions. Instead, the focus is on describing the abstract syntax of the DSL program. Figure 2a shows a grammar rule that parses if statement in an imperative language. The if statement can contain several elsif parts (indicated by “*”) and one else part (optionality is denoted by “?”). The grammar description is used to generate both the parser and the Scala case classes representing the abstract syntax of the DSL program. In general, Simpl generates a Scala case class for every grammar rule. In addition, the DSL developer can shape the class model by annotating the grammar description. In the example rule, identifiers before the equals sign (cond, ifStmt and elseStmt) are used to name the attributes that will store AST node corresponding to this rule. If a rule contains multiple sub-rule calls with the same attribute name then the corresponding attribute will contain a list of nodes. Figure 2b shows the Scala case class corresponding to the example rule. The attributes cond and ifStmt are typed as List because they refer to parts of the rule that can repeat. The AST classes can be processed either via pattern matching or by using object-oriented attribute access.

IfStatement: case class IfStatement(
    "IF" cond=Expression "THEN" ifStmt=StatementSequence,
    "ELSIF" cond=Expression "THEN" elseStmt=StatementSequence)*
    "ELSE" elseStmt=StatementSequence)?
    "END";

Figure 2: Simple grammar rule (a); and corresponding automatically generated AST class (b).
In addition to the names of the attributes, the grammar can be annotated with return types and return expressions, further modifying the AST. It is possible to specify a return type of a rule to make the rule return more general type than the case class named after the rule. For example, rule MulExpr can be made to return Expression. This has the additional effect that all the attributes referring to MulExpr rule will also have the type Expression.

The return type is useful in combination with the return expression. The return expression is a Scala expression that is called at runtime to modify an AST node that is returned by a rule. Figure 3 shows two example rules that use return types and return expressions expressions (highlighted with italic and bold font, respectively). In the first example, the ParenExpr rule returns the expression between parentheses without wrapping it in ParenExpr class. Without using return expression, the input “((<expr>)” would result in AST ParenExpr(<expr>). The second example shows how the return expressions can be used to parse binary operators. Here SimpleExpr has higher priority than MulExpr (the operands of MulExpr are parsed by SimpleExpr rule). This results in cumbersome AST if the default AST generation algorithm is used. For example, rule MulExpr applied to input “10” would output the object MulExpr(NumberLit(10), null). However, the return expression detects this degenerate case of MulExpr and returns simply NumberLit(10).

ParenExpr
returns Expression {expr}
  : "(" expr=Expression ")";

MulExpr
returns Expression {
  if (right eq null) left else _self
  : left=SimpleExpr("*" right=SimpleExpr)?
};

Figure 3: Examples of return expressions.

3.3 DSL Processing

Simpl relies on Scala programming language for expressing program transformations. This makes it possible to express transformation rules in a declarative manner and to combine elementary transformations into compound transformations. At present, Simpl does not provide direct support for specific tasks such as name resolution or type checking. These can be implemented directly in Scala or via the help of an attribute grammar library, such as Kiama [14].

Simpl provides two means for code generation. For simpler generation tasks, the developer can use the bundled StringTemplate [11] templating engine. Simpl provides glue that allows accessing AST nodes from the templates. For better control over the output, Simpl includes a pretty-printing combinator library, based on Philip Wadler’s Haskell library [16]. Using this library the developer can produce high-quality text output of complex and highly structured AST.

3.4 IDE Framework

Simpl contains IDE framework that is based on Eclipse IMP. It allows the language designer to create an Eclipse-based DSL IDE that provides typical IDE services, such as syntax highlighting, outline view, code folding, hyperlinking, occurrence marking, and automatic completion. The Simpl framework aims to offer intuitive and high-level API to DSL developer so that the IDE services can be programmed without writing any XML configuration files and without knowing any Eclipse APIs.

5 _self is a predefined symbol and refers to the unmodified AST corresponding to the rule
All the IDE services have meaningful default behaviour. For example, syntax colouring service colors keywords and comments using information in grammar description. Thus, the DSL developer can produce working IDE without writing any code at all. The default IDE services can be customized if better control over IDE behaviour is required. Simpl aims to base several IDE features on the same service implemented by the developer. For instance, occurrence marking and hyperlinking features use information provided by a “linking” service that connects identifiers to their definitions.

4 Measuring Code Metrics

4.1 Description of Experiment

Klint et al. have compared six different implementations of a rather comprehensive DSL, namely Waebric [8]. Three implementations – so called “vanilla” – were developed from scratch in Java, JavaScript, and C#. The other three implementations were written in the same languages but with the help of respective DSL tools: ANTLR, OMeta, and “M”. By using both quantitative and qualitative analysis, it was shown that the DSL tools indeed do increase maintainability of DSL implementations. This was especially noticeable in terms of code size reduction of the DSL implementations compared to the vanilla versions.

To compare Simpl with the DSL tools presented in [8], we implemented Waebric DSL using the Simpl DSL tool. In order to verify that the Simpl implementation is correct, we applied the same 100-program testsuite that was also used to verify implementations in the original paper. Therefore, the Simpl implementation was functionally equivalent to the other implementations. The grammar was implemented in the Simpl grammar description language (see Section 3.2) and the code generator for was implemented using Scala’s built-in XML library. In the following discussion, the grammar and generator implementations will be referred to as parse and eval components, respectively. Quantitative analysis was based on metrics suite proposed by Kuipers and Visser [9]. It proposes to evaluate maintainability of software by measuring volume, structural complexity and duplication of the code. The same set of metrics was used in the original study.

The volume was measured in terms of non-comment lines of code (NCLOC) and number of units in the parse and eval components. For grammar, a non-terminal is considered to be one unit. For Scala code, unit is equated to a method. Unlike other implementations in [8], the Simpl implementation does not separate code for checking the validity of AST from the eval component. Also, miscellaneous supporting code such as main objects and constant value definitions are counted as part of the eval.

The structural complexity of the code was measured by determining the percentage of NCLOC in the units that have the cyclomatic complexity (CC) higher than 6. CC of the parse component was calculated by counting decision points of the non-terminal nodes as described in [12, 8]. Alternatives (|), optionals (?), and iteration closure operators (*, +) count as a decision point. Additionally, embedded Scala code (return expression) in a non-terminal node adds to the CC count if a decision point exists there. Cyvis was used to calculate CC of the eval component.

4.2 Results and Discussion

Table 1 presents the volume measurements of the implementations. Note that the Simpl does not use separate check component. Instead, the checking functionality is included in the eval component. It can be seen that numbers for the Simpl implementation are smaller than or similar to the best numbers of the other DSL implementations. Quantitatively, the Simpl implementation consists of 5 files, 102 units, and

6Since the original study did not provide concrete figures for code duplication, we also left out this metric.

7http://cyvis.sourceforge.net/
577 NCLOC. The latter is especially remarkable as it is over 40% smaller than that of ANTLR – the next implementation in terms of the smallest NCLOC.

Table 1: Volume metrics: number of files (#F), number of units (#U), and NCLOC (#N). The data for ANTLR, OMeta, and “M” is taken from [8]. The smallest totals for each category are underlined.

<table>
<thead>
<tr>
<th></th>
<th>Simpl</th>
<th>ANTLR</th>
<th>OMeta</th>
<th>“M”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#F</td>
<td>#U</td>
<td>#N</td>
<td>#F</td>
</tr>
<tr>
<td>Parse</td>
<td>2</td>
<td>54</td>
<td>151</td>
<td>1</td>
</tr>
<tr>
<td>AST</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>22</td>
</tr>
<tr>
<td>Check</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>Eval</td>
<td>3</td>
<td>48</td>
<td>426</td>
<td>1</td>
</tr>
<tr>
<td>Misc</td>
<td>2</td>
<td>3</td>
<td>186</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>102</td>
<td>577</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2 shows the percentage of NCLOC in units with CC > 6. The parse and eval components are shown separately. As there was no eval component in the OMeta implementation, the check component data was used instead (see [8]). It is evident that the Simpl implementation of Waebric results in the simplest parse component. The complexity of the eval is similar to that of ANTLR and “M” (because OMeta implementation only includes check component, it cannot be directly compared with the other implementations).

Table 2: Percentage of NCLOC in units with cyclomatic complexity greater than 6.

<table>
<thead>
<tr>
<th></th>
<th>Simpl</th>
<th>ANTLR</th>
<th>OMeta</th>
<th>“M”</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse</td>
<td>6.6</td>
<td>20</td>
<td>26</td>
<td>10</td>
</tr>
<tr>
<td>eval</td>
<td>25</td>
<td>29</td>
<td>10</td>
<td>33</td>
</tr>
</tbody>
</table>

Overall, compared to the other implementations, the Simpl implementation of Waebric results in comparable and in some aspects simpler and shorter code. Part of it can be attributed to the simple and compact grammar description language of Simpl. The low volume and rather low complexity of the eval component can be attributed to the automatically generated class model for representing the AST and the compactness of Scala language.

5 Measuring Usability

5.1 Description of Experiment

Overview

Empirical evaluation of the Simpl toolkit was based on a controlled usability study. The study involves subjects with two levels of expertise: junior developers and senior developers, in order to determine if the level of experience affects the usability of the tool. Subjects were split into the “control group” and the “experimental group”. The control group used “standard” DSL tool, and the experimental group used Simpl – the aim being to compare the added value of Simpl with respect to the baseline. We chose the ANTLR as the tool for control group because it most closely matches the non-functional requirements described in the introduction and had strong positive impact on code size in the Waebric experiment [8]. ANTLR has similar functionality to Simpl – it includes parser generator, program transformation engine.
(tree rewriting) and code generator (StringTemplate). The main difference is that ANTLR does not include IDE generator.

The subjects were given a task of implementing a realistic, non-trivial DSL. We measured the time spent and compared the results of experimental group with the control group. In addition, after completing the task, the subjects were asked to complete a questionnaire for evaluating usability of the tools.

**The Subjects**

The subjects for this experiment were five professional programmers working at Cybernetica AS. Three subjects were junior programmers with bachelor degrees and some programming experience. Two subjects were senior programmers with master degrees and experience in using language tools and implementing programming languages. The level of programming skill was roughly consistent between subjects in the same group. The junior programmers had no previous experience in language processing tools. None of the subjects had any previous experience with Simpl or Scala. The subjects did not receive training for use of the tools. Instead, they used user manuals and other online materials.

Because the task was relatively labour-intensive, it could not be completed in a controlled lab environment. Instead, we opted for setting where the test subjects completed the task unsupervised on their normal workplaces (i.e., using workstations they used for doing everyday work). The subjects were responsible for logging the time spent on various subtasks. Although this setting provides less accurate measurements than the controlled lab setting, it is more realistic because real DSLs are not implemented over a couple of hours in a specifically set up computer. The task included installing and learning the appropriate tools (Simpl or ANTLR).

**The Task**

All the subjects performed the same task, namely, implemented a DSL for expressing spam detection rules (“SpamDetector”), similar to DSL used by SpamAssassin tool. The SpamDetector DSL allows the user to specify conditions that are matched against incoming e-mail messages. If the conditions match, the “spamness” score of the message is incremented or decremented accordingly. The DSL was moderately complex, containing arithmetic expressions and rudimentary user-defined functions.

Figure 4 shows an example SpamDetector program containing two rules. The first rule checks if Subject header of incoming message contains case-insensitive string “viagra” and if it does, increases the message’s “spamness” score by 2.0. The second rule uses named condition from_mydomain. It matches all the messages that contain (case-insensitive) string “business proposal” in their subject line and is not sent from a “good” domain.

Since the goal was to compare DSL tools, the subjects needed to implement only the DSL-related parts of the SpamDetector system. In particular, the subjects were required to implement the following components: (a) parser for the SpamDetector language; (b) program checker that detects calls to undefined conditions; (c) code generator that converts rule files to Java; (d) basic program editor/IDE that supports syntax colouring, syntax checking, outline view, and hyperlinking (navigating from reference to definition of condition). The subjects using ANTLR were free to choose means for building the editor. In practice, they adapted an example program demonstrating the creation of basic IDE for the Java programming language.

---

8This is subjective evaluation by project manager of the subjects
9In this matter, ANTLR has advantage over Simpl because of large body of online tutorials and examples.
11The subjects were not required to implement the runtime code responsible for parsing E-mail messages, extracting message header fields, normalizing space in strings, dealing with various encodings etc. Instead, the subjects could make assumption that there exists some kind of message-processing API and generate Java code that calls this API.
After completing the implementation task, the subjects were presented a questionnaire that asked their opinions of the DSL tool they used. The questionnaire for evaluating DSL tools was based on TAM. Since we did not have enough subjects to develop and verify our own scales, we used the questions and scales constructed by Recker [13] (the scales are based on TAM). The questions were used almost verbatim, the phrase “business process modelling” was replaced with “ANTLR/Simpl DSL tool”, depending on whether the subject used ANTLR or Simpl for completing the assignment. Users rated the questions on a scale of 1 to 5 where 1 designated “disagree completely”, 2 “somewhat disagree”, 3 “neutral”, 4 “somewhat agree”, and 5 “agree completely”. The questionnaire measured the following aspects of usability:

- Perceived Usefulness (PU) – the degree to which a person believes that using a particular tool would enhance his or her job performance;
- Perceived Ease of Use (PEOU) – the degree to which a person believes that using a particular tool would be free of effort;
- Confirmation (Con) – the extent to which a person’s initial expectations were confirmed;
- Satisfaction (Sat) – the extent to which a person is satisfied after adopting the particular tool; and
- Intention to Use (ItU) – the extent to which a person intends to use or continue to use a particular tool.

### 5.2 Experiment Results

Table 3 shows the amount of time the subjects spent on implementing various subtasks. Subtask “bug fixes” contains general activities not related to any specific subtask (such as final testing and fixing of the language implementation).

It must be noted that the “setup and learning” task does not accurately reflect the time spent on learning the tools. It rather indicates the time spent on the installation and studying the existing examples. According to the feedback, most of the learning time is included in the actual tasks – the subjects referred to the user manual only when they encountered some particular problem. Additionally, Simpl users also spent some time learning the Scala programming language.

The authors recognize that there were too few subjects in the study to draw any definite conclusions, however, based on the numbers in Table 3, some results should be highlighted. Firstly, the Simpl users were consistently faster than ANTLR users. For instance, the Simpl junior subjects were able to implement the problem significantly faster than the senior ANTLR user. Much of the success of Simpl can be
Table 3: Implementation times (in hours).

<table>
<thead>
<tr>
<th>Area</th>
<th>Junior developers</th>
<th>Senior developers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>J1</td>
<td>J2</td>
</tr>
<tr>
<td>Setup and learning</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Parser</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>Code generator</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Editor</td>
<td>#</td>
<td>45</td>
</tr>
<tr>
<td>Bug fixes</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>36</strong></td>
<td><strong>81</strong></td>
</tr>
</tbody>
</table>

* Subject J1 did not record the time spent on the editor and parser separately.
* Subject S2 did not actually complete the editor because of insufficient time available. After spending 16 hours on editor, he estimated the remaining amount of work to be 40 hours.

attributed to the editor – building an editor in Simpl is a relatively easy task for a person with even beginner level experience in Java or Scala. Secondly, Simpl also offers a good usability for creating non-visual parts of a language implementation. Table 3 shows that the time spent on writing the parser and the code generator are somewhat smaller for subjects who used Simpl.

Table 4 shows answers to questionnaire completed after finishing the study. Notably, the average Simpl user rates all the aspects of using Simpl equal or above 4.0 in the scale of 1 to 5 whereas the average ANTLR user rates most aspects close to 3. Given the presented results, it is evident that Simpl can be efficiently used to develop a full DSL, including parser, code generator, and editor and it might have certain advantages over ANTLR.

Table 4: Answers to the questionnaire.

<table>
<thead>
<tr>
<th>Area</th>
<th>Simpl</th>
<th>ANTLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceived Usefulness</td>
<td>4.7</td>
<td>4.0</td>
</tr>
<tr>
<td>Perceived Ease of Use</td>
<td>4.1</td>
<td>3.2</td>
</tr>
<tr>
<td>Confirmation</td>
<td>4.4</td>
<td>3.3</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>4.0</td>
<td>3.3</td>
</tr>
<tr>
<td>Intention to Use</td>
<td>4.3</td>
<td>3.0</td>
</tr>
</tbody>
</table>

5.3 Experiment Validity

Next, we discuss threats to validity. The metrics experiment (Section 4) is essentially a continuation of a previous study, we refer the reader to the validity section of the respective paper [8]. This subsection focuses on validity of the usability study (Section 5).

Construct validity determines whether the measured values really correspond to usability and usefulness. For the SpamDetector DSL we measured time spent on learning Simpl and implementing a realistic DSL. This correlates well with tool’s usefulness in practical world as the point of using DSL tool is to save time (and hence money) when implementing DSLs. As for questionnaire, the proposed questions are based on TAM model that has been validated by numerous researchers (see [13] for literature review).

Internal validity determines whether our conclusion is valid with respect to the measured data. Although the test subjects were chosen carefully, their skill level was determined mainly based on number
of years of experience. Thus, it is possible that some of the programmers in the group were weaker in some aspects compared to the others. However, as noted earlier, rather than trying to draw profound conclusions from the results we are more interested in overall usability of Simpl compared to the other DSL tool.

**External validity** determines whether it is possible to generalize the results to a larger population. The current study was conducted in a setting very similar to commercial software development. The task was fairly long (in the range of a man-week), the subjects implemented the task at their normal work environment during the normal working hours, and the subjects had freedom to make some of the design decisions (e.g. design API for the message processing). Also, all the subjects were professional programmers. The fact that they all worked for the same company, might have somewhat influenced the results. On the other hand, the target group of Simpl is mainly skilled programmers, thus the subjects fit within the scope of the study.

In addition to the discussed validity threats, the small number of participants must be taken into account. This was mainly caused by the fact that the requirements for the subjects skills were quite high. Also, as the task was time-consuming, it would have been too costly and difficult to find senior (professional) developers who would be willing to learn Simpl and spend non-negligible amount of time in this controlled study. At the same time, research by Hwang and Salvendy [6] indicates that 5-10 participants is enough to identify major usability issues.

We acknowledge that given the small number of subjects, the conclusions of the study do not have statistical significance. However, the study has provided us some valuable information about the relative strengths and weaknesses of the tools. Furthermore, it gave an indication about the time it takes for a programmer to learn the tool and develop a real DSL.

### 6 Conclusion

This paper describes Simpl DSL toolkit. Simpl is intended to cover most of the aspects of developing a DSL and to be a good fit with large-scale software development. Simpl is built on existing tools, such as ANTLR, Scala, and Eclipse. A DSL implementation built with Simpl consists of two separate components. One component implements parser, AST, program checker and code generator. This component has minimal amount of external dependencies and can be easily embedded into a larger system. The other component provides IDE for editing DSL programs. It is implemented as an Eclipse plugin.

The AST of the DSL program is stored in Scala case classes. These classes are automatically generated from grammar description and reflect the structure of the grammar. The DSL developer can modify the class model in AST by annotating the grammar description with return types and return expressions. The language processing (type checking, program transformation) and IDE services are implemented in Scala programming language. Simpl bundles two different libraries to simplify the code generation task.

We performed two tests to evaluate usability of the Simpl tool. The first test involved implementing a fairly complex DSL using Simpl. We collected code metrics and compared them to previously published implementations that used other DSL tools ANTLR, OMeta and “M”. Simpl yielded the lowest NCLOC measure and the code complexity (measured as percentage of NCLOC in units with cyclomatic complexity greater than 6) was slightly lower than with the other tools.

The second usability test was based on a controlled usability study that compared Simpl to popular DSL tool ANTLR. The subjects were junior and senior professional programmers who implemented a realistic, non-trivial DSL. After completing the task, the subjects were presented with a questionnaire that evaluated the tool’s ease of use and the subject’s satisfaction with the tool. The results showed that Simpl users implemented the task faster than the ANTLR users and rated the usability and satisfaction with the used tool higher than the ANTLR users. From these tests we conclude that Simpl does offer
usability advantages over existing DSL tools.

Our future work will involve evaluating Simpl in industrial software development projects to gain insight into Simpl’s usability and usefulness in real-life setting. The feedback collected in this way will be used to further improve Simpl and make it better in matching the needs of professional software developers.

Acknowledgement. This research is partly funded by the EU Regional Development Funds through the Estonian Competence Centre Program.

References


