Comparative Study of DSL Tools

Naveneetha Vasudevan[1]

Bournemouth University
Poole, Dorset, BH12 5BB, United Kingdom

Laurence Tratt[2]

Bournemouth University
Poole, Dorset, BH12 5BB, United Kingdom

Abstract

An increasingly wide range of tools based on different approaches are being used to implement Domain Specific Languages (DSLs), yet there is little agreement as to which approach is, or approaches are, the most appropriate for any given problem. We believe this can in large part be explained by the lack of understanding within the DSL community. In this paper we aim to increase the understanding of the relative strengths and weaknesses of three approaches by implementing a common DSL case study. In addition, we present a comparative study of the three approaches.

Keywords: DSL, Converge, Ruby, Stratego.

1 Introduction

Domain Specific Languages (DSLs) are mini-languages tailored for a specific domain, offering significant advantages over General Purpose Languages (GPLs) [3]. DSLs allow programs to be implemented at the level of abstraction of the application domain which enables programmers to develop programs quickly and effectively. Given a domain and the need for a DSL, there exist a number of tools and approaches for implementing DSLs. The classical approach to DSL implementation uses compiler tools such as LEX and YACC, where DSLs are implemented as 'stand-alone' for a particular application domain [1]. However, their application in contemporary software systems has been restricted for two reasons: the high start-up costs involved in implementing DSLs from scratch; and the lack of re-use of software artifacts from other DSL implementations [3]. Conversely, embedded approaches have been used to implement DSLs. Lisp and Nemerle [8] support construction of arbitrary program fragments at compile-time through the use of macros.

[1] e-mail: naveneetha@yahoo.com
[2] e-mail: laurie@tratt.net

In a pure embedding approach, where no macro-expanders or generators are used, DSLs are implemented as Domain Specific Embedded Languages (DSELs) using host language features such as higher-order functions and polymorphism [6].

Embedding approaches can be either homogeneous or heterogeneous [10]; in heterogeneous embedding, the system used to compile the host language, and the system used to implement the embedding are different; whereas in a homogeneous system, the systems are the same, and all its components are specifically designed to work with each other. This distinction is important as it allows one to understand the limitations of a given approach. Among homogeneous embedding approaches, compile-time meta-programming has been used extensively to implement DSLs [2,5], by allowing the user of a programming language to interact with the compiler to construct arbitrary program fragments at compile-time. Among heterogeneous embedding approaches: Stratego/XT [9] supports implementation of DSLs by program transformation through term rewriting; and Silver [11] supports implementation of DSLs through the use of language extensions, where new language constructs (for domain specific features) are translated to semantically equivalent constructs in the host language through transformation.

In similar style to Czarnecki *et al.* [2], which evaluates the compile-time meta-programming abilities of three languages, we use the case study of a generic state machine to study three different approaches that represent important, differing, points on the DSL implementation spectrum: Ruby typifies a weakened form of Hudak's vision of DSELs; Stratego/XT can embed any language inside any other; and Converge uses compile-time meta-programming to implement customisable syntax. The code for each of our examples can be downloaded from `http://navkrish.net/downloads/dsl_tools_src.tar.gz`. To the best of our knowledge, this is the first time that three 'modern' approaches to DSL implementation have been evaluated together and we hope this comparative study will benefit future implementation of DSLs.

2 Case Study: Finite State Machine

The example used in this paper is a generic state machine for a Turnstile (Figure 1) with states and transitions. The syntax for a 'transition' is represented using the UML notation `event[guard]/action`, where `event` represents an event that triggers the transition, `guard` represents the condition that must evaluate to *true* for the transition to occur and `action` represents the subsequent action. For our case study, we represent the state machine as a DSL, so that we can have a running state machine we can fire events at and examine its behaviour.

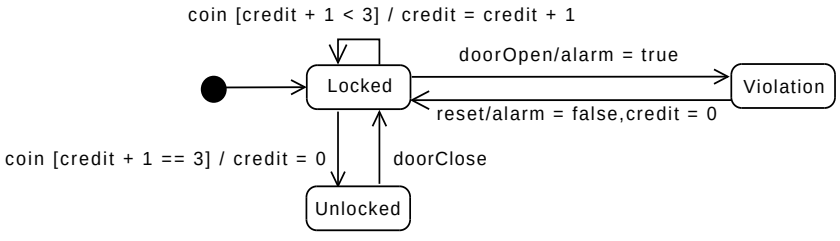

Fig. 1. State machine for a Turnstile

3 Implementation of a DSL in Ruby

Ruby is a dynamically-typed, general purpose object-oriented language [4]. In Ruby, DSLs are implemented using a combination of features such as lambda abstractions (code blocks), evaluations, dynamic typing, reflection and flexible syntax. In Ruby, a *code block* is a closure that can be used to encode domain specific information. A code block is expressed either on a single line using delimiting curly braces ({`|x| print x` }) or over multiple lines using `do` and `end` keywords. A code block encoding the domain specific information for a transition (from our case study) is as follows:

```
transition "charging" do |t|
  t.from_state 'locked'
  t.to_state = 'locked'
  t.guard do |credit|
    if (credit + 1) < 3
      true
    end
  end
  ...
end
```

In the above code, the `transition` construct that initially looks like a DSL keyword describing a transition, represents an invocation of the method – `transition` – followed by two arguments: a string, and a code block that accepts a block parameter (`|t|`). The constructs – `t.from_state 'locked'` and `t.to_state = 'locked'` – that look like DSL keywords describing the attributes of a transition, represent method invocations – `from_state` and `to_state=` – on object `t`, followed by an argument. The two variant method name styles highlight the syntactic flexibility that DSL authors in Ruby can use to tweak the language to their needs. Furthermore, the above code shows how a code block is passed as an argument to a method invocation (e.g. `transition "charging" <code block>`). In Ruby, methods accept a code block as a final argument, however, to pass the code block around, the method definition needs to include a *block argument* (a final argument of the form `&aBlock`) that would allow the code block to be implicitly converted to a `Proc` object. The following code fragment shows how adding a block argument to the definition of the `transition` method enables the code block to be passed around as a `Proc` object (`&aBlock`):

```
class Fsm
  def transition(name, &aBlock)
    transition = Transition_class.new(name)
    transition.load_block(&aBlock)
  ...
```

A `Proc` object can be executed either by using `yield` or by invoking its method `call` (e.g. `aBlock.call`), with any arguments passed to them assigned to the block parameters. The following code fragment shows how the `&aBlock` object is eventually executed by calling `yield self` (`self` refers to `transition` object from the above code fragment) and the corresponding method definitions for the `from_state` and `to_state=` constructs from the above code block:

```
class Transition_class
  def from_state(from_state)
    @from_state = from_state
  end

  def to_state=(to_state)
    @to_state = to_state
  end
```

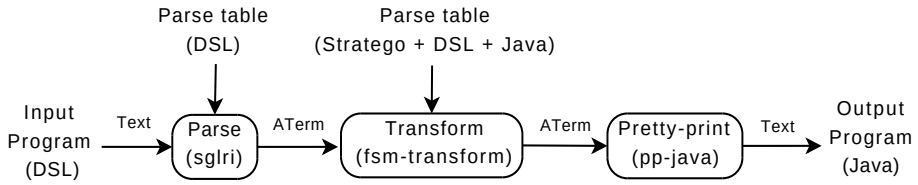

Fig. 2. The transformation pipeline in Stratego showing the various stages to implement DSLs

```
def load_block
  yield self
end
...
```

In addition to code blocks, Ruby supports dynamic typing, which allows the runtime system to implement features such as dynamic dispatch and duck typing. For instance, the `Object` class enables dynamic dispatch in every object by defining two methods: `responds_to?` checks if an object will respond to a message; and `method_missing` catches messages an object has no explicit handler for. In a similar vein to Smalltalk, Ruby supports the creation (or replacement) of methods at runtime that can then be used to dynamically manipulate the behaviour of an object.

4 Implementation of a DSL in Stratego/XT

Stratego/XT [9] is a software transformation framework that consists of the Stratego language (for implementing program transformations through term rewriting) and the XT toolset (for providing the infrastructure to implement these transformations). In Stratego/XT, programs are transformed by representing them in the form of abstract syntax trees called Annotated Terms (ATerms), and then exhaustively applying a set of term rewrite rules and strategies to them.

In Stratego/XT, DSLs are implemented using a transformation pipeline (Figure 2) consisting of three stages: a parsing stage that implements the parser for the DSL; a transformation stage that implements the transformation program using the Stratego language; and a pretty printing stage that unparses the final ATerm to the target program. The XT toolset [9] provides the necessary tools for the parsing and the pretty-printing stages. We focus our attention on the crucial stage of the transformation pipeline—the transformation program. A transformation program is implemented using a set of term-rewrite rules and strategies. A term-rewrite rule defines a transformation on a ATerm and is of the form `L : p1 -> p2`, where `L` is the rule name, and `p1` and `p2` are term patterns. A strategy is a program that supports the exhaustive application of rules to an ATerm by defining the order in which the terms are re-written. A term-rewrite rule can be written using the concrete syntax of the object languages rather than using nested ATerm patterns [12]. For instance, the assignment of an expression to a variable is expressed in concrete syntax as `|[ x := e ]|` rather than using nested ATerms—`Assign(Var(x),Expr(e))`. A condensed version of the DSL program and its corresponding transformation rules for our case study are as follows:

```
state locked
transition unlocking from locked to unlocked : coin [ credit + 1 == 3 ] / credit := 0

var-init : |[ state x_s ]| -> |[ this.states.add("~x_s"); ]|
guard-init :|[ transition x_t from x_a to x_b : x_e ttail1 ]| ->
```

```
            |[ if (...) { bstm_1 } ]| where <trans-tail> ttail1 => bstm_1
trans-tail : trans-tail |[ guard1 ]| -> |[ ... ]| where <guard> guard1 => e_1
trans-tail : trans-tail |[ action1 ]| -> |[ ... ]| where <action> action1 => bstm_1*
...
```

Further, the above code highlights two aspects. First, the embedding of meta-variables (such as `x_s` and `ttail1`) within transformation rules leads to conciseness and better readability of the transformation program as compared to the use of nested terms. Meta-variables are patterns for the syntactic elements (such as identifiers, expressions and lists) of the object language. Second, the use of a *where* clause for programmable application of rules. For instance, the `<trans-tail> ttail1` construct within the where clause of the `guard-init` rule, can invoke either of the `trans-tail` rule, depending upon the value of `ttail1` at run-time.

5 Implementation of a DSL in Converge

Converge [10] is a dynamically typed imperative programming language, with compile-time meta-programming (CTMP) and syntax extension facilities. Converge, a syntax-rich modern language, unifies concepts from languages such as Python (indentation and datatypes) and Template Haskell (CTMP).

DSLs are implemented in Converge using its CTMP facility. CTMP can be thought of as being equivalent to macros, as it provides the user with a mechanism to interact with the compiler, allowing the construction of arbitrary program fragments by user code. Converge achieves this construction of arbitrary program fragments using its compile-time meta-programming features—splicing, quasi-quotation, and insertion [10]. Splice annotations `$<...>` evaluate the expression between the angled brackets, and replace the splice annotation itself with the result (AST) of its evaluation. Quasi-quotes `[|...|]` allows the user to build ASTs that represent the program contained in them using Converge's concrete syntax. Insertions `${...}` are placed within quasi-quotes to evaluate the expression, and copy the resulting AST as is into the AST being generated by the quasi-quote.

Converge allows any arbitrary DSL to be embedded within normal source files via a *DSL block*. A DSL block is introduced within a converge source file using a variant of the splice syntax `$<<`*`expr`*`>>` where *`expr`* must evaluate to a *DSL implementation function*. This function is then called at compile-time to translate the DSL block into a Converge AST, using the same mechanism as a normal splice. DSL blocks make use of Converge's indentation based syntax; when the level of indentation falls, the DSL block is finished. A DSL block and its corresponding DSL implementation function for our case study are as follows:

```
TurnstileFSM := $<<FSM_Translator::mk_itree>>:
   ...
   state locked
   transition unlocking from locked to unlocked : coin [ credit + 1 == 3 ] / credit := 0

func mk_itree(dsl_block, src_infos):
   parse_tree := parse(dsl_block, src_infos)
   return _Translator.new().generate(parse_tree)
```

The DSL implementation function `FSM_Translator::mk_itree` is called at compile-time with a string representing the DSL block along with the *src infos* obtained from the Converge tokenizer. The Converge Parser Kit can then be used to parse this string against the user-specified grammar to produce a parse tree. This parse

tree, which contains tokens and their associated src infos, can then be traversed and translated to an AST using quasi-quotes and insertion. Converge provides a simple framework for this translation; where a translation class (`_Translator` in the above DSL implementation function) contains a function `_t_`*production name* for each production in the grammar. The `self._preorder` method can then be used to call the appropriate `_t_` function, given a node in a parse tree. The following code fragment shows how `_t_event` function gets invoked from `_t_transition` function:

```
func _t_transition(self, node):
    // transition ::= "TRANSITION" "ID" "FROM" "ID" "TO" "ID" transition_tail
    tail_node := node[6]
    if tail_node.len() != 0:
        // transition_tail ::= ":" event guard action
        event := self._preorder(tail_node[1])
        ...

func _t_event(self, node):
    // event ::= "ID"
    if node.len() != 0:
        return CEI::istring(node[0].value)
        ...
```

For our case study, we wish to translate the DSL program into an anonymous class. This class can then be instantiated to produce a running state machine `turnstile := TurnstileFSM.new()`, which can receive and act upon events (`turnstile.event("coin")`). The second argument to the DSL implementation function is a list of src infos. Src infos are covered later in Section 6.

6 Analysis and Comparison

In this section, we use and extend the dimensions identified by Czarnecki *et al.* [2] to present a comparative analysis of the three DSL tools.

Dimension	Ruby	Stratego/XT	Converge
Approach	Lambda abstractions	Term rewriting	Compile-time meta-programming
Guarantee	Syntax valid (runtime)	No	Well-typed (compile-time)
Reuse	Limited	SDF grammar	Limited
Lines of code (Grammar, transformation, and DSL program)	n/a, 89, 55	79, 95, 12	36, 173, 11
Type checking	No	Yes	No
Error reporting	Yes (runtime)	Limited (end language)	Compile-time

Table 1
A comparative analysis of Ruby, Stratego and Converge

Approach What is the primary approach supported by the DSL tool? In Ruby, DSLs are implemented using a combination of its host language features such as lambda abstractions, dynamic typing and reflection. In Converge, DSLs are implemented using its compile-time meta-programming facility, where a DSL is translated to host language constructs at compile-time. In Stratego/XT, DSLs are implemented through term-rewriting, where a source program (DSL) is transformed to a target program (e.g. Java) using a set of transformation rules and strategies. The term-rewriting is performed by the transformation program (`fsm-transform` in Figure 2) at the preprocessor stage—a stage prior to the compilation of the target language program.

Guarantees What guarantees are provided by the DSL tool in terms of syntactic and semantic well-formedness of the transformed-to constructs? In the context of this paper, syntactic well-formedness guarantees that there are no syntax related errors when the transformed-to constructs are run through the host language compiler. Although there are potentially many different semantic guarantees that could be offered, we consider only the following: that the transformed-to program does not have references to any undefined variables; and that the transformed-to program does not have any type errors. In Ruby, DSLs are essentially host language constructs, and therefore, any guarantees with regards to both syntactic and semantic well-formedness are provided by the Ruby interpreter. In Stratego, few guarantees are given with respect to producing a syntactically and semantically well-formed target AST. For instance, a meta-variable within a transformation rule can be associated with an incorrect type that can lead to the generation of an invalid AST. Similarly, the target AST can contain semantically ill-formed constructs, which are only reported at the time of compilation of the end language. In contrast, the Converge compiler guarantees the well-formedness of the translated-to host language constructs at the time of translation.

Reuse What aspects of the DSL implementation that are user-defined can be reused? We identify two aspects that are potentially re-usable: the grammar of the DSL; and the transformation module. In Ruby, since the DSLs are essentially host language constructs, the above aspects become irrelevant. Even so, the interleaving of the DSL program and the host language constructs that evaluate the DSL program limit the re-usability of the DSL implementation. In Converge, since the grammar of the DSL and the DSL constructs are closely integrated with the host language constructs that perform the translation, large sections of the DSL implementation have limited re-use. In Stratego/XT, the modular SDF definition of the object language, and sections of the transformation program that implement the expression and the type transformations can potentially be re-used for other DSL implementations.

Lines of code For a given problem, how many lines of code are required to represent the domain-specific information? When evaluating implementation of DSLs based on lines of code, there are three aspects to be noted: the grammar for the DSL; the transformation or evaluation (in Ruby) module; and the DSL program. For our case study, the number of lines of code required to implement the grammar was almost twice in Stratego as compared to Converge. In general, the size of the grammar in Stratego is likely to be higher than in Converge, due to the inclusion of the SDF definitions of meta-variables. In Ruby, since the DSLs are essentially host language constructs, the aspect that deals with the grammar implementation is irrelevant. Further the DSLs are evaluated as is, resulting in the size of the evaluation program to be generally smaller as compared to Stratego or Converge. In Converge, the nodes in the AST are traversed (and translated) systematically, whereas in Stratego, multiple nodes in the AST are transformed by using a strategy. For our case study, where 'states' and 'transitions' are essentially a list of nodes in the AST, the use of strategies in Stratego results in a smaller transformation program as compared to Converge. However, in Stratego, the size of the transformation program will also be determined by the verbosity of

the target language. In contrast to the first two aspects, the third one is relatively important as it has the potential to be implemented many times over during the lifetime of a DSL. For our case study, the size of the DSL input program in Ruby was well over four times the size in Converge or Stratego. This is primarily because the syntax of the DSLs in Converge (or Stratego) are specifically designed for the problem in hand whereas in Ruby, the syntax of the DSLs is limited to that which can be naturally expressed by the host language.

Type checking Can the DSL tool type check disjointed fragments of the DSL program at the time of transformation? We explain type checking on disjointed fragments using SQL statements as an example. If there exists two DSL fragments, where the first fragment contains the definition of a table – `CREATE TABLE emp {id int(10)}` – and the second fragment contains the 'select' statement – `SELECT * FROM emp WHERE id=x` – can the DSL tool type check the `SELECT` statement using the definition of the `CREATE` statement? In Ruby, type checking is only possible by layering an external type checker that can then be invoked prior to the invocation of the host language interpreter. Converge does not support context-sensitive translation and therefore an external program will have to be implemented to perform type checking that can be invoked at the time of translation. In Stratego, however, term rewriting can be extended with dynamic rules to perform type checking. For our SQL scenario, a dynamic rule for each of the columns of a table can be defined within the context of the 'create' statement. The transformation rule for the `SELECT` statement can then invoke the dynamic rule to perform type checking on the `WHERE` clause.

Error reporting Can the DSL tool report errors in terms of the DSL source (line number and column offset)? We identify and present a broad classification of errors that are applicable when implementing DSLs: 'parsing errors' are errors that are related to the parsing of the DSL; 'transformation errors' are errors that occur during the transformation of ASTs; and 'run-time errors' are errors that occur at the time of execution of the transformed-to constructs. In Ruby, since the DSLs are essentially host language constructs, 'parsing' and 'transformation' errors are not applicable; 'run-time' errors are reported by the Ruby interpreter at run-time. In Stratego, 'parsing' errors are reported at `parse` stage of the transformation pipeline (Figure 2). However, transformations in Stratego can lead to cascading errors that are either reported at the transformation stage, when the application of a rule fails; or at post-transformation stages – the stages following the transformation stage but prior to the execution stage of the end language – when an AST that is invalid is pretty-printed or when the target program is compiled. Further, 'run-time errors' are detected only at the time of execution of the target program. In particular, 'transformation' and 'run-time' errors are hard to debug as one needs to manually trace the errors back to the rules in the transformation program.

Converge uses the concept of *src info* to report errors precisely, in terms of the source DSL. A src info records three pieces of information: a source file; the byte offset within the source file; and the number of bytes from the initial offset. Since the DSL (and the implementation function) are embedded within the host language constructs, 'parsing' and 'transformation' errors are reported at compile-

time. Further, the tokens, the AST elements and the bytecode instructions are associated with multiple src infos that enable 'run-time errors' to be reported with stack backtraces consisting of the error location within the translated-to Converge program, translation functions, and the DSL source. For instance, introducing an error in the guard expression of a transition by changing it from `credit + 1 == 3` to `credit + 1 == "3"` results in the following stack backtrace:

```
Traceback (most recent call at bottom):
 1: File "runfsm.cv", line 20, column 4, length 23
     turnstile.event("coin")
 ...
 4: File "FSM_Translator.cv", line 294, column 40, length 18
      return [<op.src_infos>| $c{lhs} == $c{rhs} |]
    File "runfsm.cv", line 12, column 69, length 2
      transition unlocking from locked to unlocked : coin [ credit + 1 == "3" ] / credit := 0
 ...
 5: (internal), in Int.<
Type_Exception: Expected arg 2 to be conformant to Number but got instance of String.
```

The fourth entry in the backtrace is related to multiple source locations: the third and fourth line indicates the location within the source DSL (`runfsm.cv`); and the others (only one is shown for brevity) are within the DSL translator (`FSM_Translator.cv`). Thus src infos provide useful debugging information to both the user and the DSL developer to determine the cause of an error. Further, quasi-quotes provide a syntactic extension in the form of `[<src_infos>| expr |]`, which allows the addition of extra src infos to an AST element, to provide customised errors to the user.

7 Discussion

Ruby and Converge both use an homogeneous embedded approach to implement DSLs. In Ruby, DSLs are implemented using its host language features; therefore, the implementation will be quick and the DSLs implemented will be lightweight in nature. Converge supports implementation of DSLs using its compile-time meta-programming facility. The close integration of the parser kit and the compile-time meta-programming facility with its host language, enables it to provide a systematic approach to implement DSLs. The concept of src infos is unique to Converge, which enables it to report errors precisely in terms of the source DSL. However, integrated DSLs in Converge are obviously distinct from normal language constructs, which can be aesthetically jarring.

In contrast to Ruby and Converge, Stratego/XT uses an heterogeneous embedded approach and supports implementation of DSLs through program transformation. A Stratego-like approach to implement DSLs provides a consistent mechanism to transforming programs between arbitrary languages. Stratego also supports context-sensitive transformation through the use of dynamic rewrite rules that facilitates the type checking on disjointed fragments within a DSL implementation. To use the concrete syntax of the object languages within transformation rules, their grammar definitions will have to be merged, thus creating potential ambiguities within the combined grammar that will have to be resolved manually. In a pipeline approach (Figure 2) to implement DSLs, the DSL author needs to be aware of: Hudak's [6] argument of 'cost versus benefits'; and the potential need to manually inspect the results (or errors) at the end of each stage as the different stages of the pipeline are unaware of each other.

In our experience, DSL programs are much more succinct in Converge or Stratego compared to Ruby. This is because the syntax of the DSLs in Converge (or Stratego) can be customised for the problem in hand; whereas Ruby's syntax can not be extended, inherently limiting the DSLs syntax. Therefore, DSLs in Converge and Stratego are better suited to projects where DSL usage is relatively high, and is not just a quick one-off use. Our experience in implementing the case study also highlighted that accurate sources of documentation with sufficient examples are essential to effective implementation of DSLs. Being a GPL, Ruby is extensively documented on the web (e.g. [7]) which the DSL author can make use of. Converge comes with examples on how to implement DSLs that can be used as a reference. Although there are plenty of documentation available for Stratego/XT, we noted that there is no single comprehensive guide (with examples) that focuses on DSL implementation.

8 Conclusions

In this paper, we implemented DSLs using three different embedded approaches; a weakened form of homogeneous embedded approach using Ruby; a heterogeneous embedded approach using Stratego; and a homogeneous embedded approach using Converge. Further, we presented a comparative study of the above approaches using a case study. From our comparative study we observed that each approach has its merits and demerits and there is no single approach that would apply to all scenarios. Nonetheless, we have highlighted strengths and weaknesses of three approaches that could serve as a guideline for future implementation of DSLs.

References

[1] Bentley, J., *Programming pearls: little languages*, Communications of the ACM **29** (1986), 711–721.

[2] Czarnecki, K., J. O'Donnel, J. Striegnitz, and W. Taha, *DSL Implementation in MetaOCaml, Template Haskell, and C++*, Domain Specific Program Generation, LNCS **3016** (2004), 51–72.

[3] Deursen, Arie V., P. Klint, and J. Visser, *Domain-Specific Languages: An Annotated Bibliography*, ACM SIGPLAN Notices **35** (2000), 26–36.

[4] Flanagan, D., and Y. Matsumoto, The Ruby Programming Language, O'Reilly Media, Inc., 2008.

[5] Fleutot, F., and L. Tratt, *Contrasting compile-time meta-programming in Metalua and Converge*, 3rd Workshop on Dynamic Languages and Applications, 2007.

[6] Hudak, P., *Modular Domain Specific Languages and Tools*, ICSR '98: Proceedings of the 5th International Conference on Software Reuse **0** (1998), 134–142.

[7] Documentation on the Ruby programming language, URL: http://ruby-doc.org

[8] Skalaski, K., M. Moskal, and P. Olszta, Meta-programming in Nemerle, Technical report, 2004.

[9] Bravenboer, M., K. T. Kalleberg, and E. Visser, Stratego Manual, URL: http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-chapter/index.html

[10] Tratt, L., *Domain Specific Language Implementation via Compile-Time Meta-Programming*, ACM TOPLAS **30** (2008), 1–40.

[11] Van Wyk, E., D. Bodin, L. Krishnan, and J. Gao, *Silver: an Extensible Attribute Grammar System*, ENTCS **203** (2008), 103–116.

[12] Visser, E., *Meta-Programming with Concrete Object Syntax*, GPCE02 LNCS **2487** (2002), 299–315.