A Little Language for Surveys: Constructing an Internal DSL in Ruby

H. Conrad Cunningham
Department of Computer and Information Science
University of Mississippi
University, MS 38677 USA
cunningham@cs.olemiss.edu

ABSTRACT

Using a problem domain motivated by Bentley's "Little Languages" column [1], this paper explores the use of the Ruby programming language's flexible syntax, dynamic nature, and reflexive metaprogramming facilities to implement an internal domain-specific language (DSL) for surveys.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

General Terms

Design, Languages

Keywords

Domain specific language, Ruby, reflexive metaprogramming

1. INTRODUCTION

Hudak defines a *domain-specific language* (or DSL) as "a programming language tailored to a particular application domain" [10]. DSLs are usually not general-purpose languages; they instead "trade generality for expressiveness in a limited domain" [11]. Thus DSLs must be precise in capturing the semantics of their application areas [10]. They are usually small, declarative languages targeted at end users or domain specialists who are not expert programmers [10, 16].

DSLs, often known as *little languages*, have long been important in the Unix operating systems community. For example, in an influential 1986 column [1], Bentley describes the little line-drawing language `pic` and its preprocessors `scatter` (a language for drawing scatter plots of two-dimensional data) and `chem` (a language for drawing molecular structures). He also describes other well-known little languages that are used to implement `pic`: `lex` for specifying lexical analyzers, `yacc` for specifying parsers, and `make` for specifying build processes.

ACM SE '08, March 28-29, 2008, Auburn, Alabama, USA

Fowler classifies DSLs into two styles—external and internal [6]. An *external DSL* is a language that is different from the main programming language for an application, but that is interpreted by or translated into a program in the main language. The little languages from the Unix platform are in this category. The document preparation languages LaTeX and BibTeX, which the author is using to format this paper, are also external DSLs. External DSLs may use ad hoc techniques, such as hand-coded delimiter-directed or recursive descent parsers [6], or may use parser-generation tools such as `lex` and `yacc` or ANTLR [12].

What Fowler calls an *internal DSL* (and Hudak calls a domain-specific *embedded* language [10]) transforms the main programming language itself into the DSL. This is not a new idea; usage of syntactic macros has been a part of the Lisp tradition for several decades. However, the features of a few contemporary languages offer new opportunities for constructing internal DSLs.

The rise in popularity of the Ruby programming language [14] and the associated Ruby on Rails web framework [15] has simulated new interest in DSLs among practitioners. In the Ruby environment, there is significant interest in developing internal DSLs [2, 8] that are made possible by the extensive reflexive metaprogramming facilities of Ruby [3]. One interesting language of this nature is `rake`, a build language implemented as a DSL in Ruby [5].

This paper takes a problem motivated by Bentley's "Little Languages" column [1], constructing a little language for surveys, explores the DSL capabilities of the Ruby language, and designs an internal DSL for specifying and executing surveys. Section 2 describes the Ruby facilities for constructing internal DSLs. Section 3 analyzes the survey problem domain and designs a simple DSL based on the analysis. Section 4 sketches the design and implementation of the survey DSL processor. Sections 5 and 6 examine this work from a broader perspective and conclude the paper.

2. RUBY'S INTERNAL DSL SUPPORT

Ruby is an interpreted, dynamically typed, object-oriented application programming language with extensive metaprogramming facilities [3, 14]. Its features are convenient for both defining and implementing DSLs. Although those tasks are interrelated, it is useful to examine them separately.

Ruby has two groups of features that especially support *defining* internal DSLs [2, 8]—its flexible syntax and its support for blocks (i.e., closures).

A flexible syntax is important for making DSLs read in a natural way. Of special importance is the syntax for method

calls because methods are the primary means for adding operators and declarators—the verbs—to an internal DSL. Ruby method calls are flexible in three key ways: the parentheses enclosing argument lists are optional, methods may have a variable number of arguments, and the use of hash data structures in argument lists provides a mechanism similar to keyword arguments. The survey DSL exploits the first two of these features. For example, the Ruby code

```
question "Male or female?"
```

calls method `question` with one argument, a string literal giving the text for a survey question. The method `question` is defined with a second, optional parameter giving the expected number of responses if greater than 1. If the optional argument is given in a call, then it is separated from the first argument by a comma.

Of course, a DSL must also identify the entities to be operated upon—the nouns—and their attributes—the adjectives. In some circumstances, the nouns may be identifiers for host language variables or classes, but, in other circumstances, quoted string literals may need to be introduced. Quoted strings, with their "noisy" pairs of quotation marks, tend to make the DSL more difficult to read and write. Ruby provides a less "noisy" alternative, the *symbol*. A symbol is an identifier-like textual constant that is preceded by a colon. For example, a programmer might combine the use of symbols with "keyword parameters" in a call such as

```
question :text => "Male or female?", :nresp => 1
```

where the two "arguments" are collected as mappings in a hash data structure passed as an argument to the method.

Any Ruby method call may have a block attached. A block (also called a Ruby `Proc` or a closure) is group of executable statements defined in the environment of the caller and passed unevaluated as an implicit argument to the called method. The called method may then execute the block zero or more times, supplying any needed arguments for each call. The block executes with access to the environment in which it was defined. As an example, consider the Ruby code

```
response "Female" { @female = true }
```

that associates a parameterless block with the one-argument method call `response`. When executed, the block sets instance variable `@female` to the constant `true`. Blocks can either be enclosed in a pair of braces, as above, or in a `do-end` construct, which is better for multi-line blocks.

The block feature is useful for DSL construction in at least two ways [2]. First, a block provides a structuring mechanism for DSL statements. The execution of the Ruby (or DSL) statements inside the block is controlled by the called method. Second, a block enables deferred evaluation. A block can be stored in a data structure or passed on as an argument to other methods. The first way is useful in defining DSLs. Both are useful in implementing DSLs.

Ruby's extensive reflexive metaprogramming facilities are especially important for *implementing* internal DSLs [2, 8]. *Metaprogramming* is the capability of a program to manipulate programs as data. *Reflexive metaprogramming* is the capability of a program to manipulate its own program structures [17]. Reflexive metaprogramming is possible primarily because Ruby is an interpreted language whose interpreter provides programmers with hooks into its internal state. Ruby's facilities include the ability to query an object to determine its methods, instance variables, and class—features that are available in mainstream languages such as Java—and also more exotic facilities such as the ability to evaluate strings as code, to intercept calls to undefined methods, to define new classes and methods dynamically, and to react to changes in classes and methods via *callback* methods. Such features have long been a staple of languages such as Lisp and Smalltalk, but the recent interest in Ruby has helped renew interest in metaprogramming.

The implementation of the survey DSL uses the following Ruby reflexive metaprogramming facilities [8, 14]:

`obj.instance_eval(str)` takes a string `str` and executes it as Ruby code in the context of `obj`. This method allows internal DSL code from a string or file to be executed by the Ruby interpreter.

`mod.class_eval(str)` takes a string `str` and executes it as Ruby code in the context of module `mod`. This enables new methods and classes to be declared dynamically in the running program.

`obj.method_missing(sym,*args)` is invoked when there is an attempt to call an undefined method with the name `sym` and argument list `args` on the object `obj`. This enables the object to take appropriate remedial action.

`obj.send(sym,*args)` calls method `sym` on object `obj` with argument list `args`. In Ruby terminology, this *sends a message* to the object.

Now we examine how these language facilities can be used to define an internal DSL for a nontrivial domain.

3. LITTLE LANGUAGE FOR SURVEYS

The problem domain addressed here is similar to the one for Bentley's "little language for surveys" [1]. We first analyze the domain systematically and then use the results to design an appropriate DSL syntax and semantics.

The domain is a family of applications for administering simple surveys. We *analyze the domain* using *commonality/variability analysis* [4] and produce four outputs.

Scope: the boundaries of the domain—what must we address and what can we ignore.

Terminology: definitions for the specialized terms, or concepts, relevant to the domain.

Commonalities: the aspects of the family of applications that do not change from one instance to another.

Variabilities: the aspects of the family of applications that may change from one instance to another.

The *scope* focuses on the definition of a simple survey and its presentation to individuals in various forms. We do not address issues related to tabulation of the survey results.

Within this scope, we identify several specialized *terms*. These include *survey*, *title*, *question*, and *response* related to the survey's structure. To have a scope similar to Bentley's surveys [1]), we need the concepts of *conditional question*, which may be omitted under some conditions, and *silent*

question, whose result is calculated from previous responses. We also have the concept of *execution* of the survey and its presentation to a *respondent*.

The *commonalities* we identify include:

1. A survey has a *title*.
2. A survey consists of a *sequence of questions* that are to be presented to the respondents.
3. Each question has a *sequence of responses* that can be chosen by the respondent.
4. A *conditional question* may be omitted based on the responses to questions earlier in the sequence.
5. The response to a *silent question* is calculated based on the responses to questions earlier in the sequence.
6. *Execution* of the survey results in presentation of the appropriate questions and the possible responses to the *respondent* and the collection of his or her choices.

The *variabilities* we identify include the:

1. actual texts displayed for the title, questions, and responses
2. number and order of questions within the sequence
3. number of responses required or allowed for a question
4. number and order of responses within a question
5. condition under which a question may be omitted
6. method for calculating the results of a silent question
7. source of the survey specification
8. manner in which the questions are displayed and the responses collected during execution.

We use the analysis above to guide our choices for elements of the *DSL design* [11, 13]. The terminology and commonalities suggest the DSL statements and constructs. The commonalities also suggest the semantics of the constructs and the nature of the underlying computational model. The variabilities represent syntactic elements to which the survey programmer can assign values.

To avoid unnecessary language elements, we do not introduce a construct for the *survey* itself. Instead, we define a survey as the content of one DSL "file". The syntax of this file consists of one `title` statement (commonality 1) and a sequence of questions. The `title` statement has the syntax

```
title TEXT
```

where `title` is a keyword and *TEXT* is the user-supplied title text (variability 1). Syntactically, this is a call of the Ruby method `title` with one argument.

According to commonality 2 and variability 2, the body of the survey consists of a user-defined sequence of questions. We thus introduce a `question` statement to denote the basic survey question. It has the syntax

```
question TEXT, NUM_RESPONSES do
  QUEST_BODY
end
```

where the argument *TEXT* gives the text of the question (variability 1) and optional argument *NUM_RESPONSES* (variability 3) gives the number of responses expected, with a default value of 1. This construct defines a question in the survey at that point in the sequence (variability 2). Syntactically, this is a Ruby method call with one required argument, one optional argument, and an attached `do-end` block.

The *QUEST_BODY* must be structured according to commonality 3 and variabilities 1 and 4. That is, the block consists of a sequence of possible responses. However, sufficient data must be captured to enable commonalities 4 and 5 and variabilities 5 and 6 to be implemented. We thus structure the *QUEST_BODY* as a sequence consisting of an optional `condition` statement, some number of `response` statements, and an optional `action` statement. It has the syntax:

```
condition COND_BLOCK
response TEXT RESP_BLOCK
...
action ACTION_BLOCK
```

The statements above are Ruby method calls with blocks attached. When the system executes the survey, if a `condition` statement is present and its *COND_BLOCK* evaluates to false, the question is omitted (commonality 4). Otherwise, the system presents the `response` texts in the order given (commonality 3). When the respondent selects one or more of these (up to the number given on the `question` statement), the system executes the corresponding *RESP_BLOCK*s.

To respond to commonality 5 and variability 6 for silent questions, we introduce the statement `result` at the same syntactic level as `question`. However, the `result` body defines a sequence of `alternative` statements (commonality 3) that, when executed, will be selected based on the previous responses. The `result` has the following syntax, which also responds to variabilities 1 and 3 for questions:

```
result TEXT, NUM_RESPONSES do
  condition COND_BLOCK
  alternative TEXT GUARD_BLOCK
  ...
  action ACTION_BLOCK
end
```

The `condition` and `action` statements are the same as for *QUEST_BODY*. The `alternative` statements execute in the given sequence. If a *GUARD_BLOCK* is omitted or evaluates to true, then that alternative is chosen.

The blocks on the `action` and `response` statements consist of Ruby code that creates and modifies instance variables in the environment in which the survey is executed. These instance variables can then be used in the "boolean" blocks on the `condition` and `alternative` statements to allow commonalities 4 and 5 to be realized, in accordance with the flexibility needed for variabilities 5 and 6.

Commonality 6 defines the basic computational model for execution of a survey program; variabilities 7 and 8 define flexible aspects that must exist in the implementation. Now we look at the implementation of this internal DSL in Ruby.

4. INTERNAL DSL IMPLEMENTATION

The survey DSL could be implemented with a processor that executes each question-level statement immediately after it has been parsed. This *single-pass* approach would,

```
def question(txt,*args) # txt, ns, block
  if @context.level==:survey_level && block_given?
    @context.level = :question_level
    @context.qtype = :question_type
    ns = 1
    ns = args[0].to_i if args.size > 0
    @context.question=QuestionNode.new(txt.to_s,ns)
    yield   # execute block on DSL question call
    @context.survey.add_question(@context.question)
    @context.level    = :survey_level
    @context.qtype    = :no_type
  else
    # output appropriate error messages
  end
  @context.question = nil
end#question
```

Figure 1: Method `SurveyDSL#question`

however, strongly couple the evaluation logic with the parsing logic and make support for variabilities 7 and 8 difficult.

In most cases the use of a *two-pass* architecture is a better technique. The first pass reads the input, parses it, generates any needed error messages, and builds the corresponding *abstract syntax tree* (AST) [6, 12]. The AST is a tree-like data structure that represents the input expressions in an abstract form. The second pass takes the AST, presents the questions to the respondent in the required order, and collects the responses (commonality 6). This approach allows any first-pass processor to be configured with any second-pass processor, thus supporting variabilities 7 and 8. In this section, we look at the design and implementation of the AST, first-pass, and second-pass classes.

4.1 DSL Parsing

The Ruby implementation of the survey DSL uses several classes to implement the AST. At the top (survey) level, the class `SurveyRoot` represents the entire survey as specified in a DSL input file. It holds the survey title from the `title` statement and a sequence of question-level nodes.

At the second (question) level are the "abstract" class `QuestionLevelNode` and its two subclasses `QuestionNode` and `ResponseNode`. The subclasses represent the `question` and `result` statements in the survey DSL. They store the question text, the guarding condition (if any) from the associated `condition` statement, the action (if any) from the associated `action` statement, and a sequence of "responses" from the associated `response` or `alternative` statements.

The third (response) level consists of the "abstract" class `ResponseLevelNode` and its two subclasses `ResponseNode` and `AlternativeNode`. The subclasses represent the DSL's `response` and `alternative` statements.

The *first-pass parser classes* are structured according to Fowler's *Object Scoping* DSL pattern [6] using an approach Buck calls *sandboxing* [2]. The "abstract" class `SurveyDSL` implements the DSL statements as methods. Its subclass `SurveyBuilder` "evaluates" the DSL statements from a DSL input file using its superclass's methods. This evaluation parses the DSL input and builds the AST using the node classes described above. Figure 1 shows the `question` method.

The `read_DSL` method of class `SurveyBuilder` reads the DSL input from a file and evaluates it by calling the method `instance_eval` described in Section 2. The safety of the program is maintained by encapsulating the relatively unsafe

```
def action(&action)
  if @context.level==:question_level && block_given?
      && @context.question.action == nil
    @context.question.action = action
  else
    # output appropriate error messages
  end
end#action
```

Figure 2: Method `SurveyDSL#action`

`instance_eval` method call within the "sandbox" provided by a `SurveyBuilder` object.

In this design, the `SurveyDSL` class uses the *Memento* design pattern [9]. It uses an object of class `DSLContext` to store the state of the DSL parser (i.e., `SurveyDSL` object) so that it can be saved and restored. This is also what Fowler calls the *Context Variable* DSL pattern [6].

The first-pass design separates the DSL implementation methods and the parser state from the class that executes the DSL input. This, along with the filename of the DSL source being a parameter of the `read_DSL` method, provides the flexibility needed for variability 7.

Because the survey DSL input is just Ruby code, much of the work of parsing the DSL is done by the Ruby interpreter. The DSL parser must verify that the DSL program is syntactically correct and generate the corresponding AST. In addition to what DSL method has been called in `SurveyDSL`, the parser state can be characterized by four primary attributes: (1) `survey`, which is a reference to the partially constructed AST, (4) `question`, which is a reference to the `QuestionLevelNode` object currently being constructed, (3) `qtype`, which gives the current type of question-level statement (none, `question`, or `result`) being parsed, and (2) `level`, which identifies the current level of the DSL syntax (survey, question, or response) being parsed.

Suppose the following `question` statement appears in the DSL input:

```
question "What is your gender?" do
  response "Female" { @female = true }
  response "Male" { @female = false }
  action { @male = if @female then false
                              else true end }
end
```

The `read_DSL` method of class `SurveyBuilder` reads this text and evaluates it as Ruby code by calling `instance_eval`. This causes the `question` method in class `SurveyDSL` (Figure 1) to be called with a string argument and an attached block. The "optional" second argument (i.e., number of responses to be given) is set to the default value of 1.

If the parser is in the proper state, the `question` method processes the DSL statement to create a new `question` node. It changes the parser `level` to question-level and the parser `qtype` to question-type and creates a new `QuestionNode` to put in the AST. It then invokes the attached `do-end` block using the Ruby `yield` statement. When control returns from the `yield`, the `question` method stores the new node in the AST, resets the parser `level` to survey-level, and returns control to the executing `instance_eval` call.

The execution of the block attached to the `question` statement invokes the `response` method in `SurveyDSL` twice and the `action` method once. The `response` method calls create the two response-level nodes in the AST. The `action`

```
def accept(survey_visitor)
  @env.survey_title   = @title # used by DSL block
  @env.survey_answers = []     # used by DSL block
  @env.question_num   = 1      # used by DSL block
  survey_visitor.execute_title(@env,self)
  questions.each do |q|
    q.accept(@env,survey_visitor)
  end
end#accept
```

Figure 3: Method `SurveyRoot#accept`

```
def accept(env,survey_visitor)
  env.question_text       = @text # used by DSL
  env.question_num_to_sel = @num_to_sel  #  block
  survey_visitor.execute_question(env,self)
  env.question_text       = nil
  env.question_num_to_sel = nil
end#accept
```

Figure 4: Method `QuestionNode#accept`

method call sets the `QuestionNode`'s action attribute to the given block. Figure 2 shows the code for the `action` method.

The blocks attached to the `response` and `action` statements are not executed. Instead, the blocks themselves (i.e., the closures) are stored in the AST node for execution in the second pass. If the `question` statement included a `condition` statement, it would be processed similarly to the `action` statement. This technique of storing blocks uses what Fowler calls the *Deferred Evaluation* DSL pattern [6].

4.2 DSL Interpretation

The interactions between the *second pass* and AST must support variability 8, enabling different "interpreters" in the second-pass to be configured into the system. Because the little survey language has a straightforward syntax and semantics, the prototype design uses the *Visitor* design pattern [9] to structure this interaction. A complex language may require more sophisticated tree-walking logic [12].

To implement the Visitor pattern, each AST class must provide a method `accept` that takes a `SurveyVisitor` object. This method must call the appropriate visit operations on the Visitor object and pass the object to next lower level of the AST as needed. Figures 3 and 4 show the `accept` methods of the top-level AST node `SurveyRoot` and second-level AST node `QuestionNode`, respectively.

The Visitor class must extend the "abstract" superclass `SurveyVisitor` and override methods `execute_question` and `execute_result` and, if the default behavior is not appropriate, override method `execute_title`. The `execute_*` methods represent the Visitor pattern's visit operations for the various AST nodes. The prototype provides the concrete Visitor class `SurveyInteractiveText` that implements an interactive, textual user interface using the standard input and output streams. Figure 5 shows the visit operation `execute_question` from that class.

The second pass starts execution by calling the `accept` method of the AST's root, passing a `SurveyInteractiveText` instance. In Figures 3 and 4, we see that this method calls the `exectute_title` operation and then calls the `accept` operations for each of the question-level nodes, which, in turn, call the `execute_question` operation. In Figure 5, we see the implementation of the desired survey question semantics. First, the method checks whether the associated condition is

```
def execute_question(env,q)
  if q.condition == nil || q.condition.call
    display_question(env.question_num,q.text)
    resp  = {}; label = 'a'  # labels from 'a'
    q.responses.each do |r|
      display_response(label,r.text)
      resp[label] = [r.action,r.text]
      label       = label.succ
    end
    answers = get_answers(q.num_to_sel,'a'...label)
    env.survey_answers << [env.question_num,answers]
    answers.each do |a| # evaluate selected actions
      env.response_label = a          # used by
      env.response_text  = resp[a][1] #  DSL block
      act = resp[a][0]
      act.call unless act == nil
      env.response_label = nil
      env.response_text  = nil
    end
    q.action.call unless q.action == nil # eval
  else
    env.survey_answers << [env.question_num,[]]
  end
  env.question_num += 1
end#execute_question
```

Figure 5: `SurveyInteractiveText#execute_question`

satisfied. If it is not satisfied, then the question is skipped. If it is satisfied, then the method displays the question text and the possible responses and gathers the selections from the respondent.

A key aspect of the `execute_question` call is the evaluation of the blocks (i.e., Ruby `Procs` or closures) stored in the `QuestionNode` for the conditions and actions. These are groups of Ruby statements whose executions have been deferred from the first to the second pass. The blocks are invoked using the `Proc`'s `call` method. These stored blocks are parameterless, but they can create new instance variables in the `SurveyBuilder` object in which they are defined.

The Visitor's `execute_*` methods must also make the question number, response label, and the previous responses available to the executing condition and action blocks. The second-pass code uses the `missing_method` callback in the `SurveyBuilder` class to dynamically create the needed writer and reader methods. This method traps calls to undefined writer methods and uses Ruby's `class_eval` facility to create what is needed. It then uses Ruby's `send` method to re-dispatch the writer call to the new method.

Of course, class `SurveyInteractiveText` is only one possible Visitor class. Others could be implemented to provide a GUI user interface or to print a listing of the survey. Thus the design of the second pass supports variability 8.

5. DISCUSSION

Drawing on the experience in designing and evolving the JMock DSL, Freeman and Pryce make four recommendations for constructing a DSL in Java [7]. Although Ruby provides better support for DSL construction than Java, these ideas are still relevant to the Survey DSL.

Freeman and Pryce's first recommendation is to "separate syntax and interpretation into layers" [7]. The survey DSL work finds this approach to be beneficial. The first pass uti-

lizes various unusual Ruby features (e.g., reflexive metaprogramming) to express and process the little language for surveys. Except for the execution of the closures, the second pass is more conventional, using the Visitor pattern to execute the survey stored in the AST. The first pass implementation is relatively complex; the second pass more straightforward. As with JMock, the oddball aspects are separated from the more routine aspects.

The second recommendation is to "use, and abuse, the host language" [7] to enable the writing of readable DSL programs. The survey DSL uses the flexible syntax of Ruby—optional parentheses in method calls, variable-length parameter lists, and blocks—to express the survey programs in a readable, mostly declarative syntax. The DSL is defined so that its execution as Ruby code gives a shell of a recursive descent parser for the language. The DSL parser then leverages the dynamic, reflexive metaprogramming features of Ruby to recognize the syntax. Similarly, storing closures for execution in the second pass makes the implementation of the interpreter challenging. Ruby internal DSLs use the distinctive features of Ruby and perhaps flirt a bit with danger by using the reflexive metaprogramming facilities.

Freeman and Pryce's third recommendation is "don't trap the user" in the internal DSL [7]. The survey DSL implementation addresses this issue in a preliminary way. The use of the Visitor pattern in the second pass enables users to write visitors for other purposes. In the first pass, new subclasses can also be implemented for `SurveyDSL` or `SurveyBuilder` to extend the DSL. However, because the prototype is a relatively course-grained whitebox framework, a programmer wishing to extend these classes must have considerable knowledge of the current implementation. The tight coupling of the parser classes with the concrete `DSLContext` class may also cause some complications. Clearly, enabling users to extend the survey DSL is an area for future work.

The fourth recommendation is to "map error reports to the syntax layer" [7] rather than to the interpretation layer, which is invisible to the DSL user. Although this paper does not describe the error-reporting facilities, the survey DSL parser does report error messages linked closely to the input statements. Giving specific error messages is not trivial given that the Ruby interpreter does much of the parsing. The current design does not provide error messages during execution that tie back to the syntactic components. However, the straightforward mapping from DSL statements to AST nodes should enable such an approach. This, too, is an area for future work.

6. CONCLUSION

This paper relates some of the author's experiences in the systematic analysis of the survey domain and the use of the analysis to design and implement a novel domain-specific language (DSL) for expressing survey programs. The language is designed as an internal DSL in Ruby and implemented using Ruby's distinctive metaprogramming features. Ruby and the analysis, design, and implementation techniques employed prove to be effective in this instance. However, further research is needed to delineate more rigorous techniques for analyzing the domain and using the analysis results to motivate a Ruby internal DSL design. In addition, future work should seek to formulate guidelines for achieving effective and safe Ruby implementations of the resulting DSL designs.

7. ACKNOWLEDGMENTS

The author thanks Chuck Jenkins, Yi Liu, Pallavi Tadepalli, and Jian Weng for their helpful suggestions.

8. REFERENCES

[1] J. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[2] J. Buck. Writing domain specific languages. `http://weblog.jamisbuck.org`, April 2006.

[3] L. Carlson and L. Richardson. *Ruby Cookbook*. O'Reilly, 2006.

[4] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November 1998.

[5] M. Fowler. Using the rake build language. `http://martinfowler.com/articles/rake.html`, August 2005.

[6] M. Fowler. Domain specific languages. `http://martinfowler.com/dslwip/`, Work in progress 2007.

[7] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in Java. In *Companion to the Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 855–865. ACM SIGPLAN, October 2006.

[8] J. Freeze. Creating DSLs with Ruby. *Artima Developer: Ruby Code and Style*, March 2006. `http://www.artima.com/rubycs/articles/ruby_as_dsl.html`.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceeding of the 5th International Conference on Software Reuse (ICSR'98)*, pages 134–142. IEEE, 1998.

[11] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[12] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[13] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to implementation—Application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May/June 1999.

[14] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, second edition, 2005.

[15] D. Thomas and D. Heinemeier Hansson. *Agile Development with Rails*. Pragmatic Bookshelf, second edition, 2006.

[16] A. van Deursen, P. Klint, and J. Visser. Domain specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, June 2000.

[17] Wikipedia, The Free Encyclopedia. Metaprogramming. `http://en.wikipedia.org/wiki/Metaprogramming`, Accessed 22 February 2008.