On the Impact of DSL Tools on the Maintainability of Language Implementations

Paul Klint
Centrum Wiskunde & Informatica
Science Park 123
Amsterdam, The Netherlands
paulk@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica
Science Park 123
Amsterdam, The Netherlands
storm@cwi.nl

Jurgen Vinju
Centrum Wiskunde & Informatica
Science Park 123
Amsterdam, The Netherlands
jurgenv@cwi.nl

ABSTRACT

Does the use of DSL tools improve the maintainability of language implementations compared to implementations from scratch? We present empirical results on aspects of maintainability of six implementations of the same DSL using different languages (Java, JavaScript, C#) and DSL tools (ANTLR, OMeta, Microsoft "M"). Our evaluation indicates that the maintainability of language implementations is indeed higher when constructed using DSL tools.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Processors**]: Interpreters; D.3.2 [**Language Classifications**]: Specialized application languages; K.6.3 [**Software Management**]: Software maintenance

General Terms

Languages, Human Factors, Measurement

Keywords

Domain specific languages, Language engineering, Maintainability, Tools

1. INTRODUCTION

Domain Specific Languages (DSLs) promise an increase in productivity, maintainability and reliability by providing notations tailored to certain problem domains [14, 23]. Solutions can be described at a higher level of abstraction, thus narrowing the gap between problem domain an solution domain.

There are many approaches for the development of DSLs. *Internal* (embedded) DSLs reuse the syntax and parser of a general purpose host language (e.g., Ruby, Scala). *External* DSLs, on the other hand, are often developed using DSL tools. Such tools include parser generators (e.g., ANTLR, Yacc), transformation systems (e.g., ASF+SDF, Stratego, TXL) or attribute grammar systems (e.g., JastAdd, Kiama, LISA). The goal of these tools is to lower the cost of constructing DSLs.

Just like software has to evolve to remain viable, DSLs are subject to maintenance activities as well [23]. Especially DSL implementations are complex and highly detailed when compared to mainstream software applications. Therefore, maintainability of DSL implementations seems to be more of a critical factor than usual. Personally, we have also observed that maintaining language implementations requires considerable effort in practice, even when using DSL tools. Our main goal is therefore to investigate the significance of the influence of the use of DSL tools on the maintainability of the resulting implementations.

We present an initial empirical study on six implementations of the same DSL. Three implementations are developed from scratch, in Java, JavaScript and C# respectively. We call these implementations the "vanilla" implementations. The other three implementations are supported by DSL tools: ANTLR (Java) [17], OMeta (JavaScript) [26] and "M" of the Microsoft SQL Server Modeling Platform (C#) [16].

We put forward that these implementations are representative of DSL implementations in general. The vanilla implementations use common design patterns for language implementation. The respective DSL tool implementations have been reviewed by the authors of ANTLR and OMeta, and "M" experts of Microsoft Nederland, respectively.

In this paper we focus on the difference between vanilla implementations and implementations using DSL tools. Our research question is:

> **Is the use of DSL tools beneficial for the maintainability of the resulting language implementation as compared to not using them?**

We hypothesize that, given that maintenance is done by an experienced language engineer who understands the DSL tools, the answer to this question is indeed affirmative. Our experimental evaluation is necessarily targeted at invalidating this hypothesis. We will try to find indications of the DSL tool implementations being equally or even less maintainable. Such indications are certainly not unthinkable. Consider the extensive use of action code that is necessary to implement a C++ parser using Yacc: a vanilla implementation of the same parser could easily be more maintainable.

```
1  module HomePage                module Utils
2  import Utils                   def layout(name)
3  def home                         html {
4    layout("Home") {                 head title name;
5      h1 "My homepage";              body yield;
6      p "Hello world!";              }
7    }                              end
8  end
```

⇓

```
<html>
 <head>
  <title>Home</title>
 </head>
 <body>
  <h1>My homepage</h1>
  <p>Hello world!</p>
 </body>
</html>
```

Figure 1: WAEBRIC program for a simple home page, including its output.

Contributions. This paper represents the following contributions:

1. We have published more than six implementations of the same non-trivial DSL as open source software[1]. The implementations all include a parser, checker and evaluator.
2. We present quantitative data (metrics) and qualitative analyses on six of these implementations in this paper.
3. To our knowledge, this is the first work empirically investigating maintainability aspects of the use of DSL tools.
4. Our results largely confirm the hypothesis that the use of DSL tools increases maintainability.

We explicitly do not derive any recommendations from this study as to whether one should use a certain tool or not, since there are many aspects to the quality of DSL implementations; maintainability is just one of them. We consider performance, flexibility (e.g., error handling), and usability (e.g., debugging, testing etc.) as directions for future work.

2. A LITTLE LANGUAGE FOR MARKUP GENERATION

The DSL used in this study is WAEBRIC, which is a little language for XHTML markup generation. WAEBRIC is used for creating and maintaining (static) websites, by providing the user with abstraction facilities derived from functional programming. Sites can be built using reusable, functional building blocks. Figure 1 shows an example of how to generate a simple homepage.

This program consists of two modules. The first, `Homepage` imports the second, `Utils`. The WAEBRIC function `home` defines a very simple homepage. It invokes the utility function `layout` with a single argument, the string literal "Home". Additionally a block (lines 4–7) is passed into the `layout` function. This block consists of two statements. The first produces an `h1` element and the second produces a `p` element. All valid XHTML tag names are built-in "functions" that generate the corresponding elements.

[1] `http://waebric.googlecode.com`

```
1  def the-menu
2    menu({title: "Menu", kids: [...]})
3  end
4  def menu(model)
5    echo menu.title;
6    ul each (kid: model.kids)
7        menu-item(kid);
8  end
9  def menu-item(item)
10   if (item.kids)
11     li menu(item);
12   else
13     li a(href=item.link)
14       item.title;
15 end
```

⇓

Menu

- Home
- Misc
  - Contact
  - Links

Figure 2: Recursive menus in WAEBRIC with output.

The `layout` function, defined in the `Utils` module, produces a reusable skeleton for web pages, consisting of the root `html` element, containing a `head` and a `body` element. The block passed to the `layout` function is spliced into the `body` element using the built in **`yield`** statement (line 5). Note that nested within the `head` is a `title` element containing the `name` argument of `layout` (curly braces are optional if nesting occurs along a single spine).

WAEBRIC is a non-trivial language. In order to better appreciate the complexity of the implementations we present in Section 4, we highlight some distinguishing features here. As can be seen from the example in Figure 1, WAEBRIC is a modular language featuring function calls. A limited form of closures is catered for in order to pass additional code blocks into functions (cf. the body of a page passed into the `layout` function, which is then invoked using **`yield`**).

A more elaborate example is shown in Figure 2. It shows how to create a recursive menu from recursively structured data. The structure of the menu is provided to the function `menu` in the form of a data record with two fields, "title", and "kids". An excerpt of the data structure is shown in function `the-menu` (line 1–3). The function `menu-item` processes a single element of the list of children by checking if a `kids` field is present and if so, recursing to `menu` for each child. Otherwise a `<li>` element is created with an anchor based on the `link` field.

The example shows how data literals, conditionals, iteration, and recursion are used. In addition to these features, WAEBRIC has support for **`let`** bindings for introducing local variables and functions.

WAEBRIC implementations are accompanied by a light-weight semantic analyzer which checks for common mistakes. Among the

constraints that have to be checked, are the following: user defined functions are called with the right number of arguments, function definitions are unique (no duplicates), and referenced variables are either introduced as formal parameters of the enclosing function, or introduced by the `let` construct.

3. EVALUATING MAINTAINABILITY

We briefly introduce necessary concepts and methods for the evaluation of software maintainability here and explain our evaluation method for our six implementations of WAEBRIC. The results can be found in Section 4.

3.1 Software Maintainability

Software maintenance is defined by the IEEE as follows [9]:

> The process of modifying a software system or component after delivery to correct faults, to improve performance or other attributes, or adapt to a changed environment.

Maintenance can be further divided in corrective (fixing defects), adaptive (responding to a changed environment), perfective (e.g., improving performance or other attributes), and preventive maintenance (detecting and correcting undiscovered defects). *Maintainability* is then defined as the ease with which a software system or component can be maintained [9].

According to ISO/IEC 9126-1 maintainability can be subdivided in the following characteristics: analyzability, changeability, stability, testability and maintainability compliance[2]. However, as noted in [7], the standard provides no guidance as to how to relate specific measurements on source code to these characteristics. Other prevailing metrics used to estimate maintainability, such as Halstead Effort and the Maintainability Index (MI) have certain practical limitations. These source code metrics aim to quantify the maintainability of a software system by providing a single aggregated number over an entire software system. However, the formulae for computing both Halstead effort and the Maintainability index aggregate over program units using both averages and logarithms. As a consequence, the obtained metrics are hard to trace back to properties of the source code [13] .

As an alternative, Kuipers et al. [13] propose to rank a software system for maintainability assessment along the following dimensions:

Volume measured in number of modules (e.g., files, classes etc.), units (e.g. methods) and non-comment lines of code (NCLOC), should be as low as possible.

Structural complexity measured as the percentage of NCLOC in units that have a cyclomatic complexity (CC) higher than a certain value *X* should be as low as possible.

Duplication measured as the percentage of duplicated code for a given minimal duplicated-fragment size.

We have measured these dimensions to highlight possible issues with respect to maintainability. Observing an equal or larger volume, equal or more complex structural complexity or unavoidable code duplication for any of the DSL implementations as compared to the vanilla implementations would invalidate our hypothesis.

[2]Taken from [7].

	Unit	Decision point
General purpose	method, function	if, for, while, case, catch
Grammar	non-terminal	$\|$, $*$, $+$, ?

Table 1: Units and decision points in general purpose code and grammar code.

Note that we have not formally measured the percentage of duplicated code. However, we will occasionally refer to instances of duplication that have been observed in the code. The cyclomatic complexity threshold *X* is discussed in more detail below, in Subsection 3.3.

3.2 Grammar Metrics

The metrics discussed above are usually defined for ordinary code, and not for the grammar formalisms used by ANTLR, OMeta and "M". Although NCLOC and number of files can be computed in exactly the same manner as for general purpose code[3] the other metrics require a domain specific interpretation.

We have used the grammar-based interpretation proposed by Power and Malloy [20]. Under this interpretation each non-terminal corresponds to a single code unit. Similarly, the cyclomatic complexity of a grammar is defined to be equal to the number of decision points it contains. In grammars, such decision points are represented by alternatives ($|$), optionals (?), and the closure operators for iteration ($*$, $+$). A summary of the correspondence is shown in Table 1.

Although [20] only defines the CC for a complete grammar, we have counted the CC per non-terminal. This allows us to reason about the ratio between simple and complex parts of a grammar, as laid out by the maintainability model we use (Section 3).

3.3 How we measured

We have constructed generic (parse-tree-based) metric analyses using Rascal[4], a DSL for source code analysis and transformation [10]. All sources are parsed using approximate parsers for the languages involved. These parsers are approximate in the sense that they only recognize structure that is actually required for our metric extraction purposes. The resulting parse trees are processed in Rascal.

To compute NCLOC, the layout nodes of the parse trees are inspected. Layout nodes are parse tree nodes containing white space and comments. This information is sufficient to count line breaks and skip comments. Multiple consecutive line breaks are counted as one.

CC is obtained through the use of production annotations. Productions capturing syntactic constructs that involve conditional control flow are marked with annotations that can be retrieved from the parse tree. By using the same annotations in each grammar, we have been able to reuse the exact same analyses across different languages. Thus we eliminate the possibility that different tools (e.g., for Java and C#) count in different ways.

[3]The validity of this decision is discussed in more detail in 5.

[4]`http://www.rascal-mpl.org`

There is the issue of *preamble code* that will end up in the generated parser. It should contribute to the NCLOC metrics for grammar files. If the preamble contains methods these are counted as units as if they were put in ordinary source files and their cyclomatic complexity is measured accordingly.

If the production(s) for a non-terminal have *action code* attached, the CC of the code is additionally counted as CC of the non-terminal and the NCLOC of the code is added to the NCLOC of the module. Maintaining a DSL implementation naturally includes maintaining the action code, if present. Also note that action code can conditionally influence choice of alternates or trigger backtracking behavior, directly influencing the control flow of the generated parser.

To compute the percentage of NCLOC that contributes to units with high complexity, the threshold is set at 6. This means that the percentages, described in detail below, apply to units with a CC greater than 6. This relatively low threshold acknowledges the intrinsic complexity of DSL implementations; they should be kept as simple as possible.

4. QUANTITATIVE OBSERVATIONS

Here we present quantitative facts acquired about the six WAEBRIC implementations. These facts give rise to a qualitative evaluation presented later in Section 5.

4.1 Overview

We describe the differences between the various WAEBRIC implementations. Differences may be caused by the use of different general purpose languages, different developers that implemented WAEBRIC and different DSL tools applied. We have investigated every significant difference by scrutinizing the source code (Section 5).

The implementations described in this paper have originally been presented in three master's theses supervised by the second author [6, 24, 25]. As mentioned in Section 1, each implementation has been scrutinized by the best available expert in the application of the respective DSL tool.

In order to gain confidence in the fact that all six implementations actually implement the same language, a test suite of 104 WAEBRIC programs was used to compare the resulting XHTML to the output of the reference implementation which has been built by the second author using ASF+SDF[5]. All six implementations achieve 100% conformance on this test suite.

The vanilla implementations are quite similar. All implementations contain a handwritten lexer and parser. The parsers employ a recursive descent [1] strategy with variable look-ahead. The result of the parsing process is an abstract syntax tree (AST) consisting of nodes that are instances of classes in a statically typed hierarchy[6].

A common technique for tree traversal is the Visitor design pattern [5]. In the Java and C# implementation this pattern is applied for implementing the checker and interpreter. The JavaScript implementation uses explicit dispatch using `instanceof` execute actions for specific nodes.

In order to generate XHTML code from WAEBRIC programs two implementations depend on third-party libraries. First, the Java implementation uses the JDOM XML library[7]. Second, the JavaScript implementation uses John Resig's `env.js`[8] to simulate a browser environment, thus allowing internal DOM manipulation to be used for generating XHTML, even without support of an actual browser. C# uses a single custom XHTML class to generate XHTML from strings.

An overview of the different design choices in the DSL tool implementations is shown in Table 2. The terms "lexical grammar" and "context-free grammar" mean that the DSL tool provides separate (grammatical) notations for specifying scanner and parser. Since OMeta employs a scannerless formalism these two components are integrated. The row beneath the *Parse* row describes the parsing algorithm that is used by the tools.

In the context of AST implementation, "Generated" means that the tool provides automatic AST generation; in all implementations these ASTs are dynamically typed (even if the tool could facilitate otherwise). "Reused"—only in the case of OMeta—entails that the implementation reuses the component in question from its vanilla counterpart.

For tree traversal (used in check and eval components) ANTLR provides the *tree grammar* formalism. The OMeta checker uses grammar extension (inheritance) to implement visiting and matching behaviour; the evaluator is reused from the vanilla JavaScript implementation. Finally, "Dispatching" means that dynamic switch statements are used to traverse ASTs.

We conclude by observing that the DSL tools have significantly different characteristics, motivating our choice to include them all in our experiment. The following describes the volume and structural complexity results of each of the six implementations.

4.2 Volume

The results of our volume measurements are shown in Table 3. The first column lists the various implementation components. To separate grammar code from ordinary code, three components (parse, check and eval) have an additional row containing the metrics computed from grammar code: C_c indicates the ordinary code in the component, whereas C_g indicates grammar code ($C \in \{\text{Parse}, \text{Check}, \text{Eval}\}$).

Metrics for the scanning component are included in the grammar parsing row. Columns 2 till 7 represent each WAEBRIC implementation. Each implementation column is divided in three subcolumns, corresponding to number of files (#F), number of units (#U) and NCLOC (#N). An "–" indicates that the number for a certain component and implementation is zero.

Figure 3(a) graphically depicts the total size of each implementation in number of files, units, and NCLOC per implementation. Note that this graph has a logarithmic scale. Figure 3(b) shows the percentage of code reduction that has been obtained

[5] `http://code.google.com/p/waebric/source/browse/#svn/trunk/implementations/asf+sdf`.

[6] JavaScript is a prototype-based language so there are no classes. "Statically typed" thus means that every node is an instance of a named object.

[7] `http://www.jdom.org`.

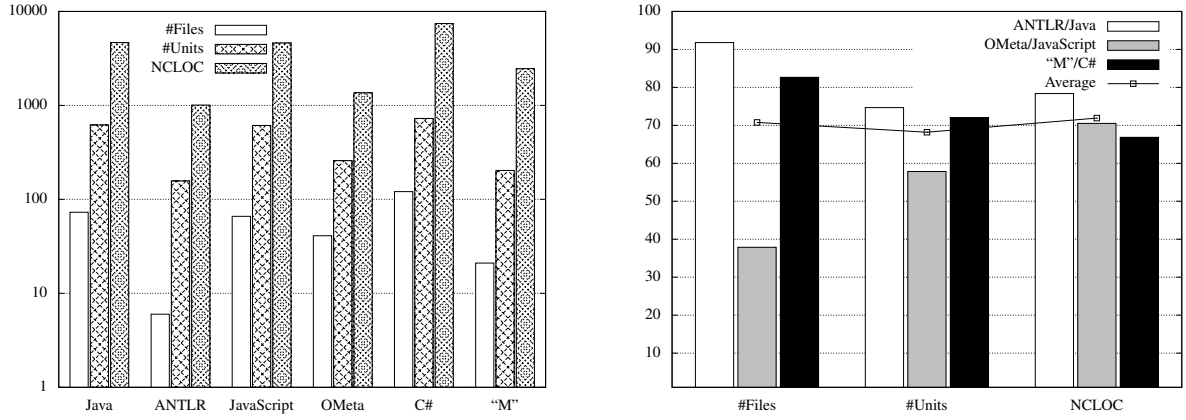
[8] `http://ejohn.org/blog/bringing-the-browser-to-the-server/`

	ANTLR	OMeta	"M"
Scan	Lexical grammar	Integrated	Lexical grammar
Parse	Context-free grammar		Context-free grammar
Technique	*LL(*)*	*PEG/Packrat*	*GLR*
AST	Generated	Reused	Generated
Check	Tree grammar	Grammar extension	Dispatching
Eval	Tree grammar	Reused	Dispatching

Table 2: Overview of implementation strategies in the DSL tool implementations

	Java			ANTLR			JavaScript			OMeta			C#			"M"		
Component	#F	#U	#N	#F	#U	#N	#F	#U	#N	#F	#U	#N	#F	#U	#N	#F	#U	#N
Scan	6	71	491	–	–	–	13	68	530	–	–	–	10	61	**725**	–	–	–
Parse_c	13	79	958	–	–	18	16	166	**1724**	3	6	51	12	91	1366	2	4	70
Parse_g	–	–	–	1	52	133	–	–	–	1	66	144	–	–	–	1	80	**312**
AST	39	360	2045	–	–	–	22	130	612	22	130	612	81	464	**3647**	–	–	–
Check_c	7	17	183	–	–	–	6	93	**597**	7	27	261	9	22	334	9	26	430
Check_g	–	–	–	2	73	**294**	–	–	–	1	10	72	–	–	–	–	–	–
Eval_c	2	65	596	–	–	–	6	153	1074	–	–	–	7	86	1264	7	91	**1574**
Eval_g	–	–	–	1	29	**377**	–	–	–	–	–	–	–	–	–	–	–	–
Misc	6	28	**387**	2	3	186	3	2	82	7	19	222	2	2	84	2	2	74
Total	73	620	4660	6	157	1008	66	612	4619	41	258	1362	121	726	7420	21	203	2460

Table 3: Volume metrics: number of files (#F), number of units (#U), and NCLOC (#N). The maximum in each row is in bold face.



(a) Volume totals

(b) Code reduction induced by DSL tool usage

Figure 3: Overview of volume measurements per WAEBRIC implementation.

Component	Java	ANTLR	JavaScript	OMeta	C#	"M"
Scan	**31%**	–	9%	–	**31%**	–
Parse_c	**22%**	–	2%	–	17%	–
Parse_g	–	20%	–	**26%**	–	10%
AST	–	–	–	–	–	–
Check_c	–	–	8%	10%	**18%**	12%
Check_g	–	**23%**	–	–	–	–
Eval_c	13%	–	4%	–	27%	**33%**
Eval_g	–	**29%**	–	–	–	–
Misc	**32%**	–	–	–	–	–

Table 4: Percentage of NCLOC in units with cyclomatic complexity higher than 6. The maximum in each row is in bold face.

by using a DSL tool, in % per measured metric: $\%_{reduction} = 100 - (\#_{tool}/\#_{vanilla}) \times 100$, where # $\in$ {#Files, #Units, NCLOC}. The line indicates the average code reduction per metric.

Figure 3 immediately shows that the use of the DSL tools consistently reduces the number of files, units and NCLOC in each pair of WAEBRIC implementations. The average reduction in code volume is around 70%. The percentage chart shows, for instance, that the reduction in NCLOC by using ANTLR is roughly 80%, which means that this implementation requires a mere one fifth of the NCLOC that are required in the Java vanilla implementation.

Parsing is the part that is affected most by the use of the tools, since all three tools are primarily parser generators. This can be easily understood since parser generators take away much of the boiler plate code required to write a parser, for instance iteration, backtracking and look-ahead. This is especially visible in the number of units metric (#U): more code units seem to be required than non-terminals, especially if we add the units for scanning and parsing. For instance, in the vanilla Java implementation, the number of units for scanning and parsing add up to 150, whereas these two phases implemented using ANTLR require only 52 non-terminals. For all implementation pairs the ratio between the number of units amounts to a factor of roughly 2 (C#/"M" = 152/80) or 3 (Java/ANTLR = 150/52, JavaScript/OMeta = 234/66).

AST code takes up a large portion of the vanilla code. Both ANTLR and "M" generate (dynamically typed) ASTs so this kind of code is completely absent from those DSL tool implementations; it is responsible for a large part of the code reduction. In the OMeta implementation the vanilla AST hierarchy is reused; we have included these numbers in the results for OMeta.

4.3 Structure

The cyclomatic complexity measurements are shown in Table 4. It shows the percentage of NCLOC that lives in units with a CC value higher than the threshold value 6. A lower percentage is better. The maximum in each row is in bold face.

At first sight, one would think that complexity is reduced: for instance, if we look at the percentages of the scanning and parsing component of the vanilla Java implementation, then we see that both percentages (31% and 22%) are higher than the ANTLR percentage (which combines scanning and parsing). "M" vs. C# shows a similar trend, except the difference is much bigger (31% and 17% vs 10%). In both cases the difference can be accounted for by look-ahead code which heavily uses conditions.

In the case of OMeta, however, the picture is different: the grammar complexity is more than twice as large as the vanilla JavaScript scanner and parser. Inspection of the code leads to the conclusion that the vanilla JavaScript implementation uses extensive factoring in the scanner and parser thus reducing CC per method. In Table 3 this is confirmed by the large number of methods (234) in those components.

The AST components in the vanilla implementations have very low complexity. This can be understood from the fact that such hierarchies basically define an algebraic data type containing primarily getters, setters, and, possibly, an accept method for traversing using the Visitor design pattern. This code can be considered "boilerplate" code.

In terms of complexity static AST hierarchies as such can reduce the cyclomatic complexity of the code that uses them, for instance in the check and eval components. This is most visible in the C# vs "M" complexity results: the eval component of "M" has an atypical high complexity. This can be understood from the fact that "M" produces dynamically typed ASTs which can only be traversed using explicitly coded dispatch. If ASTs were to conform to a static hierarchy, the Visitor design pattern could have been used. Additionally, traversal for evaluation is not structure shy: you need to visit all node types. As a consequence 33% of NCLOC is spent in high complexity methods.

In the ANTLR implementation, the complexity of check and eval is relatively high because the real work is done in action code embedded in tree grammar productions. The CC of this code added to the CC of the productions themselves, thereby increasing the overall complexity.

Finally the utility code (Misc) in the vanilla Java implementation has a high complexity percentage due to a large `main` entry method with a high CC.

5. QUALITATIVE EVALUATION

After having presented the complexity of our implementations in terms of volume and structure in the previous section, we return here to our research question: does the use of DSL tools lead to implementations that are more maintainable? We use the quantitative results to guide the discussion, and refer to qualitative aspects of the code (such as duplication) when necessary. The discussion is structured along the main design patterns used in the implementations.

5.1 Overview

Scanning and parsing. Considering the volume metrics, the use of DSL tools clearly improves the maintainability of the implementations. In other words, one generally needs to understand and adapt less code. This certainly applies in the context of scanning and parsing, since the DSL tools used in this study are basically parser generators.

Using a DSL tool for scanning and parsing also reduces the structural complexity of parser implementations. DSL tools provide high-level primitives to describe the grammar of a language in a declarative way. As such they abstract over a lot of detail that has to be taken into account when writing scanners or parsers from scratch. The grammar codes represents the intrinsic complexity of parsing WAEBRIC, while the vanilla code exhibits overhead—accidental complexity.

However, the JavaScript/OMeta case strongly contradicts the above (Table 4), and we must conclude that using a DSL tool does not necessarily make the resulting implementation of the parser less complex. Even in a relatively complicated language as WAEBRIC, a DSL tool may increase structural complexity as compared to a well-factored vanilla implementation. Nevertheless, given the positive results on volume, and the lack of code duplication we have observed, we should not invalidate our hypothesis based on this observation.

AST generation. Both ANTLR and "M" provide AST generation facilities. This obviates the need for maintaining AST boilerplate code. This is a very strong point in favour of DSL tools since

AST hierarchies co-evolve with the grammar, which leads to the duplicate maintenance problem: if the grammar changes, the AST hierarchy has to change accordingly.

AST traversal. The DSL tools used in this study do not provide extensive facilities for writing checkers and/or interpreters. Although both ANTLR and OMeta provide mechanisms to traverse AST nodes, the advantages of these features over the use of the Visitor pattern are less clear cut. This is especially the case when only a minority of nodes have to be visited (i.e. the traversal is *structure-shy*).

A case in point is the ANTLR implementation. Looking at the checker and interpreter we see that the complexity percentage of the checker and interpreter is 23%, and 29% respectively. With respect to volume, the evaluator implemented with ANTLR is not even twice as large as the checker whereas in the other implementations, the evaluators are about 3 times (Java), 2 times (JavaScript), 10 times (C#), and 3 times ("M") as large as the checkers. One would expect evaluators to be larger than checkers since they tend to "do" more than checkers. The reason for this discrepancy between ANTLR and the rest is that ANTLR tree grammars have to be completely copied for each implementation component that has to traverse the AST. As a consequence, the checker cannot be implemented in a structure-shy fashion. Our results show that the check component of ANTLR is even larger than in the vanilla implementation.

Additionally, this introduces a form of duplication that is especially hard to maintain: the tree grammars co-evolve with the normal grammar and with each other. The author of ANTLR is well aware of this limitation (cf. the footnote on page 193 of [17]) and attempts to solve this problem in [18]. The current version of ANTLR also has a feature called *tree pattern matching* that can be used to implement, for instance, well-formedness checkers in a structure-shy fashion [19].

Types and dispatch. Above we noted that AST generation in general is a good thing since it reduces code. However, in the ANTLR and "M" implementations discussed here, the generated ASTs were dynamically typed: the type of node is encoded as a field in a generic node object. This did not present a visible problem in the ANTLR implementation because tree grammars were used. In the "M" version this resulted in high complexity results for the check and eval components, because traversal had to be programmed using explicit dispatch (switch statements). A static AST hierarchy makes it possible to use the Visitor pattern.

A default Visitor, possibly generated from the grammar, has an additional advantage: it allows specialization to be used for implementing structure-shy traversals. This can be done by inheriting from the default Visitor and overriding only the relevant visit methods. This way, explicitly coded dispatch as well as co-evolution of code that follows the structure of the grammar can be minimized.

A similar scheme is possible using OMeta, which has been applied in the implementation of the check component. There, the OMeta grammar that is used for parsing WAEBRIC is specialized (using "grammar inheritance") for implementing this component. Only the relevant productions are overridden with additional action code that is used for well-formedness checking of WAEBRIC programs.

5.2 Threats to Validity

This study is subject to a number of threats to validity. We distinguish construct validity, internal validity and external validity. Construct validity applies to whether the metrics that we have used are valid indicators for maintainability (the construct of interest). Internal validity is concerned with the validity of our conclusion with respect to our measurements. External validity concerns the question to what extent we may generalize the results.

Does the set of measurements presented in Section 3 actually indicate the construct of interest, *maintainability*? Without additional research into actual maintenance of the six implementations, this question is hard to answer. However, the model is used extensively in practice by the Software Improvement Group (SIG) [7], mainly to find contra-indications of maintainability. More research validating the significance of the model is certainly warranted and would strengthen our conclusions.

Another threat to construct validity in this study might be that the NCLOC metric is traditionally defined and used on software systems implemented in general purpose languages (GPLs) and not on DSLs. Can a line of code in a DSL be compared to a line of code in a GPL? Our implicit assumption is that a language engineer when reading a line of grammar code understands it as easily and as fully as she understands a line of GPL code. This also highlights the widely accepted view that understanding takes a significant part of the time spent while maintaining software.

Nevertheless, the assumption of NCLOC compatibility between languages should be investigated further. The technique of back firing function points might be of use here where the NCLOC numbers are normalized with respect to language level [22]. A higher language level means that fewer statements are required to code a function point. Heitlager et al. report that this method, although not completely accurate, is sufficient and highly usable in practice [7]. Their results are not directly applicable here because ANTLR, OMeta and "M" are not in the table that assigns languages levels.

The primary concern with respect to internal validity is the question of optimal implementation. In order to derive valid data based on such implementation experiments the DSL tools in question should be used in the way they are intended to be used. The following actions have been undertaken in order to mitigate this risk. The implementations using ANTLR and OMeta have been reviewed by the respective tool authors and the "M" implementation has been reviewed internally at Microsoft Nederland. This at least gives some confidence that no obvious mistakes and/or idiosyncrasies exist in the implementations.

One issue of external validity is the representativeness of WAEBRIC as a DSL. Although the domain of WAEBRIC, XHTML generation, is very narrow, we think that WAEBRIC is a suitable language for experiments of this kind, for the following reasons. First, WAEBRIC has a non-trivial syntax, including expressions, statements and string interpolations. This already makes WAEBRIC more involved than many little languages around.

Second, WAEBRIC has many features that make it perhaps more like a programming language than a DSL proper: it includes the notions of conditional execution, looping, function call, closure, module import and recursion. This leads us to conclude that common aspects of programming languages have been exercised. Nev-

ertheless, the issue whether this is representative for typical DSLs is uncertain, also because DSLs are typically implemented using a compiler/code generator instead of an interpreter.

In this paper we have only investigated how properties of the source code might affect maintainability, but there are other factors that influence maintainability as well. First a tool has to be learned to be useful. This learning curve should not be underestimated, especially in the context of maintenance. Should the original developer(s) leave the company, someone else should learn the tool in order to keep the DSL implementation viable. This is even more a problem if documentation and/or support is non-existent or scarce.

Another aspect influencing maintainability is lack of IDE support. Code outliners, declare/use services, immediate feedback etc. may all alleviate the maintenance. Additionally, without IDE support for debugging, error diagnosis may be hard. If the tool uses code generation, it can be difficult to relate static or dynamic errors back to their original location in the code.

A tool may introduce additional complexity by mixing domain specific notations with ordinary general-purpose language code, for instance by having action code in between arbitrary grammar productions. This also exacerbates debugging. IDE support, if any, is even more complicated, since now the composition of two languages has to be catered for.

5.3 Related Work

Empirical research on DSLs and DSL implementation is scarce. In [2] a case-study is presented comparing the use of a DSL vs an object-oriented framework in the domain of financial engineering. More recently, Hermans et al. [8] present a user study on the perceived value of a real life DSL in industry. These papers, however, do not focus on the (dis)advantages of specific implementation approaches. In his masters thesis, Van Dijk focuses on changeability of a DSL-based web application, including the use of a DSL tool [4]. Changeability is another aspect of maintainability, next to understandability.

There is related work studying implementation approaches. For instance, [21] presents a case-study of post-design DSL embedding. This is primarily a qualitative experience report. More similar to our approach, is the work described in [11]. There, an empirical study of different implementation approaches is presented, comparing, for instance, embedding in Haskell, implementation using a language workbench (LISA), or using a parser generator (e.g., Yacc). The study combines source code metrics (volume), user questionnaires for usability and run-time performance measurements to compare different implementations. Although this work has the same objectives as ours, its scope is much wider. We have focused exclusively on maintainability and properties of the source code.

Additionally, the example DSL used, Feature Description Language (FDL) [3], is much "smaller" than WAEBRIC: it is basically a notation for boolean propositions. The grammar of FDL contains 11 non-terminals, whereas our WAEBRIC grammars have between 133 (ANTLR) and 312 ("M") non-terminals. Finally, not all FDL implementations (by design) achieve the original DSL notation, whereas we required the implementations to have feature-complete parsers.

An important task in software maintenance is understanding the existing program, hence, if a program is easier to understand, it will be easier to maintain. The effect of DSLs on program understanding is investigated in [12] and [15]. In [12] the authors describe an empirical experiment measuring the performance of end-users in three categories: learning, perceiving and evolving programs. It was shown that the success rate on average was 15% better in the context of the DSL (Graphviz' Dot language) as compared to using an application library in the same domain. A similar experiment is described in [15], this time comparing the use of XAML for user-interface construction to using C# Forms; again an average of 15% better performance is reported. Since the DSL tools studied in the current paper consist themselves of (one or more) DSLs, these results seem to corroborate the conclusion that DSL tools increase maintainability of DSL implementations.

6. CONCLUSION

Our results suggest that DSL tools do indeed increase maintainability of DSL implementations. Especially our code reduction measurements corroborate the hypothesis. The use of a DSL tool does not necessarily make the resulting implementation less complex in a structural sense, but it does usually remove the need for boilerplate structure. The structure of a parser remains intrinsically complex though, even when coded as a grammar.

Acknowledgements. The Java and ANTLR implementations have been developed by Jeroen van Schagen. Nickolas Heirbaut developed the JavaScript and OMeta implementations. Jeroen van Lieshout worked at Microsoft Nederland to develop the C# and "M" versions. We additionally thank Jeroen Quakernaat and Dennis Mulder for their support at Microsoft Nederland. We thank Terence Parr and Alessandro Warth for respectively reviewing the ANTLR and OMeta implementations. Finally, the anonymous reviewers are acknowledged for their helpful suggestions for improving this paper.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1st edition, 1986.

[2] A. v. Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39. Ilmenau Technical University, 1997.

[3] A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.

[4] D. v. Dijk. Changeability in model driven web development. Master's thesis, Universiteit van Amsterdam, Master Software Engineering, 2009.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] N. Heirbaut. Two implementation techniques for domain specific languages compared: OMeta/JS vs. JavaScript. Master's thesis, Universiteit van Amsterdam, 2009.

[7] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology (QUATIC'07)*, pages 30–39.

IEEE, 2007.

[8] F. Hermans, M. Pinzger, and A. van Deursen. Domain-specific languages in practice: A user study on the success factors. In A. Schürr and B. Selic, editors, *Proceedings of 12th ACM/IEEE Conference On Model Driven Engineering Languages And Systems (MODELS'09)*, pages 423–437, 2009.

[9] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[10] P. Klint, T. van der Storm, and J. Vinju. RASCAL: a domain specific language for source code analysis and manipulation. In *Proceedings of 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*. IEEE, 2009.

[11] T. Kosar, P. E. M. López, P. A. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[12] T. Kosar, M. Mernik, M. Črepinšek, P. R. Henriques, D. da Cruz, M. J. V. Pereira, and N. Oliveira. Influence of domain-specific notation to program understanding. In *2nd Workshop on Advances in Programming Languages (WAPL'09)*, October 2009.

[13] T. Kuipers and J. Visser. Maintainability index revisited. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007. Special session on System Quality and Maintainability (SQM).

[14] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[15] M. Mernik, T. Kosar, M. Črepinšek, P. R. Henriques, D. da Cruz, M. J. V. Pereira, and N. Oliveira. Comparison of XAML and C# forms using cognitive dimension framework. In *INFORUM Simpósio de Informática*, 2009.

[16] Microsoft. "M" modeling language. `http://msdn.microsoft.com/en-us/library/ee460940.aspx` (March 2009).

[17] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf, 2007.

[18] T. Parr. The reuse of grammars with embedded semantic actions. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008.

[19] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009.

[20] J. Power and B. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, 2004.

[21] A. Sloane. Post-design domain-specific language embedding: A case study in the software engineering domain. *Hawaii International Conference on System Sciences*, 9:281, 2002.

[22] Software Productivity Research LCC. "Programming Languages Table", 2007. version 2007d.

[23] A. van Deursen and P. Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[24] J. van Lieshout. Benchmarken van een DSL implementatie in "Oslo" en C#. Master's thesis, Universiteit van Amsterdam, 2009. In Dutch.

[25] J. L. van Schagen. Measuring the quality of domain-specific language implementation approaches: Java versus ANTLR. Master's thesis, Universiteit van Amsterdam, 2009.

[26] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS'07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.