

Comparative Study of DSL Tools

Naveneetha Vasudevan¹

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

Laurence Tratt²

*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 four approaches by implementing a common DSL case study. In addition, we present a comparative study of the four approaches.

Keywords: Domain Specific Languages, Parsing, Program Transformation.

1 Introduction

Domain Specific Languages (DSLs) are mini-languages tailored for a specific domain, offering significant advantages over General Purpose Languages (GPLs) [4]. 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 traditional approach to DSL implementation uses compiler tools such as Lex and YACC, or ANTLR, where DSLs are implemented ‘stand-alone’ for a particular application domain [1]. The main advantage of implementing DSLs

¹ e-mail: naveneetha@yahoo.com

² e-mail: laurie@tratt.net

using this approach is that the DSL author has complete control over the DSL, from its syntax to its style of execution. However, this approach is relatively little used due to high development costs [4]. Conversely, embedded approaches have been used to implement DSLs. Lisp and Nemerle [11] support construction of arbitrary program fragments at compile-time through the use of macros. In a pure embedding approach, where no macro-expanders or generators are used, DSLs are implemented as Domain Specific Embedded Languages (DSEs) using host language features such as higher-order functions and polymorphism [8].

Embedding approaches can be either homogeneous or heterogeneous [12]; 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 to implement DSLs [3,7], by allowing the user of a programming language to interact with the compiler to construct arbitrary program fragments at compile-time. Examples of heterogeneous embedding approaches are: Stratego/XT [2] supports implementation of DSLs through program transformation; and Silver [13] 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.

More recently, language workbenches are a new class of tools that provide a rich environment for building DSLs. Meta Programming System (MPS) [5] and the Intentional Domain Workbench (IDW) [15] are two such workbenches that typify this new class of tools. The workbenches essentially provide an Integrated Development Environment (IDE) with an underlying base language (IDW comes with *CL1* and MPS comes with *base language*) that supports the development, and integration of DSLs to implement a domain specific application.

In similar style to Czarnecki *et al.* [3], which evaluates the compile-time meta-programming abilities of three languages, we use a case study to evaluate four different approaches to DSL implementation. Our case study is a small but realistic DSL example of a state machine language. The four approaches we have chosen to study represent important, differing, points on the DSL implementation spectrum: ANTLR represents a traditional (non-embedded) approach to DSL implementation; Ruby typifies a weakened form of Hudak's vision of domain specific embedded languages; 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 a non-embedded approach and three ‘modern’ approaches to DSL implementation have been evaluated together and we hope this comparative study will benefit future implementation of DSLs.

The structure of the rest of this paper is as follows. Section 2 introduces the case study, which then provides the basis for our DSL implementation in ANTLR, Ruby, Stratego and Converge in sections 3, 4, 5 and 6 respectively. Section 7 presents a comparative analysis of the four DSL tools and their approaches based on a few selected dimensions and metrics. Section 8 presents our experiences on the relative strengths and weaknesses of the four DSL tools and Section 9 concludes this paper.

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.

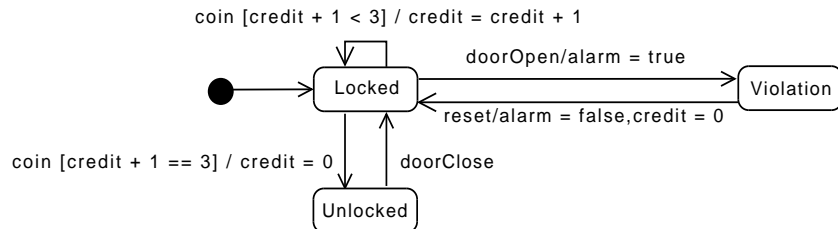


Fig. 1. State machine for a Turnstile

3 Implementation of a DSL in ANTLR

ANTLR [10] is a tool for generating parser generators. In ANTLR, the generated parser can be implemented as a translator in one of two forms: as a translator that parses the DSL program, executes the semantic actions and emits the output; or as a translator that parses the DSL program, and translates the parsed data to a target program using templates. Although both YACC [9] and ANTLR take a traditional approach to DSL implementation,

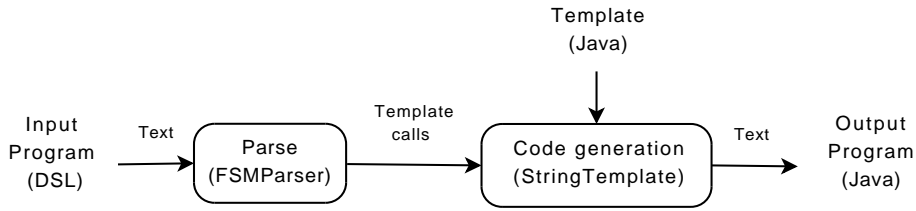


Fig. 2. The two stages required to implement DSLs in ANTLR

there are some subtle differences that are to be noted from a DSL implementation perspective. One such difference is that while in ANTLR, the parser can emit the output to a template to generate a target program, in YACC, this is no such facility available. This distinction highlights an important aspect: that the parser generated by ANTLR can be re-targetted for generating code in different programming languages. For the purposes of this paper, we discuss ANTLR as a translator that emits a target program.

In ANTLR, DSLs are implemented through translation (Figure 2). The translation is performed in two stages: the parsing stage where the input program is parsed and the parsed data is translated to template calls; and the code generation stage where these template calls are then mapped to the target language concepts. For the parsing stage, ANTLR provides the necessary libraries to generate the lexer and the parser. To generate the source code of the target program, ANTLR supports the use of *StringTemplate*—a template engine library for generating text using templates. A template is essentially a text document with template rules. A template rule contains ‘placeholders’ (expressions delimited by `< >` or `$ $`) that tell the template engine where to put the data. To generate target language constructs, a parser rule (in the grammar definition) is mapped to a template rule, which at run-time will result in the parser making a call (with data and template) to the template engine to generate the code associated with the template rule. A code fragment showing the domain specific information for a transition (from our case study) is as follows:

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

The corresponding parser and template rule (`transition` and `transition(tname,from,to,event)` respectively) for the above ‘transition’ construct is shown below:

```
transition
scope {
  String name;
  ...
  String event;
}
: 'transition' tname=ID {$transition::name=$tname.text;}
...
event=ID {$transition::event=$event.text;}
ttail {$prog::guards.add($ttail.st);} NEWLINE
```

```

-> transition(tname={new StringTemplate($tname.text)},
  ...
  event={new StringTemplate($event.text)})
;

transition(tname,from,to,event) ::=
  "this.transitions.add(new Transition(\"<tname>\",...,\"<event>\"));";

```

The above code highlights three more aspects. First, the use of an *action* within a parser rule. An action (`$transition::name=$tname.text;`) is a block of source code enclosed in curly braces, which is used to generate output or construct trees, or modify a symbol table. Second, the use of `scope` within a parser rule (`transition`) to define attributes (`name`). Attributes enable data to be shared between parser rules. Defining the `scope` within a parser rule allows the attributes to be accessed from within the rule actions, and also from the rules invoked by the `transition` parser rule (`ttail` in this case). Third, the construct `$prog::guards.add($ttail.st);` shows how the template output (`$ttail.st`) from the `ttail` rule is shared with the `prog` parser rule using the attribute `guards` that is defined within the `prog` parser rule.

4 Implementation of a DSL in Ruby

Ruby is a dynamically-typed, general purpose object-oriented language [6]. 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
...
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)
  ...
end
```

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

  def load_block
    yield self
  end
  ...
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 run-time that can then be used to dynamically manipulate the behaviour of an object.

5 Implementation of a DSL in Stratego/XT

Stratego/XT [2] is a software transformation framework that consists of the Stratego language (for implementing program transformations through term

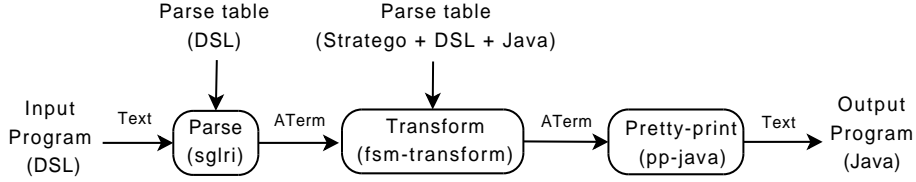


Fig. 3. The transformation pipeline in Stratego showing the various stages to implement DSLs

rewriting) and the XT toolset (for providing the infrastructure to implement these transformations). Stratego/XT achieves program transformation by representing programs in the form of abstract syntax trees, called Annotated Terms (ATerms); and then exhaustively applying a set of strategies and term rewrite rules to them.

In Stratego/XT, DSLs are implemented using a transformation pipeline (Figure 3) 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. For the parsing and the pretty-printing stages, the XT toolset provides the necessary tools (*sglri* and *pp-java* respectively). For the purposes of this paper, 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 an ATerm and is of the form $L : p1 \rightarrow p2$, 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. In Stratego, a user-defined strategy is defined using built-in strategies; for instance, a user-defined strategy ‘simple’ is defined as `simple = topdown(repeat(R1 <+ R2))`, where `topdown` and `repeat` are built-in strategies, and $R1$ and $R2$ are transformation rules. A term-rewrite rule can be written using the concrete syntax of the object languages rather than explicit ATerm constructors [14]. For instance, the assignment of an expression to a variable is expressed in concrete syntax as `| [x := e] |` rather than `Assign(Var(x), Expr(e))`. To use concrete syntax within term-rewrite rules, the Stratego meta-language has to be extended with the Syntax Definition Formalism (SDF) definition of the object language. Therefore, for our case study, where we wish to transform code fragments from DSL to Java, this involves merging the SDF definitions of Stratego (provided by Stratego compiler), Java (provided by Java-front [14]), and our DSL. Further, patterns corresponding to the syntactic elements (such as identifiers, expressions and lists) of the object language can be defined as meta-variables as part of the SDF definition. These meta-variables can then be used as variables to splice

in meta-level expressions within the concrete syntax of the object language in transformation rules. A condensed version of the SDF definition, showing the definitions of the meta-variables (`ttail` and `guard`) for our DSL is as follows:

```
variables
  "ttail" [0-9]          -> TransitionTail   {prefer}
  "guard" [0-9]         -> Guard             {prefer}
  ...
```

A code fragment showing the condensed version of our DSL program and the corresponding transformation rules showing the use of concrete syntax with embedded meta-variables (`ttail1` and `guard1`) is 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 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.

6 Implementation of a DSL in Converge

Converge [12] 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 [12]. 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 7.

7 Analysis and Comparison

In this section, we present a few dimensions (section 7.1) and metrics (section 7.2), that then serves as a basis for comparing the four DSL tools.

7.1 Dimensions

We use and extend the dimensions identified by Czarnecki *et al.* [3] to present a comparative analysis of the four DSL tools. The dimensions are listed in Table 1.

Dimension	ANTLR	Ruby	Stratego/XT	Converge
Approach	Translation	Lambda abstractions	Term rewriting	Compile-time meta-programming
Guarantees	No	Syntax valid (run-time)	No	Well-typed (compile-time)
Reuse	Limited	Limited	SDF grammar	Limited
Context-sensitive transformation	Yes	No	Yes	No
Error reporting	Limited (end language)	Yes (run-time)	Limited (end language)	Yes (compile-time)

Table 1
A comparative analysis of ANTLR, Ruby, Stratego, and Converge

Approach What is the primary approach supported by the DSL tool? In ANTLR, DSLs are implemented through translation using a template engine, where the source program (DSL) is parsed, and the data is fed to the template engine to generate the target program. In Ruby, DSLs are implemented using a combination of its host language features such as lambda abstractions, dynamic typing, and reflection. 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 3) at the preprocessor stage—a stage prior to the compilation of the target language program. In Converge, DSLs are implemented using its compile-time meta-programming facility, where the DSL constructs are translated to the host language constructs at compile-time.

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 refers to the guarantees that are provided by the DSL tool in generating a syntactically valid target program. 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 ANTLR, the parsed data is fed to the template engine which then generates the target program for a given template. There are few guarantees that can be given with respect to the well-formedness of the generated program: first, the template engine is unaware of the type of data that is being pushed from the parse; second, the well-formedness of the generated program depends on the syntactic and semantic well-formedness of the constructs within the template. 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 re-used? We identify two aspects that are potentially re-usable: the grammar of the DSL; and the transformation module. In ANTLR, the grammar has limited re-use because the parser rules are interspersed with semantic actions and template calls. However, the use of templates for code generation enables a clear separation between data (DSL) and logic (parser) from presentation (template) that allows code generation for multiple target languages. In Ruby, since the DSLs are essentially host language constructs, the aspect related to the grammar does not apply. Further, 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.

Context-sensitive transformation Can the DSL tool perform context-sensitive transformation? We explain context-sensitive transformation 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 perform the trans-

formation of the `SELECT` statement based on the definition of the `CREATE` statement? In ANTLR, context-sensitive transformation is possible by using named scopes. For our SQL scenario, an attribute can be defined within a named scope which can be then be initialised at the time of the invocation of the parser rule corresponding to the ‘create’ statement. The parser rule for the ‘select’ statement can then lookup the attribute to retrieve the definition of ‘create’ statement. In Ruby, context-sensitive transformation is only possible by layering an external program that can then be invoked prior to the invocation of the host language interpreter. In Stratego, however, term rewriting can be extended with dynamic rules to perform context-sensitive transformation. For our SQL scenario, a dynamic rule can be defined within the context of the ‘create’ statement to perform context-sensitive transformation, which can then be invoked by the transformation rule corresponding to the `SELECT` statement. In Converge, context-sensitive translation can be only be performed by implementing an external program which can then be invoked at the time of translation.

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 in Table 2. For the purposes of this paper, ‘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.

Error category	ANTLR	Ruby	Stratego	Converge
Parsing errors	Parse-time	Run-time	Parse-time	Compile-time
Transformation errors	End language compile-time	n/a	Transformation, pretty-printing, or end language compile-time	Compile-time
Run-time errors	End language run-time	Run-time	End language run-time	Host language run-time

Table 2

A comparison of the error reporting capabilities of ANTLR, Ruby, Stratego, and Converge

In ANTLR, ‘parsing errors’ are reported at the parse stage (Figure 2) of the translation with line and column number of the source program (DSL). In ANTLR, the data that is obtained from the parser is fed to the template engine along with a template, which then generates the target program. Therefore, any errors related to the translation are reported only at the time of the compilation of the target program. ‘Run-time errors’ are reported at the time of execution of the target program. Although the ‘runtime’ errors are reported at the time of execution of the target program, the errors can be manually traced back to the definitions in the grammar by using the comments in the generated parser (a parser rule has a corresponding method

in the parser and this is noted as a comment). 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 3). 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.

7.2 Metrics

We identify and present two metrics in Table 3, using which we compare the four DSL tools.

Dimension	ANTLR	Ruby	Stratego/XT	Converge
Lines of code (grammar, transformation, and DSL program)	94, 109, 12	n/a, 88, 55	79, 95, 12	36, 164, 11
Aspects to learn	2	1	4	2

Table 3
A comparative analysis of ANTLR, Ruby, Stratego, and Converge based on metrics

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 were significantly higher in ANTLR and Stratego as compared to Converge. This is because in ANTLR, the parser rules are augmented with semantic actions and template calls, and in Stratego, there are additional SDF definitions for meta-variables. In Ruby, since the DSLs are essentially host language constructs, there is no grammar implementation. The size of the transformation (or translation) program are much more concise in ANTLR and in Stratego as compared to in Converge. This is because in ANTLR and in Stratego, multiple nodes in the AST are transformed through the application of a template rule and a strategy respectively, whereas in Converge, the nodes in the AST are traversed (and translated) systematically. Therefore, for our case study, where ‘states’ and ‘transitions’ are essentially a list of nodes in the AST, the application of a template rule (or a strategy) will result in a smaller transformation program. It should be noted that in ANTLR and in Stratego, the size of the transformation program will also be determined by the verbosity of the target language. In Ruby however, the DSL programs are evaluated as is, resulting in the size of the evaluation program to be generally smaller as compared to Stratego or Converge. The size of the DSL input program in Ruby was well over four times the size of the input program in the other DSL tools. This is primarily because the syntax of the DSLs in Ruby is limited to that which can be naturally expressed by the host language whereas in the other three DSL tools, the syntax of the DSLs are specifically designed for the problem in hand.

Aspects to learn For a given tool, how many aspects need to be learned to implement DSLs? In ANTLR, there are two aspects that needs to be learned: the ‘grammar’ aspect for defining the lexer and parser rules; and

the ‘template’ aspect for generating the target program. In Ruby, DSLs are implemented using the host language constructs, and therefore there is only one aspect to be learned—the Ruby language. In Stratego, DSLs are implemented using a pipeline framework that requires learning of many different aspects: the ‘grammar’ aspect for defining the SDF definitions for the DSL; the ‘meta-variable’ aspect for defining the syntactic elements of the object language as variables; the ‘term-rewrite’ aspect for implementing the transformation program; and finally the ‘pipeline framework’ aspect to understand how the different stages of the transformation pipeline work together. In Converge, to implement DSLs, two aspects need to be learned: the ‘grammar’ aspect for defining the lexer and parser rules; and the ‘compile-time metaprogramming’ facility to perform translation.

8 Discussion

ANTLR uses a non-embedded approach to implement DSLs. ANTLR comes with ANTLRWorks [10], a grammar development environment with editing and debugging facilities that allows developers to quickly prototype and test their DSLs. The use of scopes allows data to be shared between the parser rules, which then enables context-sensitive translation. ANTLR supports the use of templates that enforces separation of data (DSL) and logic (parser) from presentation (template). This separation allows the parser to be re-used to generate target programs based on templates that target different languages. However, this separation also means that the data that is passed from the parser to the template through template calls can contain invalid constructs, which can then result in the generation of a syntactically invalid target program.

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. Stratego’s approach to DSL implementation provides a consistent mechanism to transform programs between arbitrary languages. Strat-

ego 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 implementing DSLs using a pipeline approach (Figure 3) with a relatively higher learning cost, the DSL author needs to be aware of two caveats. First, the Hudak’s [8] argument of ‘cost versus benefits’ in using DSLs for software development. Second, the potential need for manually inspecting the results (or errors) at the end of each stage, particularly when the different stages of the pipeline are unaware of each other.

In our experience, DSL programs are much more succinct in ANTLR, Converge, and Stratego as compared to DSL programs in Ruby. This is because the syntax of the DSLs in ANTLR, Stratego and Converge can be customised for the problem in hand, whereas Ruby’s syntax can not be extended, inherently limiting the DSLs syntax. Therefore, DSLs in ANTLR, Stratego and Converge are better suited to projects which satisfy two criteria: first, the syntax of the DSL is important; and second, where the application of the DSL for implementing domain specific solutions 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 open source, ANTLR and Ruby are extensively documented on the web which the DSL author can make use of. Although there is plenty of documentation available for Stratego/XT, we noted that there is no single comprehensive guide (with examples) that focuses on DSL implementation. Converge comes with examples on how to implement DSLs that can be used as a reference.

9 Conclusions

In this paper, we implemented DSLs using a non-embedded approach and three different embedded approaches. The non-embedded approach showed the traditional method of implementing of DSLs using ANTLR. The three different embedded approaches include: 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 four 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] Bravenboer, M., K. T. Kalleberg, R. Vermaas, E. Visser, *Stratego/XT 0.16: components for transformation systems*, ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM’06), ACM SIGPLAN (2006), 95–99.
- [3] 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.
- [4] Deursen, Arie V., P. Klint, and J. Visser, *Domain-Specific Languages: An Annotated Bibliography*, ACM SIGPLAN Notices **35** (2000), 26–36.
- [5] Dmitriev, S., “Language Oriented Programming: The Next Programming Paradigm,” Technical report, JetBrains, 2004.
- [6] Flanagan, D., and Y. Matsumoto, “The Ruby Programming Language,” O’Reilly Media, Inc., 2008.
- [7] Fleutot, F., and L. Tratt, *Contrasting compile-time meta-programming in Metalua and Converge*, 3rd Workshop on Dynamic Languages and Applications (2007).
- [8] Hudak, P., *Modular Domain Specific Languages and Tools*, ICSR ’98: Proceedings of the 5th International Conference on Software Reuse **0** (1998), 134–142.
- [9] Levine, J., T. Mason, and D. Brown, “Lex & Yacc,” 2nd Ed., O’Reilly Media, Inc., 1992.
- [10] Parr, Terence, “The Definitive ANTLR Reference: Building Domain-Specific Languages,” The Pragmatic Bookshelf, 2007.
- [11] Skalaski, K., M. Moskal, and P. Olszta, *Meta-programming in Nemerle*, Technical report, 2004.
- [12] Tratt, L., *Domain Specific Language Implementation via Compile-Time Meta-Programming*, ACM TOPLAS **30** (2008), 1–40.
- [13] Van Wyk, E., D. Bodin, L. Krishnan, and J. Gao, *Silver: an Extensible Attribute Grammar System*, ENTCS **203** (2008), 103–116.
- [14] Visser, E., *Meta-Programming with Concrete Object Syntax*, GPCE02 LNCS **2487** (2002), 299–315.
- [15] Documentation on Intentional Domain Workbench, URL: <http://blog.intentsoft.com/intentional.software/2009/05/dsl-devcon.html>