An Evaluation of Domain-Specific Language Technologies for Code Generation

Christian Schmitt[‡], Sebastian Kuckuk[†], Harald Köstler[†], Frank Hannig[‡], and Jürgen Teich[‡]
[‡]Hardware/Software Co-Design, Department of Computer Science
[†]System Simulation, Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—Software systems are becoming increasingly complex, requiring a deep knowledge to work and program with them. This is especially true for simulation frameworks used by scientists and engineers, but also applies to completely different domains such as mobile or web applications. To ease working with these systems, Domain-Specific Languages (DSLs) are a convenient way to enable domain experts describe settings and problems they want to solve using terms familiar to them. Building upon this specification in the DSL, a compiler transform this to the target software framework, e. g., runnable program code. To write such a compiler, a solid implementation framework is needed. In this paper, we propose criteria for the evaluation of textual programming language implementation frameworks to which we accordingly evaluate four technologies, namely Spoofax/IMP, Rascal MPL, a custom approach using C++ and a custom approach using Scala.**

I. Introduction

Mapping algorithms to a target system in an efficient way is a cumbersome task that not only takes a great amount of domain knowledge, but also requires insight into the algorithms and their implementations as well as the target system the code is to be run on. A very good example scenario for this phenomenon is the HPC domain. Already today's supercomputers employ multiple processor architectures and accelerator technologies like GPUs or Field Programmable Gate Arrays (FPGAs) to improve power efficiency and keep operation costs down. This trend towards increasingly parallel and heterogeneous systems results in a large range of software technologies, programming techniques and optimizations that have to be employed and applied by programmers to get the best performance. This, in turn, decreases performance portability, as techniques working on a certain system may be counter-effective on another. As implementing these optimizations is an error-prone and time-intensive task, it also lowers programmer productivity in general.

A solution to these problems is the use of Domain-Specific Languages (DSLs). They allow domain experts to describe a problem they want to solve, or, more general, a setting, in a declarative, abstract and compact way using concepts and terms that are familiar to them. Because no implementation details need to be specified, parallelization may be supported by a compiler, exploiting generic parallel execution patterns in the domain at different abstraction levels. Since all the details are derived by the compiler, DSL programs are much more portable and scalable: Support for new hardware features only takes an extension of the DSL compiler, not of all the DSL programs. For the transformation of a DSL program to a runnable program, the compiler may use three different sources of knowledge that assist in choosing correct and adequate optimizations:

- The domain expert gives hints which algorithms are best suited for a certain task. The compiler uses this knowledge by applying corresponding transformations, thereby replacing the abstract declarations by concrete algorithms.
- The hardware architecture expert knows about the target system the program is to be run on. This knowledge effects the realization of the previously chosen algorithms, e. g., which technologies will be used by the algorithm implementation.
- The programming expert knows which techniques to apply for getting the best performance out of an implementation. These optimizations techniques need to be applied to the previously selected implementation by the compiler.

All this information may be provided in different ways, either as part of the DSL program, separate specification files or built right into the compiler framework.

Various approaches to ease the development of DSLs have been created. Among these are language workbenches, a term coined by Martin Fowler in 2005 [5]. Characteristic about a language workbench is the ability to edit the abstract representation of a program, opposed to the conventional way of manipulating text files to modify a program. In this traditional approach, the abstract representation is generated by the compiler or interpreter while reading the text file and is lost after the translation is done. Examples of language workbenches are *Xtext* and *Spoofax/IMP*, of which the latter has been chosen for evaluation in this paper as it is of academic origin. Another approach to the creation of DSLs is the use metaprogramming languages like *Jetbrains MPS*, *MetaEdit+* or *Rascal*. For presentation in this paper, Rascal has been chosen because of its promising feature set and its academic origin. The most straight-forward approach to the implementation of a DSL for a HPC developer is the use of a general-purpose purpose language. Thus, C++ and

Scala have been selected for evaluation in this paper, the first one representing principally object-oriented, the second one representing principally functional programming languages.

We build on experiences gained in our previous work *HIPAcc*[1] [19]. HIPAcc is a DSL embedded into C++ (see section II for an explanation of the term *embedded DSL*) for 2D stencil operations coming from the domain of medical image processing. Although the language has been extended for solution of elliptic PDEs with multigrid methods [20] via a specialized syntax, we need to raise the abstraction level to allow more advanced optimizations. By decoupling the abstract algorithm description from the implementation, we gain the ability to automatically choose the best suited components for the given algorithm and the targeted hardware platform. As we plan to evaluate a broad range of optimization strategies, we require a high flexibility as well as a powerful internal semantic model of our language and its applications, which usually come with big trade-offs when using internal DSL. We therefore decided to pursue the implementation of an external DSL. Our language is to be textually represented, which we consider more natural for the definition of mathematical algorithms.

For the evaluations, we decided to invest three weeks' time per technology to learn about its concepts and create a prototype. We targeted a simple numerical code to be generated. As a first step, a DSL file containing the definition of the computational domain and variables, boundary conditions, mathematical operators and of basic statements like loops, conditionals, input/output and mathematical operations had to be read. In subsequent steps, this was transformed by processing variable and computational domain definitions into array definitions, operators into functions, etc. In the end, the resulting structure was printed out into C++ code.

In the next section, some theoretical background and basics will be explained. Section III gives a brief overview of related work and the main contributions of this paper are highlighted. In Sections IV to VII, four technologies for the creation of DSLs are presented. A rating system for language creation technologies is proposed and used in Section VIII. This will lead to the conclusions in Section IX and future work to be done is described in Section X.

II. Theoretical Background and Basics

Based on the language definition, there are two categories of DSLs: *internal* (*embedded*) and *external* DSLs. Languages of the first category use the syntax of a host general-purpose programming language and extend or restrict it by introducing new domain specific language elements like special data types, routines or macros. As external DSLs introduce a completely new syntax and semantics, in general, they are more flexible and expressive than internal ones at the cost of a higher design effort.

DSLs, just like general-purpose programming languages, may be represented in a textual or in a visual way. Very often, however, visual programming languages like LabVIEW[2] and Simulink[3] are DSLs in reality. Their visual representation often is the most natural way of describing scenarios in certain domains and thus, in the spirit of the intended use, further eases development for domain-experts.

Context-sensitive grammars are the base for *context-sensitive* languages. These grammars contain rules that have symbols on the left-hand as well as the right-hand side. *Context-free* grammars allow only a single symbol on the left-hand side. A *symbol* in this case basically means a string, consisting of one or more characters. In layman's terms, in a context-sensitive language, a symbol's meaning is dependent of the *context* it appears in.

Grammars are usually defined using a textual representation that is called *Backus-Naur Form (BNF)*. It consists of a number of rules (*productions*), where certain characters convey a certain meaning, e. g., the character | represents alternatives. The Extended Backus-Naur Form (EBNF) is a formalization of the BNF for the representation of programming language. A small example for the definition of arithmetic expressions could look like this:

```
assignment    = identifier "=" expr
expr          = number | binaryOpExpr
binaryOpExpr = expr binaryOp expr
binaryOp      = "+" | "-"
```

In this case, an expression (`expr`) could either be a number or a binary expression (`binaryOpExpr`). A binary expression consists of two expressions separated by a binary operator (`binaryOp`). Binary operators are "+" and "-" (one at a time).

A *lexer*, also called *tokenizer*, is a program that splits input into a stream of *lexical units*, also called *tokens*. The lexer's input is usually a program source file and its output is fed into the parser.

The work of matching this stream of tokens to the grammar's rules is done by the *parser*. There exist a number of different parser approaches such as "LR" and "LL" parsers for context-free grammars that differ in the way input and grammar rule matching is handled. For context-sensitive grammars, "Packrat" parsers are used.

Algebraic Data Types (ADTs) are composite data types that are created by combining different existing data types to a new one. They are particularly common in functional programming languages.

III. Related Work and Contribution

In this section, a few works related to the comparison of different language technologies are presented. Creation of a DSL for web development using the metasyntax *Syntax Definition Formalism (SDF)* and *Stratego/XT* for code generation is described by Visser [33]. Cunningham [3] describes the implementation of a DSL for surveys in *Ruby*. Pelechano et al. [26] compare *Microsoft's DSL Tools* against

[1] http://www.hipacc-lang.org/

[2] http://www.ni.com/labview/

[3] http://www.mathworks.de/products/simulink/

Eclipse Modeling Framework (EMF) and *Graphical Modeling Framework (GMF)* based on the experience of implementing prototypes for various application domains. Using the example of describing a Finite State Machine, Vasudevan and Tratt [32] compare *ANTLR*, *Ruby*, *Stratego/XT* and *Converge*.

The implications of using DSL tools on the maintenance of languages is investigated by Klint et al. [16]. Mernik et al. [21] describe usage scenarios of DSLs, while van Deursen et al. [31] and Oliveira et al. [25] focus on the theoretical aspects of DSLs.

In [18] and [17], Membarth et al. performed a comprehensive evaluation of frameworks for multi-core architectures and GPU accelerators using 2D/3D image registration.

The main contribution of this paper is the presentation of different criteria for the evaluation of language development technologies. Furthermore presentation and comprehensive comparison of four different technologies for implementation of an external DSL based on criteria centered around the main requirements *programmer productivity* and *portability*. These may be adopted not only by other researchers in the field of computational sciences, but anyone planning to construct a DSL and its respective compilation framework, regardless of the domain.

The first of the evaluated technologies is Spoofax/IMP, which will be presented in the next section.

IV. Spoofax/IMP

Spoofax/IMP[4] (also called *Spoofax Language Workbench* or just *Spoofax*) [14] aims at providing a platform for developing textual DSLs based on the *Eclipse* platform. Using Spoofax, language designers are able to specify a grammar using SDF, a metasyntax for the definition of context-free grammars, which is then used as input for automatic parser generation. IDE features for the DSL like syntax highlighting, code folding, bracket matching and others are provided automatically and may be customized if necessary.

Another part of the Spoofax Language Workbench is the *Stratego Program Transformation Language* [1], which provides programmable term rewriting capabilities and enables code generation as well as implementation of advanced features like syntax completion.

Development started in 2007 with Stratego and SDF editors for Eclipse [12]. Today, the Spoofax Language Workbench is based on the Eclipse platform and the IMP framework, and includes libraries for parsing (JSGLR) and a Stratego/J runtime.

Data-Types and Types: Since the grammar of the DSL is specified in SDF all concomitant types are available. Using the automatically generated parser, an abstract representation of the given DSL syntax is produced. The internal representation is based on Java objects storing all necessary information [14]. Inside the Stratego part of the workbench, transformations work on Terms, which are functional and immutable data types. Originally, Stratego allowed only so-called Annotated Term data types [30] (a special kind of abstract data type designed for communication of tree-like data structures), but functionality has been extended to general Java objects implementing a special interface.

Parsing: A Scannerless Generalized LR (SGLR) parser based on a context-free grammar following SDF is generated. The parser works in real-time and is executed as a background process providing live parsing. Potential ambiguities within the grammar can be resolved by specifying precedence and associativity information. Furthermore, a permissive grammar, i. e., one that contains error recovery rules, can be automatically derived and ensures good parsing even when facing multiple syntactic errors [13].

Pattern-Matching and Transformations: In the Spoofax Language Workbench transformations can be specified using the Stratego language [1]. To do so, the programmer creates rules which match a given (first order) term and describe the replacement to be performed. Specification of more detailed matching is possible via conditions like parameter values. Using one or more of these rules allows the aggregation of a strategy, which then can be used to access visitor functions where multiple visiting strategies are available. Additionally, the implementation of context-sensitive transformations is possible by using strategies and dynamic rewrite rules [2]. Here, otherwise context-free rules are created at transformation time and thus provide means to be modified according to the current context. This, however, also means that any (context) information that will be needed in later transformations has to be gathered beforehand. Depending on the application, the amount and variability of this information can be quite high, possibly resulting in bloated code. Lastly, it is also noteworthy that various utility functions are built-in and ready to use.

Conclusion: Using Spoofax/IMP as the base for a DSL is possible and provides several benefits, especially towards the generation of an end-users' interface. Features like syntax highlighting and code completion, error checking, marking and recovery are generated automatically and provided as part of an Eclipse-based IDE. The underlying code, as well as the complete project, can be distributed as a stand-alone Eclipse plugin.

Despite its long history, the tool is not very stable. During development and testing, restarts of the IDE were often necessary due to errors.

V. Rascal

Rascal[5] is a Meta-Programming Language (MPL) in development since late 2008 [6] and licensed under the Eclipse Public License (EPL), allowing the use, modification, and redistribution of its source code freely in open source software as well as in proprietary software. Its goal is to be a language for the analysis, transformation, generation and visualization of source code, regardless of the kind or combination of programming language [10]. It is in active development by the SWAT group at CWI Amsterdam[6] and is the successor to the

[4] http://spoofax.org

[5] http://www.rascal-mpl.org/

[6] http://www.cwi.nl/research-groups/software-analysis-and-transformation

ASF and ASF+SDF Meta Environment[7]. Thus, its syntactic features are directly based on SDF [9] and its transformation and manipulation features are inspired by term rewriting and functional languages such as ASF+SDF [29] and Stratego [1]. Different parts of Rascal, however, are influenced by various other projects and theoretical works.

Rascal is available either as a plugin to Eclipse or as a standalone command-line tool, which requires the Java JDK. No further external dependencies are required to build a standalone DSL compiler. The Eclipse plugin supports syntax highlighting, outlining, and interactive visualization, but no code completion. It can be extended to support the DSL syntax [28] and thus to create an editor for the newly designed DSL.

Language Design: Rascal is an imperative language with a static type system that does not support run-time casts and is not object-oriented. It provides parametric polymorphism and sub-typing to allow the definition of generic functions. Functions are first-class values and can be passed to other functions as closures [15]. Rascal also supports control flow statements like `if`, `switch`, `while` and and exception handling using `try`, `catch` and `finally`. The standard `for` loop is replaced with a `for` comprehension, which is equivalent to a `foreach` statement in other languages. By using a list comprehension of numbers inside a `for` comprehension, the standard incrementing `for` loop can be expressed, similar to *Python*. In general, the syntax of Rascal's language constructs is similar to the ones found in established programming languages like C++ or Java. Code can be structured into functions and further aggregated in modules which are equivalent to source code files.

Data-Types and Types: The standard set of simple data types for expression of Boolean values, integers and floating point numbers as well as strings is provided. Furthermore, basic abstract data types like tuples, lists, sets and maps are available. There is a shorthand for a set of tuples of the same static type that is called *relation* and a special type for holding location information (like filename, current line and column) when parsing source code. Of high importance are ADTs, which can be user-defined or derived automatically from the grammar and are one way to process the parse tree. ADTs can be annotated with arbitrary values, i. e., with the previously mentioned location type or custom type information [15].

Parsing: Rascal generates a SGLR parser for context-free grammars. The grammar can be specified in an EBNF-like form and various disambiguation information like associativity and precedence can be specified [10]. The generated parse tree consists of ADTs, which need to be specified previously, and is annotated with location information.

Pattern Matching and Transformations: Every case distinction in Rascal is implemented via pattern matching. Different matching operators, namely string matching based on regular expressions, list and set matching, deep matching (up to an arbitrary depth) and negative matching as provided. Matching can be performed against basic and abstract data types (like ADTs). Rascal also provides support for matching of variables and binding variables to concrete syntax fragments like a `while` loop's condition and body [10].

All of these methods can be used at any location (for instance in `if` statements), in arbitrary combination and support wildcards like "don't care". Elements of a data structure (such as a parse tree) may be visited and matched with `visit` expressions. A visit expression corresponds to a recursive visitor known from general purpose programming languages but includes pattern matching expressions and can be used on objects of any type [15]. In case of a pattern having multiple matches, automatic backtracking happens, but a successful match may also be discarded manually. For each match, code to be executed or a transformation replacing the matched node may be specified.

Conclusion: Rascal is a well-thought tool for the creation of DSLs and the respective compilation frameworks behind. It provides many good and helpful features, like automatic annotation of locations done by the parser or the ability to generate a visual representation of an Abstract Syntax Tree (AST). It generates parsers only supports context-free grammars. Context-sensitive transformations may be specified in a short and expressive notation. Furthermore, end-user editor support with syntax highlighting and error tracking can be generated.

Stability is good, but Rascal still needs to mature in terms of usability. Error messages often are not helpful, e. g., when there is an error in the definition of an ADT, Rascal will only output it was unable to find the suitable constructor, thus hinting for the wrong place to fix it. There are hidden assumptions that have to be met (i. e., names of data types that have to match) and that make it difficult to understand the work flow, especially when debugging. In general, debugging capabilities lack some power, particularly in the field of pattern matching: It is unclear, for what reasons a pattern did or did not match a given data structure.

Documentation on Rascal is of high quality and features simple examples, but lacks deepness. Java libraries (and thus, natively compiled libraries) can be used in Rascal.

VI. Custom C++ Framework

For implementing a custom DSL framework, there are a variety of language possibilities and since usage of C++ is far-spread in HPC, it is also a language of choice for this investigation.

Language Design: C++ is a statically typed object-oriented general purpose programming language which is being developed since the late seventies and matured to one of the most widely used and accepted programming languages. Despite its long history, it is still actively developed, especially in the last few years, and a lot of new features were added with the new standards. With C++11, for instance, elements from functional programming languages like *Lambda Functions* and *Function Binding* became available. Employing these new additions allow a short, but powerful syntax for the specification of transformations and pattern matching.

[7] http://www.meta-environment.org/

When programming C++, a common addition is the usage of the *Boost* [8] source library, which adds new (data) types and features. For our project, and for better portability, we decided to use solely header-only functions from Boost, thus avoiding the need for compiled libraries.

Data-Types and Types: Simple and complex data types are provided, where complex data types may be represented through structs and classes. Additionally, (multiple) inheritance, and thus class hierarchies, as well as template classes, are available. Furthermore, datatypes from the Standard Template Library (STL) and Boost like lists, tuples, maps and smart pointers can be used.

Parsing: As writing a parser by hand is cumbersome and error-prone, the classic combination of *Flex* and *Bison* is used. Flex[9] is a generator for scanners (also called tokenizers) in C or C++ and is published under the BSD license. From a table of regular expressions and tokens, it can produce source code that reads input from a file or standard input and returns the tokens it matched for further processing, e. g., parsing. Bison[10] is a generator for deterministic or generic LR parsers for context-free grammars, which can output C, C++ or Java source code. Bison itself is licensed under the GNU Public License (GPL) and the generated result uses significant amounts of code of Bison, but an exclusion clause prevents the GPL applying to it [7], so the generated code may be used also in proprietary projects. If a specified grammar rule matches the token stream supplied by the scanner, associated AST construction code or other code will be executed.

While there are other tools available, Flex and Bison generate C++ classes and are thoroughly tested and proven to work. They may also be integrated in the compilation process, easing changes to the language syntax.

An alternative approach to parsing would be the use of Boost Spirit. While it is as powerful as the combination of Flex and Bison, it was not selected in the first place as its syntax is more complex and compile times are increased due to the heavy use of C++ templates.

Pattern-Matching and Transformations: The most straightforward approach is probably implementing matching functions by hand. This means all comparisons have to be specified, although similar functions can be grouped. Nevertheless, it is obvious that this technique is infeasible for a project of our complexity. Instead, it is advisable to look for different approaches to reduce or at least hide the complexity. One solution is taking advantage of the polymorphism inherent to C++. In detail, the different node types are modeled as class definitions in a large class hierarchy where virtual transformation functions are specified as required. A recursive visitor then can be added with a single line of code per class using variadic macros and specialized template functions, even without introspection. The remaining matching functionality may be further facilitated using additional utility functions and macros.

Within transformations, all information that has been gathered by previous transformations up to the current program state can be accessed freely. Two basic rewriting strategies, exchanging nodes from within the tree and setting up a new modified tree at each transformation step, are easy to implement and to use.

Conclusion: Using C++ in the way described above is a viable approach to the creation of a custom DSL construction framework. Just after we finished our evaluation on this approach, a similar method for the realization of pattern matching in C++ has been published by Solodkyy et al. [27]. Compile times and runtime performance are excellent, as is developer support by various IDEs and their debugging capabilities. Since C++ is an accepted standard and used by millions of developers, extensive documentation is available and many third-party components can be employed.

A basic problem, however, is that required but missing functional aspects have to be emulated with great effort and partially very unorthodox code which is cumbersome to implement and quite error-prone. Additionally, problems arise when using too many features from the new C++ standards which chronically reveal incompatibilities with different compilers. For some functions, this can, at least to some extent, be mitigated or resolved by using Boost instead.

The missing support for the generation of a DSL IDE is another draw-back, as this results in a lot of work that has to be done manually. Conceivable approaches include basic Eclipse or *vim* plug-ins, possibly reusing parts of the parser and error-checking, as well as advanced Eclipse plug-ins using the previously presented Spoofax language workbench.

VII. Scala

Scala[11], developed at EPF Lausanne, Switzerland, since 2001 and released to the public in 2004, is a general-purpose object-functional language, meaning it is both an object-oriented and a functional programming language at the same time. While Scala itself is licensed under a BSD style license, the code written in Scala is subject to the developer's personal choice of license. It runs inside the Java VM and can interact with Java libraries, but also take advantage of the Java Native Interface (JNI) [23]. A port to Microsoft .NET CLR exists, but development has stopped [11]. Scala itself is actively maintained and developed with the most recent version, 2.10.3, released in October 2013. It is very stable and mature and forms the base of many academical and industrial software projects, e. g., it is the base of Liszt, an embedded DSL for solving mesh-based PDEs [4], and powers the back-end of the popular social network Twitter [8].

An Eclipse plugin providing (amongst other functionality) code completion and syntax highlighting is available, but there exist many other comparable IDEs and editors for various operating systems. Apart from the Scala compiler or interpreter requiring the JVM, no external dependencies are needed for the creation of an external DSL.

[8] http://www.boost.org/

[9] http://flex.sourceforge.net/

[10] http://www.gnu.org/software/bison/

[11] http://www.scala-lang.org/

Scala does not come with a way to automatically generate an IDE matching a DSL specification. However, bindings to the Swing GUI elements library are provided, so building a custom editor and re-using parts of the actual DSL compiler implementation is possible.

Language Design: Scala's design concentrates on mechanisms for abstraction, composition and decomposition. It features a conventional syntax with an innovative type system, of which we will highlight the relevant parts for our target in a later paragraph. Scala is statically typed and supports run-time casts [23]. Since version 2.10, Scala comes with its own Reflection API allowing for more advanced runtime information and more powerful operations than the previously available Java Reflection API. This allows for dynamic (at runtime) manipulation of objects in much more flexible ways.

Data-Types and Types: While Java's data types may be used in Scala code, equivalents to the simple types and many abstract data types are provided natively. Scala features an unified type system, meaning all types derive from the base type `Any`. This also implies that all values are objects, unlike Java's special simple data types. Scala works around Java's type erasure, so the complete type information is available at runtime, easing transformations by providing more runtime information. Case-classes are the equivalent to ADTs and are automatically transformed to standard classes by the compiler by generating getter and setter functions and a set of other default methods. Classes (as well as functions) may be nested. Scala also supports *traits*, which correspond to Java interfaces, but may be partially implemented. Classes and traits support *mixin composition*, which is a special form of inheritance that cannot be expressed with multiple inheritance: The delta in functionality between a superclass (or supertrait) and subclass (or subtrait) can be added to another class (or trait) without overwriting any of the functionality implemented there [23].

Parsing: Scala provides support for *Parser Combinators*, which is a functional approach to the implementation of recursive descent parsers. A *parser combinator* is a combination (i. e., a higher-order function) consisting of different, specialized parsers modeled as functions. That way, a context-sensitive grammar can be implemented.

The grammar can specified in a form similar to the EBNF, which is basically a DSL embedded into Scala. Using translations, input that has just been parsed may be preprocessed before it is returned by the parser and added into the AST. However, data structures may be chosen freely [24], so construction of an AST may be reduced to those areas where it makes sense (e. g., in calculation terms or imperative programming statements). In other settings, more appropriate data types may be employed, e. g., a list when saving key-value-assignments.

Pattern-Matching and Transformations: The aforementioned case-classes are well suited for pattern-matching. While a whole case-class type can be matched, it is also possible to extract and match its data members (which are equivalent to its constructor arguments).

Visiting is implemented via the `match` keyword and traversing a tree of case-classes is possible in a recursive manner. Transformations may be specified using the => operator specifying the new element to put in place, however, the underlying structure is immutable, which means it cannot be changed. To work around this issue, a (sub)tree can be duplicated [23], enabling support for backtracking in case a transformation was applied that later proved to be suboptimal.

Conclusion: While Scala is very popular for the creation of *embedded* DSLs, it is as capable for the creation of *external* ones. It provides powerful parser combinators that allow the use of context-sensitive grammars, easing specification of a more expressive DSL and reducing the amount of complex (context-sensitive) transformations needed. Transformations can be described in a short and expressive way and easily access all the information that has been gathered previously. Furthermore, generation and traversal of complex data structures is easy.

However, since there is no support for DSL IDE generation, this work has to be done manually. Building on the parser and the compilation framework's error-checking, an editor could be built using Scala's binding for the GUI elements framework *Swing*. Since Java code may be used anywhere in Scala code, any other GUI toolkit that provides Java bindings will also work.

A high-quality API Reference[12] as well as the complete language specifications [22] for Scala are provided. Due to its huge user base and widespread use, a lot of third-party documentation, tutorials and examples do exist. Consequently, many ready to use components and example implementations to common problems exist which might be incorporated.

VIII. Comparison

In this section, a rating based on various criteria will be proposed, accordingly to which the four technologies will be evaluated thereafter. We will present a weighted rating with values based on our experiences and considerations. However, the reader may feel free to adapt the weights depending to their points of view. In order to provide a fair comparison, we exclude the influence of previous knowledge about a technology on its rating by not comparing the progress made in the given timeframe.

A. Comparison criteria

In this subsection, we define a number of criteria according to which the investigated technologies will be compared. These criteria can be grouped into two categories. Criteria of the first category focus on the technology itself and therefore are called *core criteria*. The second category comprises *environment criteria* that attribute to the surroundings of a technology, e. g., its documentation and development activity. Furthermore, certain criteria are labeled *hard*, meaning a technology has to fit. On the other side, it would be nice if a technology meets the *soft* criteria.

[12] http://www.scala-lang.org/api/current/

TABLE I
TABULAR OVERVIEW OF CRITERIA, WEIGHTS AND ACHIEVED POINTS PER TECHNOLOGY. POINTS PER CRITERION ARE IN THE RANGE $[-2; 2]$, RESULTING IN A POSSIBLE TOTAL WEIGHTED SUM OF $[-84; 84]$.

Criterion		ω	**Spoofax**	**Rascal**	**C++**	**Scala**
Core criteria:						
License	*hard*	5	2	2	2	2
Development phase	*hard*	5	-1	0	2	2
Expressiveness	*hard*	5	2	2	1	2
Matching of concepts	*soft*	3	2	2	0	2
External dependencies†	*soft*	3	-2	0	1	1
Context-sensitive grammar	*soft*	2	-2	-2	-1	2
DSL IDE generation	*soft*	1	2	1	0	0
Debugging	*soft*	3	0	0	2	1
Component availability	*soft*	1	0	-1	0	1
Code structuring	*soft*	2	2	2	2	2
Weighted sum		–	17	26	36	51
Environment criteria:						
Documentation quality	*soft*	3	0	1	2	2
Documentation quantity	*soft*	3	0	-1	2	2
User base size	*soft*	1	0	-1	2	2
Development activity	*soft*	2	0	2	2	2
Development team size	*soft*	1	1	1	2	2
Developer IDE	*soft*	2	1	1	2	2
Weighted sum		–	3	6	24	24
Total weighted sum		–	**20**	**32**	**60**	**75**

† Includes *technical base*, e. g., Java RE

Core Criteria / Hard Criteria: Very important is the technology's **license**. Naturally, we want to use the technology in our development process as well as bundle the generated code with our project's source code files. Therefore, this has to be permitted by the technology's license. As we consider releasing our project under an open-source license such as the (L)GPL or the BSD license, the require the technology to be available under a compatible license. To be usable for day-to-day work, the technology should be mature and in a stable **development phase**. We expect not to be surprised by changes in the API or other key elements of the technology that otherwise will result in porting the code of our project to the changes that were , so it is usable for day-to-day work.

Another *hard* criterion is the ability to describe transformations and find information in the program's intermediate representations for specifying optimizations in a short and elegant way that can be easily understood later and by other programmers. We call this **expressiveness**.

Core Criteria / Soft Criteria: **Matching of concepts** is a *soft* criterion and denotes how much a technology's design and concepts match and support our vision. Concepts not matching usually cause workarounds and misuse of technology features in the code to be written, which we want to avoid. The technological base should be portable, so our project's compiler not only runs on widespread desktop operating systems, but possibly also on cluster login nodes. Most critical to this are **external dependencies** of which a technology should have as less as possible. Examples for external dependencies are required runtimes or external programs that have to be called during compilation of the DSL compiler. Both belong, like the following criteria, to the *soft* criteria category. Support for **context-sensitive grammars** will allow the construction of a more expressive DSL. To help the end-user creating DSL programs, we wish for the facility to **generate a DSL IDE** or an editor with syntax highlighting. For providing a comfortable developer experience, it is desirable to have extensive **debugging** mechanisms and ready to use **available components**. To ease collaboration between different developers and groups, clean ways for **structuring of code** are desirable.

Environment Criteria: All criteria in this category are *soft*. While a technology should be intuitive to use, providing a **high-quality and extensive documentation** is very important. Also, in case of questions, a large and active **user base** is a great resource of helpful comments. Furthermore, **development activity** and **development team size** are important factors concerning bugfixing, addition of new features and clarification of questions. A **developer IDE** with supporting

features like syntax highlighting and code completion is another great help for developers.

B. Evaluation

Table I merges the technology evaluation in a short way. A technology receives a number of points in the integer range $[-2; 2]$ for the fulfillment of each of the presented criteria. Each criterion has been weighted according to its relevance for our overall goal. Possible values are 5 for *hard* and $[1; 3]$ for *soft* criteria, depending on the importance. The total weighted sum is calculated by the summation of the multiplication of a technology's points for a criterion with its weight ω. Points are given according to our estimation after a usage time of about three weeks per technology to develop a prototype involving parsing and some of the transformations elementary for our envisioned project goal.

The licenses of all evaluated technologies fit our requirements and also render a source code release under an open-source license possible. In terms of stability or assumptions that have to be met, both Spoofax and Rascal are not as mature as C++ and Scala, resulting in a lower score in the criterion *development phase*. Expressiveness of the custom C++ solution is not as powerful as in the other solution, as also the concepts do not match perfectly. This results in a more difficult notation of transformations and data handling, which in turn lowers programmer productivity. Spoofax gets the most negative score on *External Dependencies* for being available only as an Eclipse plugin, making it hard to run on a cluster node. In contrast, Rascal code can be ran using Java's JVM and the Rascal command-line tool. As the Scala compiler generates Java bytecode, a JVM is sufficient to run a Scala program. C++ programs are usually compiled on the target platform, so a C++ compiler, Boost, Flex and Bison are needed for this approach. However, all of these are installed on an average Linux system by default, so only one point has been subtracted here.

Furthermore, Spoofax and Rascal only support context-free grammars, yielding another negative score. In C++, a different parser generator or a handwritten parser could be used, resulting in a slightly higher score. Debugging capabilities of all investigated approaches are usable, but especially extensive in C++ thanks to its large tool collection and widespread use. Code can be structured very good in every one of the four solutions.

Documentation for the specialized technologies Spoofax and Rascal is not as plentiful and good as for C++ and Scala. As general purpose programming languages are much more popular, obviously their user base is much larger, also resulting in more ready-to-use components and libraries available.

IX. Conclusions

Based on a number of core and environment criteria, an evaluation of four technologies for the implementation of a DSL, namely Spoofax/IMP and Rascal as well as custom approaches using the general purpose programming languages C++ and Scala, was made.

Especially in the core criteria, Scala scores well and in the environment criteria, it is on par with C++. Both approaches specialized towards language creation score lower, a fact that might be surprising at first, but probably can be attributed to their focus on a smaller target community. A general-purpose language, especially such popular languages such as C++ and Scala, obviously has (in terms of developers) a much larger target audience, driving the technology's development towards various requirements. This results in a development pace, intensity, and diversity that is hard to match for smaller teams of specialized approaches.

Rascal, however, has a high development activity that, if this rate can be held, will help it to become a very viable and usable solution in its field in the medium term. Scala already is a very reliable technology and not only provides many of the features Rascal offers, but also has a few advantages of its own, as described previously.

Thus, Scala will lay the ground for the future work described in the next section.

X. Future work

From the above analysis, we derive to develop a DSL for the solution of PDEs using automatically generated and optimized stencil code for heterogeneous HPC systems using Scala. Thus, future research will incorporate the necessary semantics and syntax incorporating concepts and terms of this particular domain. The investigation of concepts on how to represent the domain expert's knowledge in the compiler, i. e., the selection of suitable parameters for automatic discretization of the PDE as well as the selection of matching algorithms for solution of the resulting system of equations, is another challenge. Furthermore, future research work will also include the transformations needed for parallelization and performance optimization for different target hardware architectures, employing technologies like offloading to GPU and FPGA accelerators as well as matching communication strategies using MPI.

XI. Acknowledgments

This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 "Software for Exascale Computing" in project under contracts TE 163/17-1 and RU 422/15-1.

References

[1] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-–2):52–70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[2] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, 2006.

[3] H. C. Cunningham. A little language for surveys: constructing an internal DSL in Ruby. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 282–287, New York, NY, USA, 2008. ACM.

[4] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[5] M. Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

[6] Github. Rascal repository. https://github.com/cwi-swat/rascal/, 2013. [Online; accessed: 2013-10-07].

[7] GNU Software Foundation. *Conditions for Using Bison*, 2012. [Online; accessed: 2013-10-08].

[8] K. Greene. The secret behind Twitter's growth. 2009. [Online; accessed: 2013-10-23].

[9] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual–. *SIGPLAN Notes*, 24(11):43–75, Nov. 1989.

[10] M. Hills, P. Klint, and J. J. Vinju. Program analysis scenarios in Rascal. In *Proceedings of the 9th international conference on Rewriting Logic and Its Applications (WRLA)*, pages 10–30, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] J. Iry. MSIL backend is unworking, unused, and unmaintained. https://issues.scala-lang.org/browse/SI-6772, 2012. [Online; accessed: 2013-10-07].

[12] K. T. Kalleberg and E. Visser. Spoofax: An interactive development environment for program transformation with Stratego/XT. In T. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA)*, ENTCS, pages 47–50, Braga, Portugal, March 2007. Elsevier.

[13] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In G. T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 44 of *ACM SIGPLAN Notices*, pages 445–464, New York, NY, USA, October 2009. ACM Press.

[14] L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In M. Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 17-21, 2010, Reno, NV, USA*, pages 44–463, 2010.

[15] P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.

[16] P. Klint, T. van der Storm, and J. Vinju. On the impact of DSL tools on the maintainability of language implementations. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA)*, pages 10:1–10:9, New York, NY, USA, 2010. ACM.

[17] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for GPU accelerators: A comprehensive evaluation using 2d/3d image registration. In *Proceedings of the 9th IEEE Symposium on Application Specific Processors (SASP)*, pages 78–81.

[18] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for multi-core architectures: A comprehensive evaluation using 2d/3d image registration. In *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS)*, volume 6566 of *Lecture Notes in Computer Science (LNCS)*, pages 62–73. Springer.

[19] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Generating device-specific GPU code for local operators in medical imaging. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 569–581, Shanghai, China. IEEE.

[20] R. Membarth, F. Hannig, J. Teich, and H. Köstler. Towards domain-specific computing for stencil codes in HPC. In *Proceedings of the 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 1133–1138, Salt Lake City, UT, USA. IEEE.

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

[22] M. Odersky. The Scala Language Specification, 2013.

[23] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[24] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Series. Artima, Incorporated, 2010.

[25] N. Oliveira, M. J. ao Varanda Pereira, P. R. Henriques, and D. da Cruz. Domain-Specific Languages – A Theoretical Survey. In *Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA)*, pages 35–46, 2009.

[26] V. Pelechano, M. Albert, J. Muñoz, and C. Cetina. Building Tools for Model Driven Development. Comparing Microsoft DSL Tools and Eclipse Modeling Plug-ins. In *Desarrollo de Software Dirigido por Modelos*, pages 429–429, 2006.

[27] Y. Solodkyy, G. Dos Reis, and B. Stroustrup. Open Pattern Matching for C++. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 33–42, New York, NY, USA, 2013. ACM.

[28] J. Van Den Bos, P. R. Griffioen, T. Van Der Storm, et al. Rascal: Language technology for model-driven engineering. In *ICT. Open*, Nov. 2011.

[29] M. G. J. van den Brand, A. v. Deursen, J. Heering, H. A. d. Jong, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction (CC)*, pages 365–370, London, UK, 2001. Springer-Verlag.

[30] M. G. J. van den Brand, H. D. Jong, P. Klint, and P. Olivier. Efficient annotated terms. *SOFTWARE, PRACTICE & EXPERIENCE*, 30:259–291, 2000.

[31] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[32] N. Vasudevan and L. Tratt. Comparative study of DSL Tools. *Electron. Notes Theor. Comput. Sci.*, 264(5):103–121, July 2011.

[33] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School (GTTSE)*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Braga, Portugal, 2007. Springer.