Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 50 (2008) 390–405
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

A preliminary study on various implementation approaches of domain-specific language ☆

Tomaž Kosar [a,*], Pablo E. Martínez López [b], Pablo A. Barrientos [b], Marjan Mernik [a]

[a] University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, 2000 Maribor, Slovenia
[b] Universidad Nacional de La Plata, Facultad de Informática, LIFIA (1900) La Plata, Buenos Aires, Argentina

Received 14 March 2006; received in revised form 8 March 2007; accepted 6 April 2007
Available online 21 April 2007

Abstract

Various implementation approaches for developing a domain-specific language are available in literature. There are certain common beliefs about the advantages/disadvantages of these approaches. However, it is hard to be objective and speak in favor of a particular one, since these implementation approaches are normally compared over diverse application domains.

The purpose of this paper is to provide empirical results from ten diverse implementation approaches for domain-specific languages, but conducted using the same representative language. Comparison shows that these discussed approaches differ in terms of the effort need to implement them, however, the effort needed by a programmer to implement a domain-specific language should not be the only factor taken into consideration. Another important factor is the effort needed by an end-user to rapidly write correct programs using the produced domain-specific language. Therefore, this paper also provides empirical results on end-user productivity, which is measured as the lines of code needed to express a domain-specific program, similarity to the original notation, and how error-reporting and debugging are supported in a given implementation.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Domain-specific languages; Embedded approach; Preprocessing; Compiler/interpreter; Compiler/interpreter generator; Extensible compiler/interpreter; Commercial-Off-The-Shelf

1. Introduction

A domain-specific language (DSL) is a language designed to provide a notation tailored toward an application domain, and is based only on the relevant concepts and features of that domain. As such, a DSL is a means of describing and generating members of a program family within a given domain, without the need for knowledge about general programming. By providing notations tailored to the application domain, a DSL offers substantial gains in productivity and even enables end-user programming. The main disadvantage of DSLs is the cost of their development, requiring both domain and language development expertise. This is one of the reasons why DSLs are rarely used in solving software engineering problems. The other one is the lack of guidelines and experienced reports on DSL development. In a recent survey on DSLs, various patterns on when and how to develop them were identified and explained in a qualitative manner [19]. However, the quantitative approach was stated as an open problem in the DSL research field. Can the costs and benefits of DSLs be reliably quantified? This paper presents a first step in this direction.

Although at present many articles have been written on the development of particular DSLs, there are none, to the best of our knowledge, that compare different implementation approaches on the same DSL. Since these DSLs were implemented for very different domains and used various

☆ This work is sponsored by bilateral project "Implementation of DSLs: Evaluation of Approaches" (code ES/PA02/E01, BI-BA/03-05-002) between Slovenia and Argentina, part of the program SECyT-MESS.
* Corresponding author. Tel.: +386 22207448.
E-mail address: tomaz.kosar@uni-mb.si (T. Kosar).

0950-5849/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2007.04.002

implementation approaches, it is hard to be objective and speak in favor of a certain domain-specific language implementation approach. The main goal of this paper is to compare various implementation approaches conducted on the same selected DSL. This comparison is based on the project *IDEA* – Implementation of DSLs: Evaluation of Approaches – carried out by the University of Maribor, in Slovenia, and the University of La Plata, in Argentina. In particular, we have taken the short guidelines given by Mernik et al. [19] on how to proceed with DSL implementation, and extended them, by providing a detailed description of our practical experience with each of them [17].

In order to perform a comparison of these implementations, several approaches were selected for testing, and then a fixed DSL was chosen as a case study. The study carried out was to implement the chosen DSL using all the selected approaches, to write programs using this DSL, and to compare the results obtained. Further, results regarding comparison were used to investigate the following claims about DSL development taken from the literature:

- The embedded implementation approach is more efficient when compared with other implementation approaches [8,13].
- Turning an API into a DSL incurs in a lot of additional work [10].
- Original notation is not fully feasible for all DSL implementation approaches [19].
- Error-reporting is unsatisfactory in all DSL implementation approaches [4,15].

These statements are the object of our study and are further examined in this paper. The organization of this paper is as follows. Related work and basic characteristics of DSLs are briefly explained in Section 2. An overview of the comparisons carried out on the specific DSL (together with the basis for the case study) are given in Section 3. Key findings are given in Section 4. Finally, concluding remarks are summarized in Section 5.

2. Related work

In order to implement a DSL, a programmer can choose from among several implementation approaches, and of course, the most suitable one should be chosen. However, there is a shortage of guidelines for helping DSL developers choose the most suitable implementation. The first attempt to classify these approaches was carried out by Spinellis [29], listing design patterns for DSLs. Later, these patterns were improved and extended by Mernik et al. [19]. For example, the Spinellis' pattern classification does not include traditional general-purpose language (GPL) design and implementation techniques, but Mernik's does. Moreover, the classification of patterns by Spinellis [29] does not correspond to any obvious way of classifying into decision, analysis, design, and implementation patterns. For example, the data structure representation pattern is classified as a creational pattern by Spinellis, while it is obvious that this pattern can be implemented using the different implementation patterns described. Mernik et al.'s implementation patterns are briefly presented below:

Preprocessing. DSL constructs are translated into constructs in the base language. Static analysis is limited to that done by the base language processor. There are important sub-patterns.

> *Macro processing.* In this approach, the meaning of new constructs is defined in terms of other constructs in the language.
> *Source-to-source transformation.* DSL source code is translated into the source code of an existing language. The translation is usually defined via syntax-directed patterns.
> *Pipeline.* Processors successively handling the sub languages of a DSL and translating them to the input language of the next stage.
> *Lexical processing.* Only simple lexical scanning is required, without complicated tree-based syntax analysis.

An advantage of this approach is that implementation of the DSL is very easy since most, if not all, semantic analysis is postponed and performed by the base language processor. However, this absence of semantic analysis is also the main disadvantage because there is no static checking and it is impossible to make optimizations at the domain level. Another disadvantage is that error reporting is a problem because error messages are in terms of base language concepts.

Embedding. In this approach, existing mechanisms in the host language are used to build a library of domain-specific operations. The syntactic mechanisms of the host language are used to express the idiom of the domain. The embedded DSL inherits the generic language constructs of its host language and adds domain-specific primitives that are closer to the DSL user. An advantage of this approach is that the compiler or interpreter of the host language is reused as it is, for the DSL. The main limitation is in the expressiveness of the syntactic mechanisms in the host language. In many cases, the optimal domain-specific notation is compromised because of the limitations of the host language. As with the previous approach, error reporting is also problematic, for the same reasons as before.

Compiler/interpreter. In this approach standard compiler/interpreter techniques are used to implement a DSL. In the case of the compiler, the DSL constructs are translated into base language constructs and library calls. A complete static analysis is done on the DSL program/specification. In the case of an interpreter, DSL constructs are recognized and interpreted using a fetch-decode-execute cycle. An important disadvantage is the cost of building a compiler/interpreter from scratch. However there are some advantages such as closer syntax to the notation used by domain experts, and good error reporting.

Compiler generator. This approach is similar to the previous one, except that some of the compiler/interpreter phases are implemented using language development systems or so-called compiler writing tools (compiler-compilers). In this manner the implementation effort is minimized, hence minimizing the disadvantages of the previous approach.

Extensible compiler/interpreter. The GPL compiler/interpreter is extended using domain-specific optimization rules and/or domain-specific code generation. Facilities like reflection and introspection are very useful for this approach when they are applicable. Compared to the previous two approaches, implementation effort is minimized due to the reuse of an existing complete compiler infrastructure. As a disadvantage it should be noted that extending a compiler is hard: extreme caution is needed to prevent any interference of domain-specific notation with an existing one.

Commercial Off-The-Shelf (COTS). Existing tools and/or notations are applied to a specific domain (e.g., XML-based DSLs). This approach provides a feasible alternative when solving particular domain problems (e.g., XML brings promising solutions in processing and querying documents). In general, XML tends to be cumbersome for humans to read and write.

Empirical research in software engineering is a difficult but important discipline. In order to avoid questionable results from experiments, certain conditions must be considered while preparing them. In [1] a framework is introduced to motivate and replicate studies. The proposed framework concentrates on building knowledge about the context of an experiment and is based on organizing sets of related studies (family of studies). These studies contribute to a common hypotheses which does not vary for individual experiments. The research presented in this paper is rather a comparison – a single study of domain-specific implementation approaches, where our comparison is the first step toward experimenting on domain-specific implementation approaches. Therefore, we have followed the guidelines, mentioned below, in order to prepare our comparison:

- comparison validity,
- context of the study,
- measurement framework, and
- presentation of key findings, focused on hard/unexpected/controversial results.

No other similar work on comparing DSL implementation approaches appears in literature, while the systematic study on DSL development [19] is rather new. In a certain sense our work is similar to that carried out by Prechelt [23], which provides some objective information comparing seven programming languages (C, C++, Java, Perl, Python, Rexx, and Tcl). The pros and cons of various programming languages were discussed on a given programming problem (string processing program), where programmers were asked to provide the same set of requirements. All implementations were compared for several properties, such as run time, memory consumption, source text length, comment density, program structure, reliability, and the amount of effort required to write them. Our study differs from Prechelt's in its object of 'comparison' (in our study this is a domain-specific implementation approach), the context of comparison and the extent of the experiment – in [23] several implementations were carried out in seven languages, while our comparison contains optimized samples on 10 diverse DSL implementation approaches.

3. Methodological issues

Many problems have to be faced while preparing the comparison. In the following section these issues are exposed and discussed according to their influence on comparison validity.

3.1. Subject of comparison

More than 3000 programming languages, general-purpose as well as domain-specific, have been developed in the past [32,33]. Only a small portion of them have been properly documented (e.g., [2]). Hence, a complete comparison of language implementation approaches will probably never be achieved. In a recent paper [19] around 40 DSLs were categorized using the approaches listed above. The study [19] showed that most of DSLs were implemented using the compiler approach (42.1%), followed by embedding (15.8%), and source-to-source sub-pattern (13.2%). Therefore, all the most commonly used implementation approaches have been included into our case study. Communities advocate their DSL implementation approaches and currently work on improvement of those techniques, supporting tools, etc. This study aims to help those communities by providing some realistic information for comparing different DSL implementation approaches, which is the subject of our comparison in this experiment. All implementation approaches were tested on the same DSL, in order to prepare comparison rigorously and systematically, and obtain realistic results.

In order to choose our case study, we took into account that we needed a DSL that is neither too simple nor too complex. One good language that we had been using is a language to capture the variable parts of application domains. It is called FDL (Feature Description Language) [7] and it was created to provide a textual description for feature diagrams [6], which is a technique used in domain analysis (specially FODA – Feature Oriented Domain Analysis [16]).

Feature diagrams and their textual representation FDL, provide ways to express what a given system is composed of. We used a simple example, in order to introduce both notations simultaneously. This example concerns a computer hardware composition. The feature diagram is pre-

sented in Fig. 1 and the corresponding FDL description in Fig. 2. Basically it expresses that a IO device is composed of several parts (indicated by the tree-like structure in the diagram, and by the operation all in the program). Some of these parts are optional (indicated by the empty circle in the diagram, and by the operation opt in the program), whilst some of the others are mandatory (full circle in diagram, default in program). Additionally, some features are atomic (i.e. they are basic units, indicated by leaves of the tree in the diagram, and by identifiers, starting with a lowercase letter in the program) and some of the others are composed of subfeatures – this is indicated by the tree-like structure in the diagram, and by *named features* (feature names starting with a capital letter) in the program. In regard to composite features, there are two ways to select alternatives: the first one is to pick only one option from among several (identified by the empty triangle in the diagram, and by operation one-of in the program), and the second one is to pick several of the options (identified by the full triangle in the diagram, and by operation more-of in the program). The meaning illustrated by the feature diagram or the FDL program is the set of all possible system configurations. Ninety-six various compositions of the IO devices are possible, for the given example.

The language FDL introduced in [7] has many benefits over the graphical description. One of them is its ability to express constraints. The purpose of constraints is to limit the variability of the system. A constraint can have one of the following forms:

- f1 requires f2: if feature f1 is present, then feature f2 must be present as well,
- f1 excludes f2: if feature f1 is present, then feature f2 must not be present,
- include f: feature f must be present,
- exclude f: feature f must not be present.

If the constraints webcamera requires microphone and include lcd are introduced to the example in Fig. 1, the original 96 possibilities are reduced to just 36.

In order to evaluate an FDL program, the following transformation steps (reductions) need to be applied [7].

Regularization. Named features are substituted with the corresponding definitions. For the given example, the regular FDL form is:

```
IOdevice : all    (opt(Printer), mouse, Display , opt(webcamera),
                   keyboard    , opt(microphone), opt(Receiver) )
Printer  : one-of (laser       , inkjet                          )
Display  : one-of (crt         , lcd                             )
Receiver : more-of(speaker     , headphones                      )
webcamera requires microphone
include   lcd
```

Fig. 2. IOdevice program using DSL.

$$\mathsf{all}(\mathsf{opt}(\mathsf{one-of}(\mathsf{laser},\mathsf{inkjet})),\mathsf{mouse},\mathsf{one-of}(\mathsf{crt},\mathsf{lcd}),\ \mathsf{opt}(\mathsf{webcamera}),\mathsf{keyboard},\mathsf{opt}(\mathsf{microphone}),\ \mathsf{opt}(\mathsf{more-of}(\mathsf{speaker},\mathsf{headphones})))$$

Normalization. Normalization is obtained using rules that simplify a given feature expression by eliminating duplicated features and degenerate cases, for the various constructors. Since detailed descriptions of all the normalization rules are unimportant for this paper, only the fifth normalization rule (from [7]) is shown below – it removes the nested appearance of features all.

$$[N5]\ \mathsf{all}(\mathsf{Fs},\mathsf{all}(\mathsf{Ft}),\mathsf{Fs}\prime) = \mathsf{all}(\mathsf{Fs},\mathsf{Ft},\mathsf{Fs}\prime)$$

(note that Fs and Fs' refer to possibly empty lists of features, while Ft refer to an inclusion of some features).

Expansion. Expansion rules are applied after normalization in order to transform normalized forms into disjunctive normal forms (DNF). Again, any detailed description of the rules is outside the scope of this paper. Only the second expansion rule (from [7]) is presented – it translates a feature starting with an all and containing an optional feature f into two different all features (enclosed in one-of): one with and one without the optional feature f.

$$[E2]\ \mathsf{all}(\mathsf{Ft},\mathsf{opt}(\mathsf{f}),\mathsf{Fs}) = \mathsf{one-of}(\mathsf{all}(\mathsf{Ft},\mathsf{f},\mathsf{Fs}),\mathsf{all}(\mathsf{Ft},\mathsf{Fs}))$$

Constraint satisfaction. This phase checks the conditions imposed, removing those configurations that do not satisfy the rules. The rules for the constraint operator requires are presented. Other constraint rules are described in [7].

$$[S3]\ \frac{\neg\mathtt{is\text{-}element}(\mathsf{A2},\mathsf{Fs}) \wedge \neg\mathtt{is\text{-}element}(\mathsf{A2},\mathsf{Fs}\prime)}{\neg\mathtt{satisfies}(\mathsf{all}(\mathsf{Fs},\mathsf{A1},\mathsf{Fs}\prime),\mathsf{Cs}\,\mathsf{A1}\,\mathsf{requires}\,\mathsf{A2}\,\mathsf{Cs}\prime)}$$

$$[S4]\ \frac{\mathtt{is\text{-}element}(\mathsf{A2},\mathsf{Fs}) \vee \mathtt{is\text{-}element}(\mathsf{A2},\mathsf{Fs}\prime)}{\mathtt{satisfies}(\mathsf{all}(\mathsf{Fs},\mathsf{A1},\mathsf{Fs}\prime),\mathsf{Cs}\,\mathsf{A1}\,\mathsf{requires}\,\mathsf{A2}\,\mathsf{Cs}\prime)} = \mathtt{satisfies}(\mathsf{all}(\mathsf{Fs},\mathsf{A1},\mathsf{Fs}\prime),\mathsf{CsCs}\prime)$$

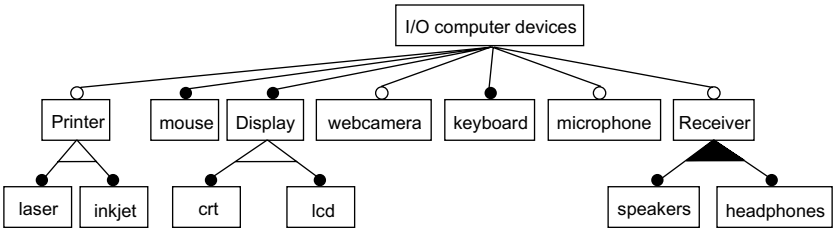

Fig. 1. Feature diagram for computer I/O devices.

3.2. Importance of the subject

Of course, the question might arise as to how representative FDL is as a domain-specific language. This question is hard to answer, since the term *representative* can be highly subjective. One way of determining how representative, is to compare it against other DSLs. The following DSLs were chosen for comparison with FDL:

- Production Grammars (PG) for software testing [28],
- A DSL that allows experimentation for the different regulation of traffic lights (RoTL) and supports the domain-specific analysis of junctions [18],
- Context-Free Design Grammar (CFDG),[1] designed for generating pictures from specifications. DSL domain is similar to Elliot's work [9],
- GAL, a well-known DSL used to describe video device drivers [31], and
- Extended BNF (EBNF), a standard notation used to unambiguously define language grammar.

Grammar size metrics defined by Power and Malloy [22] were used for comparison. They are the following:

- number of terminals (TERM) and non-terminals (VAR),
- McCabe cyclomatic complexity (MCC),
- Halstead effort (HAL), and
- average of right hand side size (AVS).

The number of productions (PROD) was also added to these metrics.

Despite being size metrics, TERM and VAR can nonetheless provide useful information about grammar. A greater maintenance effort might be required if a grammar has a large number of non-terminals (VAR). The McCabe cyclomatic complexity (MCC), when adapted for grammars, measures the number of alternatives for a grammars' non-terminals. A high MCC value denotes a large effort required for grammar testing and a greater potential for parsing conflicts. A big AVS value indicates a less readable grammar and may also impact performance for some parsers, since more symbols need to be placed on the parse stack. The Halstead effort metric (HAL) estimates the effort required to understand the grammar; the HAL number is directly proportional to the effort of understanding the grammar.

Grammar metric comparisons between FDL and other DSLs were obtained by *SynC tool* [22] and are summarized in Table 1. It can be seen that, against most metrics FDL is comparable to many of the above-mentioned DSLs (PG [28], RoTL [18], and EBNF). Of course, DSL complexity depends on the domain and can reach the complexity of some GPLs. For instance, domain-specific language GAL is comparable by MCC and HAL to the general-purpose language Oberon [26].

3.3. Conditions when applying comparison

The results from a family of experiments can be generalized when repetitive experiments can be proved. According to Basili et al. [1], repetition is strongly connected to a certain 'consensus' that is agreed upon before starting an experiment. In our study, consistency of results was obtained through the following comparison conditions for all DSL implementation approaches:

- DSL implementors were chosen according to their previous GPL experiences, and
- DSL implementations were reviewed by other experts in this field, to obtain code as optimal as possible.

Additionally, DSL implementors were advised:

- to follow the same design issues presented in the technical report [17] (DSL grammar, API that was given to DSL implementors by UML diagrams, etc.),
- to include minimum functionality for presenting the domain,
- to prepare DSL notation as optimally as possible for the end-users (we consider the notation as presented in Section 3 of the technical report [17], as optimal),
- to provide the same output for all compiled DSL programs, and
- to compile and run end-user programs from a shell.

For DSL end-users, the following rules were defined:

- DSL end-users are experienced GPL programmers, but have only general knowledge about using DSLs, and
- usage of the same IDE tools for writing end-user programs on specific DSL implementation.

3.4. Comparison validity

Making general conclusions on the basis of empirical study in the process of software development (and also other research fields) is extremely risky, whilst the reliability of results is hardly connected to the experimental

Table 1
FDL comparison to other DSLs

DSL	TERM	VAR	MCC	HAL	AVS	PROD
FDL	13	11	11	2948	3.91	22
PG	9	5	5	774	3.4	7
RoTL	23	12	6	3890	3.833	13
CFDG	28	12	28	6698	6.833	40
GAL	73	75	90	33296	3.853	131
EBNF	20	18	22	5523	3.778	28

[1] Context Free Design Grammars, available at http://www.chriscoyne.com/cfdg/index.php

environment and its context variables [1]. The threats to the validity of our comparison are given below.

3.4.1. DSL implementation approaches

DSL implementation approaches, together with subpatterns, are given in Section 2. The approaches in Table 2 were taken from among this list. This table also denotes the programming languages used by DSL implementation approaches, DSL implementors integrated development environment (IDE) and end-user environment. Few other approaches mentioned in Section 2 were inapplicable, due to the domain chosen, namely pipeline and lexical processing. The former is inapplicable because the FDL language cannot be divided into the core language and extensions which can be expressed by the core language. Hence, pipeline approach is inappropriate. The latter is inapplicable because FDL semantics is complicated enough that programs cannot be processed with simple lexical replacement. However, since most of the approaches were tested, we believed that this shortcoming is not a serious threat to the validity of our comparisons.

3.4.2. Functionality of DSL implementations

Comparison of DSL implementation approaches is possible only if the same functionality is provided. Many factors can affect comparability among various programming languages. For instance, API details were given to the DSL implementor by UML diagrams. These diagrams are useful accessories for those who use object-oriented languages, however they bring less information for those who use functional programming languages. Therefore, FDL design issues were also needed for functional programming languages (see [17], Section 4.3.). It is important to note that all the conducted implementations which were considered inappropriate for comparison (incomplete implementations, run-time failures, etc.) were removed from further investigations.

3.4.3. Programmers capabilities

Two options for implementing the DSL were considered at the beginning of this study. Either each programmer involved in this project writes all DSL implementations, or each implementation is done by the single most appropriate programmer, and later reviewed by senior researchers. The first possibility offers greater advantages: more implementations of the same DSL implementation approach can even-up the differences among individual programmers and can be statistically processed. However, a programmer with skills in six different programming languages and knowledge of specific programming techniques is hard to find. Therefore, the second option was chosen for the DSLs implementation phase. In our humble opinion, this decision does not affect or threaten the validity, since implementations were also reviewed by others. Therefore, the result obtained should be equal or better than a mean obtained from implementations conducted by several programmers. Another threat to validity is represented by single case studies. The results presented in this paper could become more reliable when similar results occur in other research with different contexts. Using these comparisons/experiments we can build up knowledge through a family of related experiments.

3.4.4. Reporting end-user efficiency

Experienced GPL programmers were included, in the end-users' effort study, but they had only very general experience about using DSLs. Diverse students were chosen from undergraduate to postgraduate. These programmers came with different backgrounds and knowledge, and could have affected the accuracy of the results. Therefore, this part of the study was statistically analyzed.

4. The study

In this section we compare committed DSL implementation approaches. This comparison offers empirical data on the effort involved using different DSL implementation approaches, and evaluates how friendly the implementations thus obtained are for the users. Quantitative results from implementation effort are presented as stacked bars. Significant observations can be obtained directly from these charts. The end-user friendliness of the resulting environment' is displayed as box plots: each of the horizontal lines representing a subset of end-user data conducted for specific implementation; the small gray circles represent individual data; the box indicates the middle

Table 2
DSL implementation approaches' comparison details

Approach	Language	DSL implementor execution platform	End-user execution platform
Source to source	Java	JBuilder 9	Text editor
	LISA	Lisa 2.0	Text editor
	Haskell	Hugs98	Text editor
Preprocessing	C++	Visual Studio 6.0	Text editor
Embedded	Haskell	Hugs98	Text editor
Handwritten interpreter	Java	JBuilder 9	Text editor
Compiler generator	LISA	Lisa 2.0	Lisa 2.0
	SmaCC	Visual Works 7.3	Text editor
Extensible compiler	C#	Mono 0.9	Text editor
COTS	XML	XML Spy	XML Spy

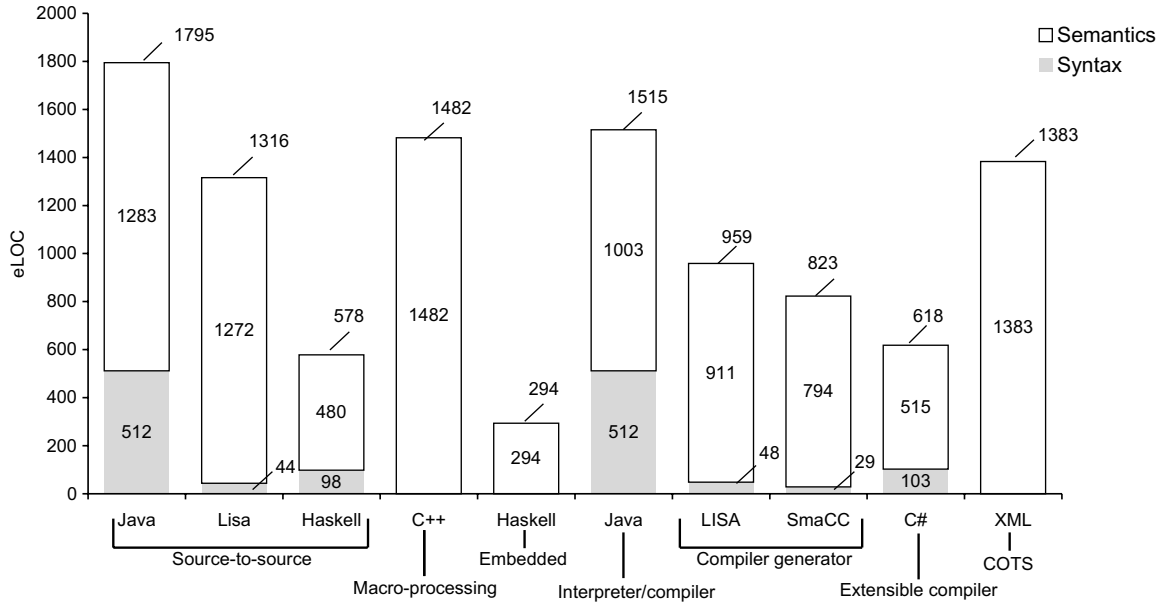

Fig. 3. Comparison of code in size.

section of data (between 25% and 75%); the big black dot stands for arithmetic mean. Some observations for end-user efficiency were also statistically tested. The ANOVA test (analysis of variance) [3] was used, which tests the differences between the observed samples. For the statistical significance of a test, the p value is substituted by words: *statistically independent* for $p < 0.01$ and *statistically dependent* for $p > 0.01$.

4.1. Implementors' effort

Analysis of implementation effort is a difficult task, when measuring DSL. Very different languages were used, from object-oriented and functional to imperative and hybrid. On the other hand, different approaches contain different sections of code – for example, while the interpreter/compiler approach can have hundreds of lines of code for syntactic analysis, the embedded approach will have none.

4.1.1. Comparisons between implementation approaches

The first and basic comparison describes the effort needed to implement a DSL; a rough measure of this effort can be obtained in terms of the number of lines of code (LOC). We measured the size of code with the numbers of effective lines of code (*eLOC*) – that is, all lines that are not blanks, standalone-braces, or parentheses.

The DSL development effort in terms of eLOC is presented in Fig. 3. Most of the approaches presented in this paper require some kind of syntactic and semantic analysis for the problem domain. Therefore, the language processing effort is divided into syntactical and semantical parts, represented in *Syntax* and *Semantics* stacked-bars. In the *Syntax* bar (gray color) we present the eLOC of the tokenizer and parser. Semantic equations are included in *Semantics* bar (white color). In these cases where there is no need for syntactic analysis – such as in the embedded approach – the corresponding bar in the graph is left empty.

From the eLOC point of view, it takes less effort to implement DSLs by embedding, followed by source-to-source (Haskell [24]), extensible compiler (C# [30]), and compiler generators (SmaCC,[2] LISA [20]). Much greater effort is required when DSL is implemented using COTS (XML [25]), macro processing, interpreter/compiler or source-to-source (Java [12]). Looking at the total number of lines for each approach, it can be observed that the embedded approach using Haskell requires a much less amount of coding than the rest.

Measuring eLOC is connected to the language chosen because some languages are more expressive than others and need less lines of code to express the same thing. Jones [14] defined, for each language, the average source statements for the same functionality – a *language factor* (see legend in Fig. 4). For example, on average, 38 lines of code are needed to implement an equivalent functionality in Haskell, while Java requires 53 lines of code.

Some of the languages used in this work (LISA, SmaCC) are absent in Jones's work [14]. We took the closest programming language/tool available for each of the used languages in our comparison. For instance, for SmaCC we used Smalltalk language factor, since semantics is given by Smalltalk statements. In Fig. 4, XML is denoted as *N/A* (not-available), since the language factor is unknown. The ratio of actual lines of code (eLOC) with the language factor was computed in order to eliminate

[2] SmaCC – parser generator for Smalltalk, available at http://www.refactory.com/Software/SmaCC/

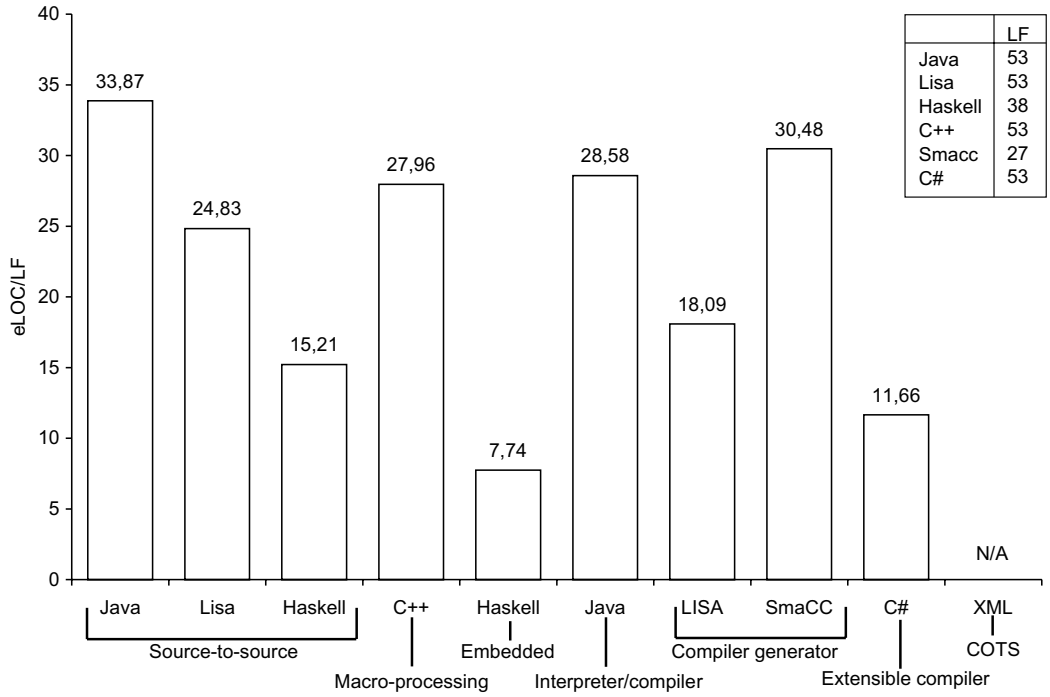

Fig. 4. Comparison of codes after normalizing with the language factor.

language expressiveness. Despite the language expressiveness being eliminated, the results are similar to those presented in Table 3. Less effort (smaller ratio) was required when DSL was implemented using Haskell as host/base language, extensible compiler/interpreter and compiler generator approach (LISA). Interesting insight about compiler generators can be deduced from Figs. 3 and 4, where SmaCC and LISA are confronted. SmaCC is Yacc[3] implementation integrated into the Smalltalk development environment, with very basic functionality. On the other hand, LISA is a compiler generation-dedicated tool, where special emphasis is placed on the modularity, extensibility, and reusability of language specifications. Both implementations contain similar eLOC. Implementation on LISA scores much better, after normalization using the language factor. We can conclude that the above-mentioned features of LISA increase programmers' productivity. On the other hand, using automatically generated tools to work with code [11], such as language knowledgeable editor (an editor that is aware of the regular definitions of the lexicon) and various inspectors (e.g., finite state automata visualizer, syntax and semantic tree animators) are very helpful in the process of language definition.

4.1.2. Turning API to DSL

Application libraries are serious competitors to DSLs. In combination with an application library, any GPL can act as a DSL. However, the shortcomings of application libraries such as limited domain-specific notations and the inability of domain-specific analysis, verification, optimization and transformation restrict their usefulness. In most cases, we never get beyond the application library stage despite the numerous advantages of DSLs [19]. One of the reasons is the lack of reliable knowledge about DSLs. Therefore, it is interesting to investigate how much effort is needed to turn an API into a DSL. To achieve this, we split the implementation into two parts (Fig. 5): the API (white bar in Fig. 5) and the rest of the work (*DSL*, denoted as a shaded bar).

The common belief is that turning API into DSL takes a lot of additional work (see Section 1). A first glance at Fig. 5 might confirms this. However, a closer look reveals some interesting insights. Source-to-source and compiler/interpreter approaches require much more additional work when turning API into DSL. On the other hand, embedded and COTS (XML) approaches can already be seen as a basic form of application libraries, with many of the already mentioned shortcomings of APIs. Therefore, we decided to categorize the approaches into two categories. The first category contains source-to-source and interpreter/compiler approaches. The second category covers the macro-processing, compiler-generators and extensible compiler approaches. The non-categorized are embedded and COTS (XML) approaches. Comparing categories 1 and 2 in Table 3 it can be concluded that, if the proper implementation approach is used (macro processing, compiler generator, and extensible compiler) the DSL implementor can gain in effort, since selecting a proper implementation approach on FDL gave smaller mean ($M = 88.00$) and standard deviation ($Std.\ dev. = 46.93$) in terms of eLOC than selecting other approaches

[3] Yacc – parser generator, available at http://dinosaur.compilertools.net

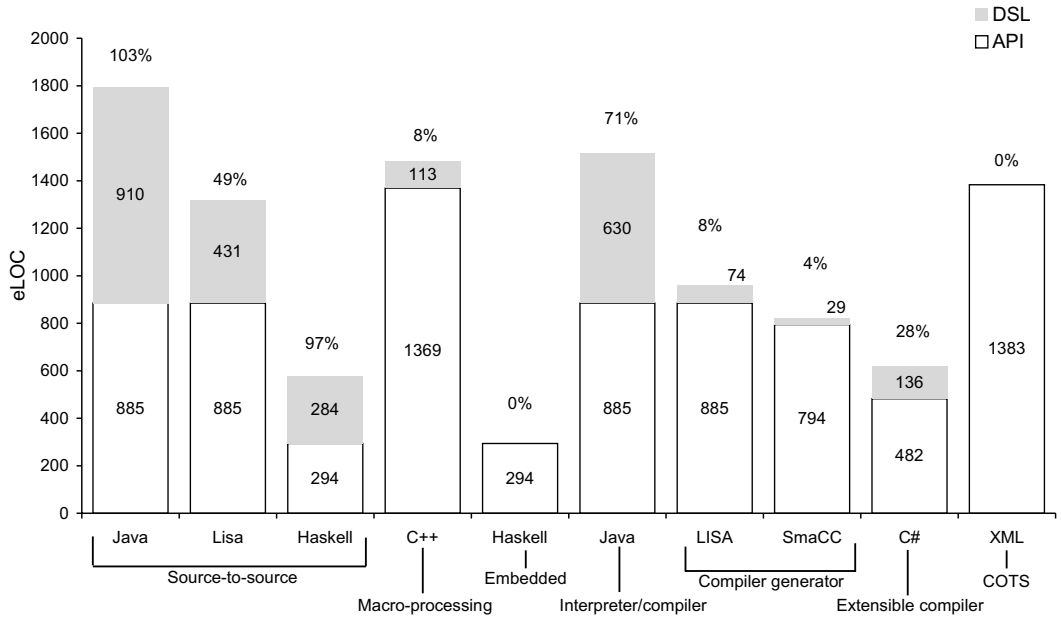

Fig. 5. Comparison of APIs with DSLs.

($M = 563.75$, *Std. dev.* $= 270.90$). Additionally, statistical tests confirm that variations in both categories vary drastically.

Fig. 5 also shows the ratio between API and DSL parts (see percentages above the stacked bars). For instance, in the handwritten compiler/interpreter approach 71% additional code was needed to obtain a DSL. The implementation cost can be modest using compiler generators (e.g., when using LISA only 8% additional work was needed).

4.2. End-user effort

4.2.1. End-user program length

As previously mentioned, APIs are serious competitors to DSLs. Domain-specific functionality in API is simply achieved through object creation and method invocation (if object-oriented GPL is used), while domain-specific notation is usually sacrificed. This can be clearly seen by comparing a GPL program using the API (Fig. 6) and a DSL program (Fig. 2) using the original domain notation. The advantages of DSLs compared to API are obvious in respect of eLOC and understandability. Both programs (Figs. 2 and 6) describe a simple computer hardware composition from which different valid system configuration can be obtained. For details about the FDL please refer to Section 3.1.

Table 3
Comparison of code after categorization

Category	*N*	DSL mean	*Std. dev.*
1	4	563.75	270.90
2	4	88.00	46.93

Using source-to-source (Java, LISA, Haskell), interpreter/compiler (Java), and compiler generator (LISA, SmaCC) we were able to achieve the requested original notation (Fig. 2), while in other approaches we experienced some drawbacks: we were unable to express the requested DSL notation due to the reuse of language facilities or simply to follow the rules of language used. For example, in macro-processing (Fig. 7) instead of a comma, the operator '|' is used as a list separator. The overloaded comma operator can affect ordering properties and this can confuse users. Therefore, avoiding overloading comma is recommended. Moreover, the order of features is reversed (from composite features – e.g., Receiver, Display to main concept IOdevice). This is because of the C++ language restriction that macro definition must appear before the use of a macro. On the other hand, atomic features have to be expressed as string using quotes.

When embedding the language FDL into Haskell, we found that most of the work has already been done by the API – so-called *combinator library*, that is used for expressing the syntax and semantics constructs of a given domain. Fig. 8 presents the complete end-user program for the IODevice. Some observations about notation follow. Instead of using atomic features with a name, we use the representation of them as strings (literals between double quotes "), and the operation `atomic`, that transforms a string into an atomic feature. This is why the FDL expression mouse is written `atomic "mouse"` in the program. The second observation about notation is that, in Haskell, function names can be converted into infix notation by using *backquotes* (‘). This explains the definition of `constraintl`. Note also that list notation (square brackets) is different than that requested by the original

```
boolean norm = true;
//First feature definition ->
  // IOdevice : all (opt(Printer), mouse, Display, opt(webcamera), keyboard,
  // opt(microphone), opt(Receiver))
FeatureList ioList = new FeatureList(norm);
  // opt(Printer)
Feature printerName = RuleManager.buildAtomicFeature("Printer");
Feature printerOpt  = RuleManager.buildOptionalFeature(printerName,norm);
ioList = RuleManager.buildFeatureList(ioList,printerOpt,norm);
  // mouse
Feature mouseAF = RuleManager.buildAtomicFeature("mouse");
ioList = RuleManager.buildFeatureList(ioList,mouseAF,norm);
  // Display
Feature displayName = RuleManager.buildAtomicFeature("Display");
ioList = RuleManager.buildFeatureList(ioList,displayName,norm);
  // opt(webcamera)
Feature webcamAF  = RuleManager.buildAtomicFeature("webcamera");
Feature webcamOpt = RuleManager.buildOptionalFeature(webcamAF,norm);
ioList = RuleManager.buildFeatureList(ioList,webcamOpt,norm);
  // keyboard
Feature keyboardName = RuleManager.buildAtomicFeature("keyboard");
ioList = RuleManager.buildFeatureList(ioList,keyboardName,norm);
  // opt(microphone)
Feature micAF  = RuleManager.buildAtomicFeature("microphone");
Feature micOpt = RuleManager.buildOptionalFeature(micAF,norm);
ioList = RuleManager.buildFeatureList(ioList,micOpt,norm);
  // opt(Receiver)
Feature recName = RuleManager.buildAtomicFeature("Receiver");
Feature recOpt  = RuleManager.buildOptionalFeature(recName,norm);
ioList = RuleManager.buildFeatureList(ioList,recOpt,norm);
  // IOdevice : ...
Feature ioFeature    = null;
Feature ioAllFeature = RuleManager.buildAllFeature(ioList,norm);
ioFeature            =
  FeatureDefinition.addFeature(ioFeature,ioAllFeature,
       (AtomicFeature)RuleManager.buildAtomicFeature("IOdevice"));

<...>
Feature printerF = RuleManager.buildOptFeature(printerList,norm);
ioFeature        =
  FeatureDefinition.addFeature(ioFeature, printerF,
       (AtomicFeature)RuleManager.buildAtomicFeature("Printer"));

<...>
```

Fig. 6. IOdevice program using API.

```
BeginSpecification(IODeviceSpec)
BeginFeature
    feature Receiver = more_of( "speaker" | "headphones"      )
    feature Display  = one_of ( "crt"     | "lcd"             )
    feature Printer  = one_of ( "laser"   | "inkjet"          )
    feature IOdevice = all  ( opt(Printer)| "mouse"
                            | Display     | opt ("webcamera"  )
                            | "keyboard"  | opt ("microphone" )
                            | opt (Receiver)                  )
EndFeature
BeginConstraints
    constraint "webcamera" requires "microphone"
    constraint include "lcd"
EndConstraints
EndSpecification
```

Fig. 7. C++ code (macro processing approach) representing the example for IOdevice.

notation. Moreover, in several places definitions from API (e.g., Feature, Constraint, Spec) have to be used, which can be hard to understand for an end-user programmer without experience in functional programming.

Similar observations can be noticed for the extensible compiler approach (Fig. 9) where class definition is extended with domain-specific notations (features and constraints). Here, the end-user programmer needs to be familiar with the method Main(), passing results as parameters (strResult), using packages (System), etc.

```
ioDevice, printer, display, receiver :: Feature f => f
ioDevice  = all [opt (printer)     , atomic "mouse"
                , display          , opt (atomic "webcamera" )
                , atomic "keyboard", opt (atomic "microphone")
                , opt (receiver) ]
printer   = one_of [ atomic "laser", atomic "inkjet"         ]
display   = one_of [ atomic "crt"  , atomic "lcd"            ]
receiver  = one_of [ atomic "speakers", atomic "headphones"  ]

constraint1, constraint2 :: Constraint
constraint1 = "webcamera" `requires` "microphone"
constraint2 = include "lcd"

spec :: Feature f => Spec f
spec = Spec ("IOdevice" , ioDevice )
            [("Printer" , printer ),
             ("Display", display ),
             ("Receiver", receiver) ]
            [ constraint1, constraint2 ]

 <...>
```

Fig. 8. Haskell code (embedded approach) representing the example for IOdevice.

```
using System;
class hello_IOdevice {

begin_spec(IOdevice,strResult)
   begin_features()
     feature IOdevice = all     ( opt(Printer), "mouse"          ,
                                  Display     , opt("webcamera") ,
                                  "keyboard"  , opt("microphone"),
                                  opt(Receiver)                  )
     feature Printer = one_of   ( "laser"     , "inkjet"         )
     feature Display = one_of   ( "crt"       , "lcd"            )
     feature Receiver = more_of ( "speaker"   , "headphones"     )
   end_features()

   begin_constraints()
     constraint "webcamera" requires "microphone"
     constraint include     "lcd"
   end_constraints()
end_spec()

static void Main(){
    Console.WriteLine (strResult);
}
}
```

Fig. 9. C# code (extensible compiler approach) representing the example for IOdevice.

It can be claimed that implementations from Figs. 7–9 contain just minor deviations from the requested notations (see Fig. 2). However, as will be presented later, these shortcomings have a big impact on end-user productivity. These implementations are definitely much better than the COTS (XML) approach (Fig. 10), where the obtained notation is cumbersome to write and read.

eLOC for 14 FDL programs was measured (Fig. 11) in order to have a rough idea of how different the resulting code might be. It can be observed that DSL programs in some approaches (COTS, API, and the embedded approach) need many more lines of code in order to write

```
<featureDiagram>
  <featureDefinition>
    <featureName>IOdevice</featureName>
  <featureExpression>
    <all>
      <featureList>
        <featureExpression>
          <option>
            <featureExpression>
              <featureName>Printer</featureName>
            </featureExpression>
          </option>
        </featureExpression>
        <featureExpression>
          <atomicFeature>mouse</atomicFeature>
        </featureExpression>
        <featureExpression>
          <featureName>Display</featureName>
        </featureExpression>
        <featureExpression>
          <option>
            <featureExpression>
              <atomicFeature>webcamera</atomicFeature>
            </featureExpression>
          </option>
        </featureExpression>

  <...>
```

Fig. 10. XML code (COTS approach) representing the example for IOdevice.

specific FDL programs. For example, the embedded approach needed almost 5 times more lines of code to express FDL programs. Additionally, Fig. 11 confirms the advantages of DSLs compared to API solution, since DSL programs were at least 8 times shorter for the observed FDL domain. Note, that in this observation, the COTS (XML) approach was excluded.

The average program length of the observed DSL programs in Fig. 11 is 11.25 eLOC. The mean was much bigger for some implementation approaches. On the other hand, the middle half confirms that, whilst DSL programs increase in size, the variation of DSL programs stays the same for these approaches (macro processing, embedded, and extensible compiler approaches). Therefore, the ANOVA test supports the fact that DSL end-user effort (in terms of eLOC) is statistically independent of their chosen implementation approach.

However, the eLOC metric does not contain enough information about program differences. As denoted in Figs. 7–10, solutions can be very different from what eLOC measurement expresses. In order to evaluate these differences, a similarity factor was used (Fig. 12) between the obtained and desired program notations. The idea here was to use a known plagiarism-detection system. Many software similarity systems exist, such as MOSS [27], however these systems use some kind of syntax analyzers, which makes them very useful for detecting copies when known GPL is used. Since notation differs in our DSLs, we had to use a context independent copy detection system. One we know well is integrated into our information system and used for educational purposes [21]. This similarity system uses fuzzy logic. This system was used to compare all programs with the original FDL notation. Similarity results show how much specific notation differs from the original FDL notation. One hundred percentage of similarity with original FDL notation was achieved in the source-to-source, interpreter/compiler, and the compiler generator. In all other approaches (macro processing, extensible compiler, embedded, and COTS) the similarity is not as good as that presented with eLOC measurement – the similarity is around or less than 50%.

This study also measured the time to successfully write and execute FDL programs, where correct results must be obtained. End-users used different tools (see Table 2) for specific implementation (e.g., XML-Spy, LISA IDE) in order to accomplish this task. The results are presented in Fig. 13. Differences in notation resulted in bigger end-user effort times. All approaches, where original notation was not achieved (XML, embedded, extensible compiler, macro-processing), resulted in worse end-user productivity. Even when taking into account that, for example, a special tool was used when using XML. On the other hand, there was still a difference in approaches where 100% similarity with original notation was achieved (source-to-source, interpreter/compiler, compiler generator). Since end-users were using an integrated development environment with a language knowledgeable editor in the LISA compiler

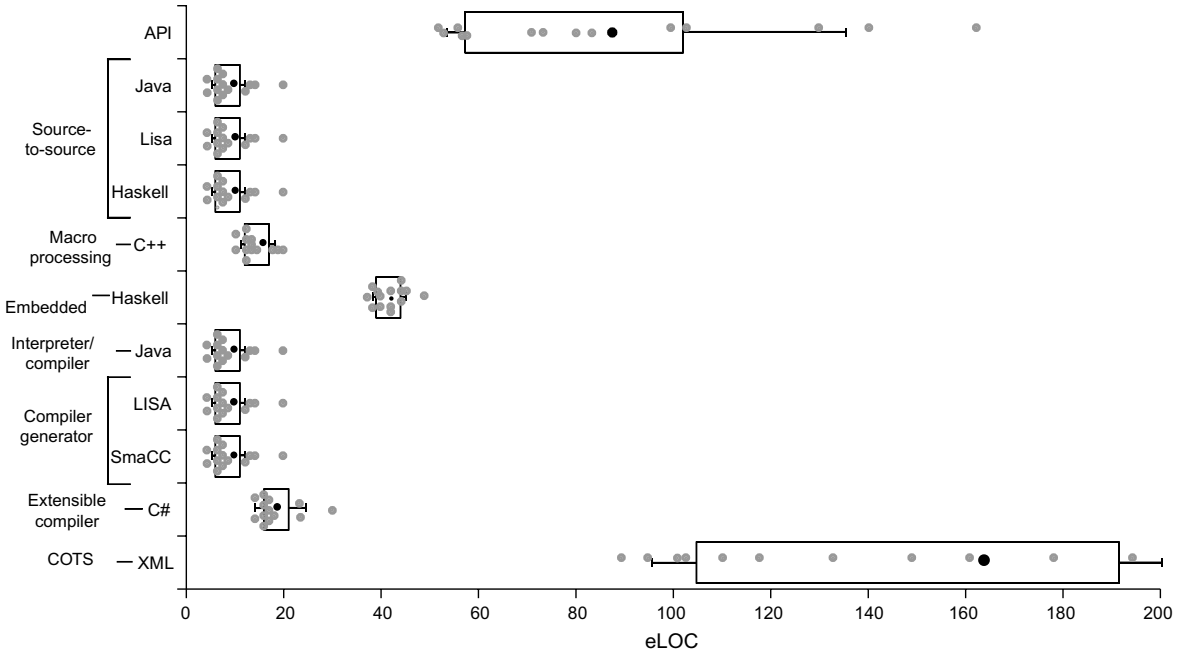

Fig. 11. Comparison of DSL program sizes (eLOC) together with API ($N = 14$).

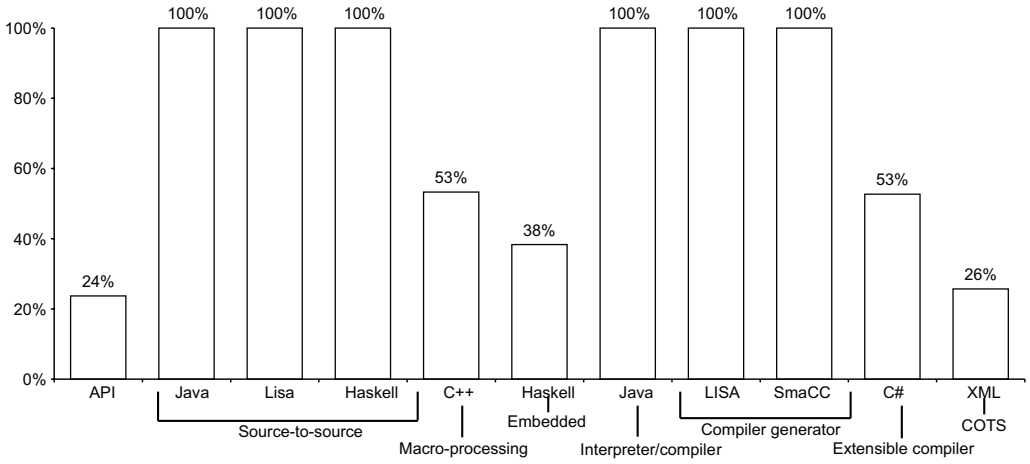

Fig. 12. Similarity comparison of DSL program

generator, they were most efficient when using this approach. This is further evidence that end-user productivity can be enhanced by proper tools. But, good tools alone (e.g., XML Spy) are not enough if notation is compulsory. The importance of syntax [19] should not be underestimated.

A couple of first-year students of computer science with no prior knowledge of DSLs were selected in order to obtain results on error reporting. After explaining the basic notions of FDL to them, they were challenged to write a couple of examples, and report the results. This experiment was repeated with young researchers, teaching assistants, and other laboratory members at the University of Maribor, and the combined results are presented in Fig. 14. They were asked to measure the debugging facilities and error reporting understandability for each approach. A five-graded scale, going from very bad to very good, was used to measure these subjects. It can be concluded from Fig. 14 that better error reporting (mean > 3) was achieved in the extensible compiler, interpreter/compiler and compiler generator approach (LISA).

Last, but not least, performance comparison between different DSL implementation approaches was included in this study. Performance issues might be very important for some domains (e.g., embedded systems). For the performance test, a figurative FDL program was invented for the purpose of performance testing. The executed program has 20 lines of code and the corresponding expanded version contains 11,880 configurations. Additionally we also measured other programs but since the results were similar, they were excluded from Table 4.

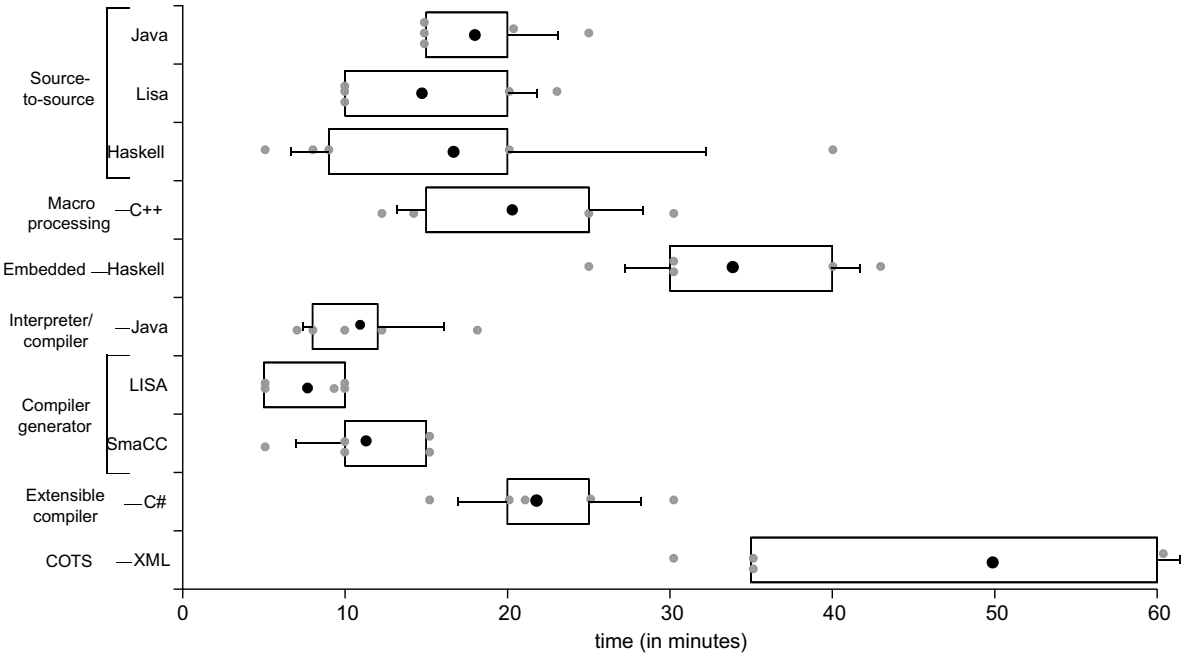

Fig. 13. End-user time effort to implement a DSL program ($N = 5$).

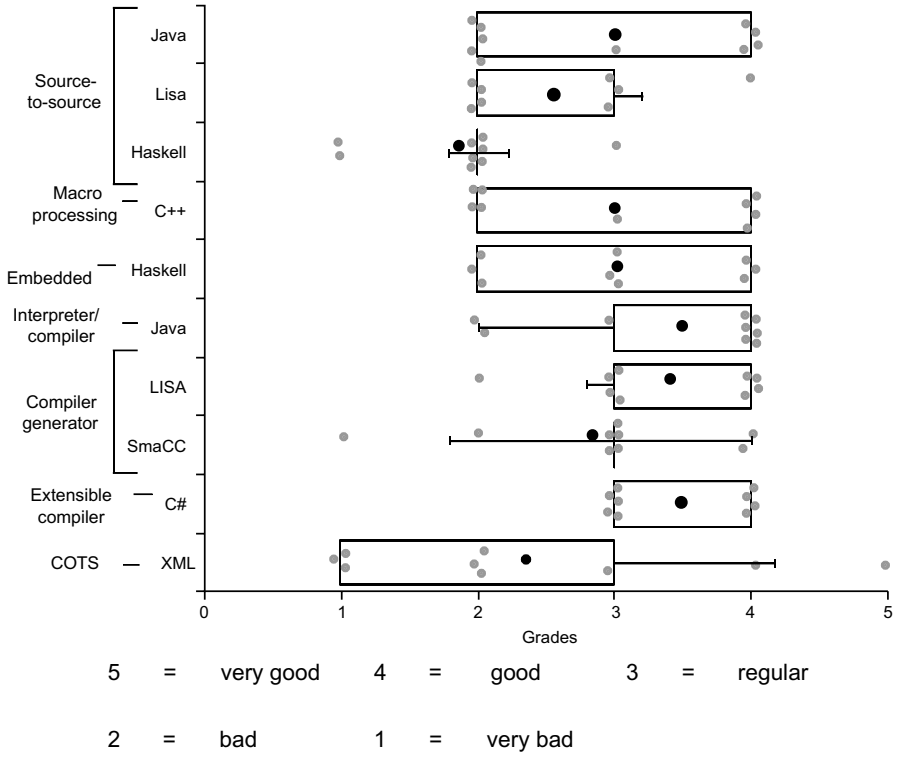

Fig. 14. End-user experience on provided DSL debugging and error reporting ($N = 9$).

The compilation process was carried out using standard compilers of each base/host language (e.g., ghc for Haskell, Mono for C#, javac for Java based approaches, etc.). We took this decision based on the idea that most of DSL implementors would use standard compilers and tools. Additionally, we compiled each code with no additional parameters. All approaches were tested on the same computer, run from the console and the average of three tests was used for each approach. The FDL program is read from the file (without user interaction), executed, and the transformation results stored in a text file. The fastest DSL translator was obtained using the extensible compiler approach, followed by translators written in Haskell as host/base language, macro processing, and translators

Table 4
Comparison of DSL program time performance

Approach	Language/Tool	Time
Source-to-source	Java	7min 41s 021 ms
	Lisa	5min 23s 075ms
	Haskell	8s 980ms
Macro processing	C++	57s 887ms
Embedded	Haskell	5s 756ms
Interpreter/compiler	Java	7min 30s 951ms
Compiler generator	LISA	5min 49s 902ms
	SmaCC	7min 59s 257ms
Extensible compiler	C#	119ms
COTS	XML	N/A

implemented in Java. On the other hand, the COTS approach is the only approach that did not return a result for transformation, although the program was running for several hours. Usually, if not always, compiled code is more efficient than the interpreted code and this must be taken into account by the implementor if efficiency is an important issue. The performance of interpreter approach can be speeded up by using various techniques (e.g., partial evaluation and program specialization [5], and compiling embedded languages [8]).

4.3. Comparison remarks

A summary for these comparison of DSL implementation approaches from Figs. 3–5, 11–13, and 14 are presented in Table 5, where rankings were used for individual comparison.

Comparing DSL implementation approaches is a hard task. Throughout the study two prospects on comparisons regarding DSL implementation approaches were shaped: implementation and end-user effort. Standard metrics were used (eLOC, FP) to achieve fair comparisons among different implementation approaches. These standard metrics show that the most efficient way, in terms of lines of code, to implement a DSL is the embedded approach (Table 5). In the future our task is to find some additional DSL – specific metrics which could better express the differences between approaches.

From the DSL end-user point of view, our comparison shows that original notation (Table 5) was fully achieved in some implementation approaches (ranking 1, columns D and E of Table 5), with positive influence on end-user productivity (column F, Table 5). Moreover, understandability of notation, programming interface, debugging, and error reporting are unsatisfactory in some implementation approaches (COTS, source-to-source, embedding).

Finally, which implementation approach gives the best solution? To answer this question we accumulated rankings from Table 5. However, such rankings must be used with extreme care and should not be generalized in all situations. In practice, several parameters need to be considered, such as implementation language, developer experience, end-user background, DSL performance (compilation, verification, optimization), time-to-market, and other critical issues.

5. Conclusions

The meaningful insight of this paper is to offer empirical data involving ten DSL implementation approaches, and statistical data about the user-friendliness of these implementations. This paper describes a comparison based on the same case study (FDL), where certain anecdotal claims from literature were taken into consideration. Our key findings are listed below:

- Standard metrics were used (eLOC, eLOC/LF) to achieve fair comparisons among different implementation approaches. These standard metrics show that the most efficient way, in terms of lines of code to implement a DSL, is the embedded approach. Normalizing with the language factor did not change this fact.
- The comparison section also shows that turning API into a DSL is not that difficult in terms of additional eLOC if the proper implementation approach (compiler generator, macro processing, extensible compiler) is chosen.

Table 5
Ranking of DSL implementation approaches

Approach	Language	Implementation effort			AVG1	End-user effort				AVG2
		A	B	C		D	E	F	G	
Source-to-source	Java	10	9	10	9	1	1	6	4	3
	Lisa	6	5	7	6	1	1	4	8	5
	Haskell	2	3	9	3	1	1	5	9	6
Macro processing	C++	8	6	4	6	7	7	7	4	7
Embedded	Haskell	1	1	1	1	9	9	9	4	9
Interpreter/compiler	Java	9	7	8	8	1	1	2	1	1
Compiler generator	LISA	5	4	5	3	1	1	1	3	2
	SmaCC	4	8	3	5	1	1	3	7	3
Extensible compiler	C#	3	2	6	2	8	8	8	1	7
COTS	XML	7	N/A	1	N/A	10	10	10	9	10

A = Comparison of code in size; B = Comparison of codes after normalizing; C = Turning an APIs into a DSLs; D = Comparison of DSL program sizes (eLOC); E = Similarity comparison of DSL program; F = End-user time effort to implement a DSL program; G = End-user experience on provided DSL debugging and error reporting; AVG1 = Ranking on implementation effort; AVG2 = Ranking value on end-user effort.

- Original notation was hard to achieve in some implementation approaches (COTS-XML, embedding, macro-processing, extensible compiler) with consequences for DSL end-user productivity.
- Error reporting and debugging are unsatisfactory in some implementation approaches (COTS-XML, source-to-source, embedding).

Many authors [8,13,15] promote the embedding approach as the most appropriate one. According to implementation effort, our study supports this claim. However, the embedding approach has significant penalties when end-user effort is taken into account. We strongly believe that the effort needed by an end-user to rapidly write correct programs is, in many cases, more important than the effort required by a programmer to implement a DSL. Nevertheless, there is no straight relation between implementation and end-user effort in all situations. The decision in favor of any implementation approach depends on the context a DSL uses and depends on the languages, developers, end-users and critical issues such as performance, analyzability, time-to-market, etc. Our simplified advice to future DSL developers is: when small groups of users are going to use a new DSL (error reporting is not that important) and when notation should not be strictly obeyed, then the recommended approach is embedding. Otherwise, the recommended solution is to implement a full DSL compiler using compiler generators.

We consider that the results of the study are reliable despite the threats of validity. This work presents a preliminary study on the subject of DSL implementation approaches done on a representative language FDL. The experiment needs to be repeated on other domains, in order to obtain more general results. It is planned to revise this comparison using other representative languages, to see how it impacts on DSL development, and end-user effort. We hope that this work will stimulate other researchers contributions to this experiment. In order to implement our case study (FDL) we took domain analysis from [7]. Domain analysis is an important part of DSL development. In the future it is also planned to focus on domain analysis and DSL design, which is often argued as a more demanding task than implementation.

To the best of our knowledge, no other experiment nor comparison of DSL implementation approaches for the same case study appears in literature. Further research into this topic is necessary to advocate the strengths and weaknesses of various approaches and trade-offs between them for diverse domains. Results can then be generalized and become more reliable. The study presented is only a first step towards this goal.

Acknowledgements

Thanks to Jerónimo Irazábal, Diego Yanivello, Federico Feller, Mariana Báez, and David Krmpotić who helped with some of the implementations of FDL. We would like to thank James Power and Brian Malloy for giving use their tool SYNQ [22] for grammar metric calculation. The authors thank anonymous reviewers for their detailed and constructive comments that helped us to increase the quality of this work.

References

[1] V. Basili, F. Shull, F. Lanubile, Building knowledge through families of experiments, IEEE Transactions on Software Engineering 25 (4) (1999) 456–473.
[2] T.J. Bergin Jr., R.G. Gibson Jr., History of Programming Languages II, Addison-Wesley, 1996.
[3] G.E.P. Box, W.G. Hunter, J.S. Hunter, Statistics for Experimenters: Design, Innovation, and Discovery, second ed., Wiley, Interscience, 2005.
[4] M. Bravenboer, R. Vermaas, J. Vinju, E. Visser, Generalized type-based disambiguation of meta programs with concrete object syntax, in: Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05), Springer-Verlag, 2005, pp. 157–172.
[5] C. Consel, R. Marlet, Architecturing software using a methodology for language development, in: Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming, vol. 1490, September 1998, pp. 170–194.
[6] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, Reading, MA, 2000.
[7] A. van Deursen, P. Klint, Domain-specific language design requires feature descriptions, Journal of Computing and Information Technology 10 (1) (2002) 1–17.
[8] C. Elliott, S. Finne, O. de Moor, Compiling embedded languages, in: International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG'00), Springer-Verlag, Berlin, 2000.
[9] C. Elliott, An embedded modeling language approach to interactive 3D and multimedia animation, IEEE Transactions on Software Engineering 25 (3) (1999) 291–309.
[10] J. Heering, Application software, domain-specific languages, and language design assistants, in: Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000), 2000.
[11] P.R. Henriques, M.J. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, H. Wu, Automatic generation of language-based tools using the LISA system, IEE Software 152 (2) (2005) 54–69.
[12] C.S. Horstmann, G. CornellCore Java – Advanced Features, vol. 2, Sunsoft Press, Mountain View, CA, USA, 1998.
[13] P. Hudak, Modular domain specific languages and tools, in: Proceedings: Fifth International Conference on Software Reuse, IEEE Computer Society Press, MD, 1998, pp. 134–142.
[14] C. Jones, Applied Software Measurement: Assuring Productivity and Quality, second ed., McGraw-Hill, NY, 1997.
[15] S. Kamin, Research on domain-specific embedded languages and program generators, Electronic Notes in Theoretical Computer Science 12 (1998).
[16] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
[17] T. Kosar, P.E. Martínez López, P.A. Barrientos, M. Mernik, Experiencing diverse implementation approaches for domain-specific languages. Technical report, University of Maribor, National University of La Plata, 2005. FERI report RAJ-T0501 <http://marvin.uni-mb.si/technical-report/RAJ-T0501.pdf>.
[18] S. Mauw, W.T. Wiersma, T.A.C. Willemse, Language-driven system design, International Journal of Software Engineering and Knowledge Engineering 6 (14) (2004) 625–664.

[19] M. Mernik, J. Heering, A. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344.
[20] M. Mernik, V. Žumer, M. Lenič, E. Avdičaušević, Implementation of multiple attribute grammar inheritance in the tool LISA, ACM SIGPLAN Notices 34 (6) (1999) 68–75.
[21] M. Lenič, J. Brest, E. Avdičaušević, M. Mernik, V. Žumer, Information system for laboratory work management, in: Proceedings of the 5th Euromedia Conference 2000 (Euromedia'2000), SCS Europe BVBA, cop., 2000, pp. 245–249.
[22] J.F. Power, B.A. Malloy, A metrics suit for grammar-based software, Journal of Software Maintenance and Evolution: Research and Practice 16 (6) (2004) 405–426.
[23] L. Prechelt, An empirical comparison of seven programming languages, IEEE Computer 33 (10) (2000) 23–29.
[24] S.P. Jones, J. Hughes (Eds.), Haskell 98: a non-strict, purely functional language, February 1999. Available at: <http://www.haskell.org/onlinereport/>.
[25] E.T. Ray, Learning XML. A Nutshell Handbook, O'Reilly & Associates, Inc., USA, 2001.
[26] M. Reiser, N. Wirth, Programming in Oberon – Steps Beyond Pascal and Modula, first ed., Addison-Wesley, Reading, MA, 1992.
[27] S. Schleimer, D.S. Wilkerson, A. Aiken, Winnowing: local algorithms for document fingerprinting, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, NY, 2003, pp. 76–85.
[28] E.G. Sirer, B.N. Bershad, Using production grammars in software testing, in: Proceedings of the 2nd Conference on Domain-Specific Languages, pp. 1–14. USENIX Association, 1999.
[29] D. Spinellis, Notable design patterns for domain-specific languages, The Journal of Systems and Software 56 (1) (2001) 91–99.
[30] T. Thai, H.Q. Lam, .NET Framework Essentials. A Nutshell Handbook, third ed., O'Reilly & Associates, Inc., USA, 2003.
[31] S. Thibault, R. Marlet, C. Consel, Domain-specific languages: from design to implementation – application to video device drivers generation, IEEE Transactions on Software Engineering 25 (3) (1999) 363–377.
[32] Categorized Lists of Computer Programming Languages. Available at: <http://en.wikipedia.org/wiki/list_of_programming_ languages>.
[33] Collection On Computer Programming Languages. Available at: <http://www.people.ku.edu/nkinners/langlist/extras/ langlist.htm>.