Comparative Study of DSL Tools

Naveneetha Vasudevan[1]

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

Laurence Tratt[2]

Middlesex University
The Burroughs, Hendon, London, NW4 4BT, 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, which can offer significant advantages over General Purpose Languages (GPLs) such as Java [5]. When developing software systems in a GPL, one often comes across situations where a problem is not naturally expressible in the chosen GPL. Traditionally one then resorts to finding a suitable workround (within the framework provided by the GPL) to encode the solution. One of the drawbacks of using such a workaround is that the program can become complex, thus making it far less comprehensible than the developer had wished for. The lack of expressivity in a GPL can be overcome by

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

using DSLs. DSLs allow programs to be implemented at the level of abstraction of the application domain which enables quick and effective development of software systems. Given a domain and the need for a DSL, there exist a number of tools and approaches to implement DSLs. The traditional approach involves implementing DSLs as 'stand-alone' systems using compiler tools such as Lex and YACC, or ANTLR [1]. Such an approach provides the DSL author with complete control over the DSL, from its syntax to its style of execution, but leads to high development costs as each implementation tends to be engineered from scratch [5].

In contrast to the traditional technique, an embedding approach – where DSLs are implemented by embedding them within a host GPL – can also be used. An embedding approach allows the DSL to inherit the infrastructure of the host language, and thus facilitating the reuse of the software artifacts (such as syntax, semantics etc.) leading to reduced software development cost.

Embedding approaches can be either homogeneous or heterogeneous [13]: 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 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. Examples of heterogeneous embedding approaches are: Stratego/XT [2], which supports the implementation of DSLs through program transformation; and Silver [14] which supports the 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. Among homogeneous embedding approaches: Lisp and Nemerle [12] support the development of embedded languages through the use of macros; in a pure embedding approach – where no macro-expanders or generators are used – DSLs are implemented using host language features such as higher-order functions and polymorphism [9]; compile-time metaprogramming has been used to implement DSLs [4,8], by allowing the user of a programming language to interact with the compiler to construct arbitrary program fragments at compile-time.

More recently, a new class of tools *language workbenches* has emerged, which provide a rich environment for building DSLs. The Meta Programming System (MPS) [6] and the Intentional Domain Workbench (IDW) [16] are two workbenches that typify this new class of tools. The workbenches essentially provide an Integrated Development Environment (IDE) with underlying base languages. For instance, IDW comes with *CL1* language; and MPS comes with three base languages: *structure* for defining the abstract syntax of a language; *editor* for defining the concrete syntax of a language; and *semantics* for defining the semantics of a language. Using these language building

tools, DSLs can be developed and integrated to implement a domain specific application.

In this paper we evaluate four approaches to DSL implementation. In similar style to Czarnecki *et al.* [4], which evaluates the compile-time meta-programming abilities of three languages, we use a case study to evaluate these approaches. Our case study is a small but realistic DSL example of a state machine language. Although our work involves a single case study, the DSL implemented for our case study is indicative of a much wider range of DSLs which have been implemented thus far. The four approaches we have chosen to study represent important, differing, points on the DSL implementation spectrum: ANTLR represents a traditional stand-alone 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 stand-alone approach and three 'modern' approaches to DSL implementation have been evaluated together and we hope this comparative study will benefit future users and implementers of DSLs and DSL tools.

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 4, 5, 6 and 7 respectively. Section 8 presents a comparative analysis of the four DSL tools and their approaches based on selected dimensions and metrics. Section 9 then presents our experiences of the relative strengths and weaknesses of the four DSL tools.

2 Case Study: Finite State Machine

The case study used in this paper is a state machine (Figure 1) of a turnstile machine 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. We implement this case study in different approaches, in each creating an executable state machine that we can fire events at and examine its behaviour. For each approach, we define a state machine language (in a syntax appropriate to that approach), and then implement the state machine language for the turnstile machine.

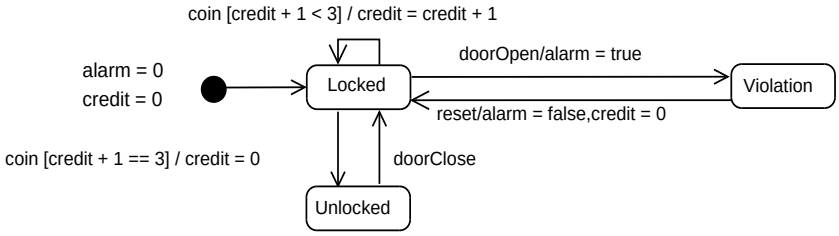

Fig. 1. State machine for a Turnstile

3 Dimenstions and Metrics

In order to evaluate the four implementations of our case study, we use a set of dimensions and metrics. For the purposes of this paper, a 'dimension' refers to a property of a DSL implementation that can not be measured quantitatively whereas a 'metric' refers to a property that can (and therefore numerical data can be extracted from the DSL implementation). We use and extend the dimensions (Table 1) identified by Czarnecki *et al.* [4] to present our comparative analysis. We then identify and define two metrics (Table 2) with which we extend our analysis.

Dimension	Description
Approach	What is the primary approach supported by the DSL tool?
Guarantees	What guarantees are provided by the DSL tool in terms of syntactic and semantic well-formedness of the transformed-to constructs?
Reuse	Can the 'user-defined' aspects of the DSL implementation be reused?
Context-sensitive transformation	Can the DSL tool perform context-sensitive transformation?
Error reporting	Can the DSL tool report errors in terms of the DSL source (line number and column offset)?

Table 1
List of dimensions

Metric	Description
Lines of code	For a given case study, how many lines of code are required to represent the domain-specific information?
Aspects to learn	For a given case study, how many aspects need to be learned to implement a DSL?

Table 2
List of metrics

4 Implementation of a DSL in ANTLR

ANTLR [11] is a parser generator tool that provides a framework for implementing language translators. In ANTLR, the generated parser can be implemented as a translator in one of two forms: it can execute the semantic

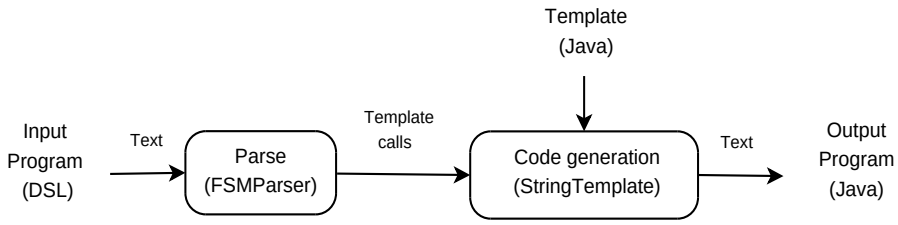

Fig. 2. The two stages required to implement DSLs in ANTLR

actions; or it can execute the semantic actions to generate a target program using templates. For the purposes of this paper, we discuss ANTLR as a translator that emits a target program.

We have chosen to implement DSLs in ANTLR through translation (Figure 2) using templates. The translation process has two stages: the parsing stage where the input program is parsed and the parsed data is fed as arguments 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 program for a given grammar. To generate 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 where each rule contains 'placeholders' (expressions delimited by `< >` or `$ $`) that tell the template engine where to put the data. Although ANTLR supports code generation for only a handful of GPLs, there is an open source community for developing code generation libraries for new target languages. For our case study we translate code fragments from our DSL to Java (target language). We explain the two stage translation process using the 'transition' construct (from our DSL program) as an example. 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 rules (`transition` and `ttail`) for the above 'transition' construct are shown below:

```
transition
scope {
  String name;
  ...
  String event;
  List actions;
  boolean isguard;
}
    :   'transition' tname=ID {$transition::name=$tname.text;}
        ...
        ':' tevent=ID {$transition::event=$tevent.text;}
        ttail {$prog::guards.add($ttail.st);} NEWLINE
        -> transition(tname={new StringTemplate($tname.text)},
                      ...
                      event={new StringTemplate($tevent.text)})
```

```
    ;

ttail
@init {
  $transition::actions = new ArrayList();
  $transition::isguard = false;
}
    :   g=guard? actionstats?
        -> {$transition::isguard}?
           guardBlock(guardcond={$g.st},
                      t_name={$transition::name},
                      ...
                      t_event={$transition::event},
                      actions={$transition::actions})
        -> guardBlock(guardcond={new StringTemplate("true")},
                      t_name={$transition::name},
                      ...
                      t_event={$transition::event},
                      actions={$transition::actions})
    ;
```

Each element (on the RHS) of a parser rule (`transition` and `ttail` in the above code) can be followed by an *action*. An action is a block of source code written in the target language (and enclosed in curly braces) that is used to generate output or construct trees, or modify a symbol table. An action is executed immediately after the preceeding element has been matched. For instance, in the above `transition` parser rule, when the element `ID` (`tname=ID` construct) is matched, it results in the action – `$transition::name=$tname.text;` – to be executed. This action initialises the attribute `name` defined in the `scope` block of the transtition rule. In ANTLR, there are essentially two type of scopes: a named global scope defined outside any rule; and a rule scope (unnamed) defined within a rule. A global scope is named and therefore any rule can access it by its name whereas a rule scope is accessible only to the current rule and to all the rules invoked by it. Scopes provides a mechanism to exchange data between parser rules. For instance, the attribute `name` that was initialised when processing the action (`$transition::name=$tname.text;`) from the `transition` rule can now be accessed within the `ttail` (`guardBlock(guardcond=$g.st,t_name=$transition::name,...)`) rule for generating target language constructs.

In ANTLR, to generate a target language construct, a parser rule needs to be mapped to a template rule. Then, given a template containing the definition of the template rule, the parser (at run-time) will invoke the template engine to generate the necessary constructs in the target language. For the above `transition` and `ttail` rules, template rules – `transition(...)` and `guardBlock(...)` – will be invoked. The template rules (`transition(...)` and `guardBlock(...)`) defining the constructs in the target language (Java) are shown below:

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

```
guardBlock(guardcond,t_name,t_from,t_to,t_event,actions) ::= <<
    if (transition_name.equals("<t_name>") &&
        ...
        transition_event.equals("<t_event>")) {
        if ( (<guardcond>) && ... ) {
                //actions here
                <actions; separator="\n">
                _guard = true;
        }
    }
>>
```

5 Implementation of a DSL in Ruby

Ruby is a dynamically-typed, general purpose object-oriented language [7]. In Ruby, DSLs are implemented using a combination of features such as lambda abstractions (code blocks), evaluations, dynamic typing, reflection and flexible syntax. We explain how these features combine to implement the 'transition' construct from our DSL program as an example. 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 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 is essentially an invocation of the method – `transition` – followed by two arguments: a string, and a code block that accepts a parameter (`|t|`). In Ruby, invoking a method requires a context in the form of an instance of an object or a class. For our example where we want to execute the `transition` method, the context is provided by an instance (`@fsm = Fsm.new`) of the state machine class (`Fsm`). To execute the `transition` method for the `fsm` instance, Ruby's evaluation method *instance_eval* can be used. The instance_eval method allows a string or a code block to be evaluated in the context of an instance of a class. Therefore, for our case study, where the `transition` constructs are contained within a text file, we can read the file as a string and then evaluate the constructs for the `fsm` instance. A code fragment showing the definition of the `load` method that evaluates the DSL program using the instance_eval method and the definition of the `transition` method is as follows:

```
class Fsm
  # takes file (DSL program) as an argument
  def load(fsm_dsl)
    instance_eval(File.read(fsm_dsl),fsm_dsl)
    ...
  end

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

In Ruby, methods accept a code block as a final argument. However, if a method is defined with a *block argument* (an ampersand-prefixed final argument of the form `&aBlock`), then a code block (supplied as an argument to the method) will be implicitly converted to a *Proc* object. A Proc object is essentially a Ruby object representing a block of code which can then be passed around as an object and executed either by using `yield` or by invoking its `call` method (any arguments passed to the `call` method will be assigned to the block parameters). In the above code, since the `transition` method defines a block argument (`&aBlock`), the Proc object associated with it is passed as an argument to the `load_block` method of the `transition` object. 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):

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

  def to_state=(to_state)
    @to_state = to_state
  end

  def guard(&guardBlock)
    @guard_block = guardBlock
  end

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

The above code also shows the corresponding method definitions (`from_state(from_state)` and `to_state=(to_state)`) for the transition attributes (`t.from_state 'locked'` and `t.to_state = 'locked'`). The two variant style of invoking (and defining) methods – with and without the equal sign – is indicative of an important aspect of Ruby as a DSL tool: syntactic flexibility. In addition to code blocks, Ruby supports dynamic typing, which allows the runtime system to implement features such as dynamic dispatch

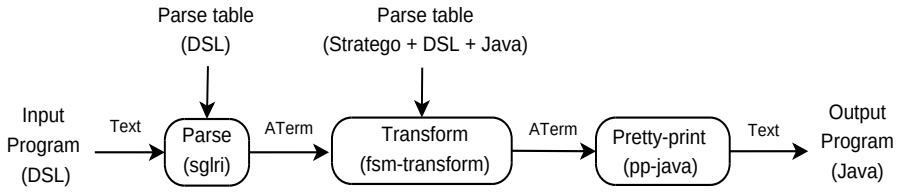

Fig. 3. The transformation pipeline in Stratego showing the various stages to implement DSLs

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.

6 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 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, tools (*sglri* and *pp-java* respectively) from the XT toolset can be used. 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 -> p2`, where `L` is the rule name, and `p1` and `p2` are term patterns. A strategy is a program that supports the application of rules to an ATerm by defining the order in which the terms are re-written. For instance, to apply rules R1 and R2 sequentially for a single top-to-bottom traversal on an AST, a `topdown` strategy – denoted by `topdown(R1 <+ R2)` – can be used. To apply these rules repeatedly for a single top-to-bottom traversal, the `topdown` strategy can apply the `repeat` strategy – denoted by `topdown(repeat(R1 <+ R2))` – which can then invoke the rules (R1 and R2) until no more rule applies. Further, built-in strategies can be combined to define a user-defined

strategy; for the above example, a user-defined strategy 'simple' can be defined as `simple = topdown(repeat(R1 <+ R2))`. The ability to define strategies is useful in two ways: first, it enables the reuse of rules and strategies; and second, it enables abstraction by masking the low-level actions on an AST.

In stratego, a term-rewrite rule can be written either by using nested ATerms or by using the concrete syntax of the object language [15]. For instance, the assignment of an expression to a variable can be expressed using nested ATerms (`Assign(Var(x),Expr(e))`) or using the concrete syntax of the object language (`|[ x := e ]|`). To use concrete syntax within term-rewrite rules, the Stratego meta-language has to be extended with the grammar definition of the object language. For our case study, where we want to transform code fragments from DSL to Java, this involves merging the grammar definitions of Stratego (provided by Stratego compiler), Java (provided by Java-front [15]), and our DSL. Further, the grammar definition of an object language can be extended with meta-variables (patterns corresponding to the syntactic elements such as identifiers, expressions and lists of the object language) which can then be used as variables to splice in meta-level expressions within the object language constructs in a transformation rule. For the 'transition' construct where we want to create meta-level expression for the `TransitionTail` and `Guard` elements, meta-variables (`ttail` and `guard`) are defined as part of the grammar definition. A condensed version of the grammar definition, showing the definition for the `TransitionTail` and `Guard` elements and their corrsponding definition of the meta-variables (`ttail` and `guard`) is as follows:

```
context-free syntax
  ...
  "transition" Id "from" Id "to" Id ":" Id TransitionTail -> Transition {cons("Transition")}
  TransitionTailG | TransitionTailGA | TransitionTailA -> TransitionTail {cons("TransitionTail")}
  Guard                          -> TransitionTailG {cons("TransitionTailG")}
  "[" Exp "]"                    -> Guard {cons("Guard")}

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

The above definition of the variables allow us to replace the DSL constructs corresponding to the `TransitionTail` and `Guard` elements in a transformation rule with meta-variables. The following code fragment shows the 'transition' construct (from our DSL program) and a subset of the transformation rules with embedded meta-variables (`ttail1` and `guard1`):

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

guard-init : |[ transition x_t from x_a to x_b : x_e ttail1 ]| ->
             |[ if ((transitionName.equals("~x_t") && ...) { bstm_1 } ]|
             where <trans-tail> ttail1 => bstm_1
trans-tail : trans-tail |[ guard1 ]| ->
                        |[ if ( e_1 ) { _guard = true; return _guard; } ]|
                        where <guard> guard1 => e_1
```

```
trans-tail : trans-tail |[ action1 ]| ->
                        |[ if ( true ) { bstm_1* _guard = true; return _guard; } ]|
                        where <action> action1 => bstm_1*
...
```

The use of a *where* clause in a transformation rule enables the programmable application of rules. For instance, the `<trans-tail> ttail1` construct within the where clause of the `guard-init` rule, will invoke either of the `trans-tail` rule, depending upon the value of `ttail1` at run-time. The value returned from invoking the `trans-tail` rule on the `ttail1` ATerm is assigned to the `bstm_1` meta-variable which is then spliced back into the transformation rule to generate the target language construct (the grammar definition of the Java language defines `bstm_1` as a meta-variable).

7 Implementation of a DSL in Converge

Converge [13] 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 [13]. Splice annotations `$<...>` evaluate the expression between the angled brackets, and replace the splice annotation itself with the result (AST) of its evaluation. For instance, the splice annotation `$<x>` tells the compiler to evaluate 'x' at compile-time and replace it with the result (AST) of that evaluation. Quasi-quotes `[|...|]` allows the user to build ASTs that represent the expression inside it. For instance, while the Converge expression `2 + 3` evaluates to 5, the quasi-quoted expression `[| 2 + 3 |]` evaluates to an AST of the form `add(int(2),int(3))`. Insertions `${...}` are splice annotations placed within quasi-quotes. Splices within quasi-quotes are evaluated differently to splices outside quasi-quotes. They are not evaluated at compile-time but copied as-is into the code that the quasi-quote transforms to. For instance, the quasi-quoted expression `[| $<x> + 2 |]` would result in an AST along the lines of `add(x,int(2))`, where `x` must evaluate to a valid AST.

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 SM_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 (Src infos are covered later in Section 8). Using the Converge Parser Kit (CPK) this string is parsed to produce a parse tree. This parse tree containing tokens and their associated src infos, is traversed and translated to an AST using quasi-quotes and insertion. The CPK provides a simple framework – a generic parse tree `Traverser` class – to perform this translation: for each node n in the parse tree that requires translation, a corresponding translation function `_t_`n should be defined. Given a node in the parse tree, the `self._preorder` method can then be used to call the appropriate `_t_` function. For our case study we implement `SM_Translator` class (inherits `Traverser::Traverser` class) that contains the necessary `_t_` functions (`_t_system`, `_t_transition` etc.). The `generate` function initiates the translation process by invoking `_t_system` function (`system` is the top-level rule for our grammar). A code fragment showing the definition of the `SM_Translator` class and the use of `self._preorder` method to invoke the necessary `_t_` function depending upon the value of the node is as follows:

```
class SM_Translator(Traverser::Traverser):

    func generate(self, node):
        return self._preorder(node)

    func _t_system(self, node):
        sts := [] // States
        tns := [] // Transitions
        ...

        while i < node.len():
            ndif node[i][0].name == "state":
                sts.append(self._preorder(node[i])) // invokes _t_state function
            elif node[i][0].name == "transition":
                tns.append(self._preorder(node[i])) // invokes _t_transition function
            ...

        return [|
            class:
                states := ${CEI::ilist(sts)}
                transitions := ${CEI::ilist(tns)}
```

```
            ...

            func event(self, e):
                ...
    |]

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])
        guard := self._preorder(tail_node[2])
        ...
    else
        ...
```

The `generate` function from the above code fragment returns an anonymous class that is essentially a representation of the DSL program. Converge provides Compiler External Interface (CEI) package to interface with the compiler. The CEI package provides a range of functions to create an AST without using Quasi-quotes for code fragments which can't be expressed using concrete syntax (e.g. an `if` statement with an arbitrary number of `elif`s). In the above code fragment the construct – `CEI::ilist(tns)` – essentially returns an AST containing a list of transition objects. The anonymous class returned from the `generate` function can then be instantiated to produce a running state machine `turnstile := TurnstileFSM.new()`, which can receive and act upon events (`turnstile.event("coin")`).

8 Analysis and Comparison

8.1 Dimensions

In this section, we present our comparative analysis of the four DSL tools based on the dimensions listed in Table 1. Table 3 compares the DSL tools, based on which we present our analysis. We also provide our views from an end user perspective on how the DSL tools compare for each of these dimensions. For the purposes of this paper an end user refer to a developer implementing the DSL translation program.

Approach 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

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

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

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.

Based on the experience in implementing our case study, we believe each of these approaches have their strengths and weaknesses. Among embedding approaches: although pure embedding approach (e.g. Ruby) provide a much quicker way of implementing DSLs, the syntax of the DSL will be limited by the syntax of the host language; heterogeneous approach (e.g. Stratego) supports code generation to any target language but the end user might face a much steeper learning curve; and finally a homogeneous approach (e.g. Converge) provides a middle ground in that it requires much less learning but DSLs can only be translated to the host language.

Guarantees In the context of this paper, the guarantees that an approach can provide relate to syntactic or semantic well-formedness. Although there are potentially many different semantic guarantees that could be offered, we consider only the following two (since the errors related to them were the more prominent ones for our case study): 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 syntactic and semantic 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 inter-

preter. 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 syntactic and semantic well-formedness of the translated-to host language constructs at the time of translation.

From an end user perspective, one wishes to minimise errors related to syntactic and semantic well-formedness. Since in Ruby and Converge errors related to well-formedness are reported (at run-time and compile-time respectively), diagnosing such errors is lot easier. In ANTLR and Stratego, since well-formedness errors are reported only when compiling the end language, users may have to revisit the transformation program or the grammar definition to determine the source of the offending construct (see the *error reporting* dimension).

Reuse We identify two aspects that are potentially reusable: the grammar of the DSL; and the transformation module. In ANTLR, the grammar has limited reuse because the parser rules are interspersed with semantic actions and template calls. However, ANTLR supports the use of templates for code generation that enables a clear separation between data (DSL) and logic (parser) from presentation (template). This, for a given grammar, allows code to be generated 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 reusability 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 reuse. 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 reused for other DSL implementations.

From an end user perspective, one wishes to maximise the reusability of the user-defined aspects. In ANTLR the reusability of the grammar for multiple targets is useful in cases where code needs to be generated for multiple languages. In Ruby and Converge, the DSL constructs are embedded within the host languge program thus making it difficult to reuse any of the user defined aspects. In Stratego, when code needs to be generated for multiple object languages, the grammar definitions related to meta-variables and sections of the transformation program related to type and expression sub-systems can potentially be reused.

Context-sensitive transformation For the purposes of this paper, we define context-sensitive transformation as: a transformation where the application of a rule is scoped over the context where the rule is defined rather than the context where the rule is being applied. 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 transformation 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 retreive 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 only be performed by implementing an external program which can then be invoked at the time of translation.

From an end user perspective, we want to be able to perform transformation based on contextual information. ANTLR's approach of using scopes to perform context-sensitive transformation is rather simple (and therefore easy to implement) whereas Stratego's approach of using dynamic rewrite rules, although is quite powerful and has many applications [3], requires a much in-depth knowledge of the Stratego language.

Error reporting We identify and present a broad classification of errors that are applicable when implementing DSLs in Table 4. 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.

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

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 4
A comparison of the error reporting capabilities of ANTLR, Ruby, Stratego, and Converge

of the compilation of the target program. Run-time errors are reported at the time of execution of the target program. Although the run-time 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 commment). 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. Run-time errors are reported 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.

From an end user perspective, we want the DSL tool to report errors in terms of the source DSL. In ANTLR and Stratego, errors related to transformation are reported only at the time of the compilation of the end language, thus leading to increased development time and cost. In Ruby and Converge, the ability to report an error with a complete stack trace results in much quicker implementation.

8.2 Metrics

In this section, we present our comparative analysis of the four DSL tools based on the metrics listed in Table 2. The numerical data for these metrics (derived from DSL implementation in sections 4, 5, 6 and 7) are shown in Table 5, based on which we present our analysis. We also provide our views from an end user perspective on how the DSL tools compare for each of these metrics.

Metric	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 5
A comparative analysis of ANTLR, Ruby, Stratego, and Converge based on metrics

Lines of code 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.

From an end user perpective, we are only interested in the aspect—number of lines of code of the DSL program. This is because the other two aspects will be implemented only once during the lifecyle of a DSL whereas programs in a DSL will be implemented potentially many times over. In general, DSL programs implemented using a pure embedding approach (Ruby) will always lack the expressive power (and therefore less succinct) as compared to the tools that support customisable syntax (ANTLR, Stratego, and Converge).

Aspects to learn 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 'piepline framework' aspect to understand how the different stages of the transformation pipeline work together. In Converge, to implement DSLs, two aspects needs to be learned: the

'grammar' aspect for defining the lexer and parser rules; and the 'compile-time metaprogramming' facility to perform translation.

From an end user perspective, we want to learn as few aspects as possible in implementing DSLs and we want the implementation to be relatively simple (in terms of technical complexity). Ruby's approach scores on both accounts. Both ANTLR and Converge score the same in terms of number of aspects that needs to be learned but implementation is quicker in Converge as it has better error reporting facilities. Implementing DSLs in Stratego is relatively complex as it requires understanding of many different components for the various stages of the pipeline.

9 Discussion

ANTLR uses a stand-alone approach to implement DSLs. ANTLR comes with ANTLRWorks [11], 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) which then allows the grammar to be reused for generating target programs in 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. 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.

Based on our case study, 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 where a DSL will be applied many times over rather than a quick one-off use.

In terms of the overall complexity of a tool in implementing DSLs, Ruby and Stratego are on the opposite ends of the spectrum with ANTLR and Converge somewhere between the two. Ruby requires only one aspect to be learned and therefore DSL implementation is simple. Stratego's approach however is relatively complex because the DSLs are implemented using a pipeline approach (Figure 3) with each stage requiring an understanding of its various components. Therefore, implementation costs are likely to be higher in Stratego compared to the other approaches. Based on the experience in implementing our case study, we conclude that implementing DSLs in Ruby is easy but under-powered; in Stratego the implementation is difficult but highly flexible; and the implementation in Converge is somewhere in the middle.

Our study also highlighted that accurate sources of documentation with sufficient examples are essential to effective implementation of DSLs. Ruby being an open source GPL, is 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. ANTLR and Converge come with examples on how to implement DSLs that can be used as a reference.

10 Conclusions

In this paper, we implemented DSLs using a stand-alone approach and three different embedded approaches. The stand-alone 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] Bravenboer, M., A. V. Dam, K. Olmos, E. Visser, *Program Transformation with Scoped Dynamic Rewrite Rules*, Technical Report, Utrecht University, 2005.

[4] 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.

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

[6] Dmitriev, S., "Language Oriented Programming: The Next Programming Paradigm," Technical report, JetBrains, 2004.

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

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

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

[10] Levine, J., T. Mason, and D. Brown, "Lex & Yacc," 2nd Ed., O'Reilly Media, Inc., 1992.

[11] Parr, Terence, "The Definitive ANTLR Reference: Building Domain-Specific Languages," The Pragmatic Bookshelf, 2007.

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

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

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

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

[16] Documentation on Intentional Domain Workbench, URL: http://blog.intentsoft.com/intentional_software/2009/05/dsl-devcon.html