Building Domain-Specific Languages for Model-Driven Development

Jesús Sánchez Cuadrado and Jesús García Molina, *University of Murcia*

Embedding a domain-specific language in a dynamic language is an alternative to constructing a compiler or interpreter. It can improve program readability and development time.

Today, the popularity of dynamic languages such as Python and Ruby is growing beyond their use as scripting languages. In fact, Sun, Microsoft, and other companies are supporting some dynamic languages on their development platforms. Developers increasingly find that dynamic languages' features help them enhance their productivity, while common misconceptions about them, such as poor performance and reliability, are disappearing.

Our experience with dynamic languages comes from Smalltalk, but in the last two years we've gained further experience by using Ruby to develop a tool for model-driven development (MDD).[1] The tool is based on the concept of a *domain-specific language*, which is a language specifically designed to perform a task in a certain domain. In an embedded DSL, the designer extends the host language's constructs with domain-specific constructs instead of building a parser. Here, we demonstrate that the features of dynamic languages make them appropriate to serve as host languages.

Our tool lets users deal with basic MDD tasks by providing several embedded DSLs. Here we present four of them:

- A DSL to create metamodels lets us specify a modeling language's abstract syntax;.
- A DSL to validate models provides a syntax like that of the Object Constraint Language to specify restrictions on models.
- A model-to-model transformation language lets us specify declarative mappings between constructs of modeling languages.
- A model-to-code language lets us generate code (that is, text) from a source model.

Because we intend our tool mainly for experimentation, we decided to use embedded DSLs to achieve an adaptable implementation in a short development time.

We use the DSLs we've created to illustrate techniques and features commonly found in dynamic languages such as metaprogramming, dynamic scoping, closures or code blocks, introspection, dynamic evaluation, and so on. These techniques and features have enabled us to write highly adaptable code. In some cases, our approach's runtime performance compares well with that of Java-based alternatives.

0740-7459/07/$25.00 © 2007 IEEE

```ruby
class Relational < MetamodelKeywords
  metaclass 'Table' do
    attribute 'name', String
    reference 'cols[*]', 'Column', 'owner'
  end

  metaclass 'Column' do
    attribute 'name', String
    attribute 'type', String
    attribute 'primary', Boolean
    reference 'owner', 'Table', 'columns'
  end

  metaclass 'ForeignKey' do
    reference 'referencedTable', 'Table'
    reference 'cols[*]', 'Column'
  end
end
```

(a)

```ruby
class MetamodelKeywords
  def self.metaclass(name)
    @classes << Ecore::EClass.new(name)
    yield
  end

  def self.attribute(name, type)
    att = Ecore::EAttribute.new
    att.name = name
    ...
    @classes.last.attributes << att
  end

  def self.reference(name, type,
                     opposite = nil)
    ...
  end
  ...
end
```

(b)

Figure 1. (a) Using an embedded DSL to specify a relational metamodel; (b) using part of a DSL implementation to define metamodels.

A simple DSL to define metamodels

Metamodeling is a key aspect of MDD. A *metamodel* defines the abstract syntax of a modeling language or DSL. A metamodeling language such as MOF (Meta Object Facility) or Ecore lets us define a metamodel using object-oriented constructs such as classes, associations, and generalization.

A graphical editor is the most convenient way to define a metamodel, but developing one is time consuming. Since a metamodel language can be considered a DSL for defining metamodels, we decided to implement one for this task; however, instead of hand-crafting a parser, we implemented it as an embedded DSL. Embedding a DSL into a language is a well-known technique in dynamic languages. It has its foundations in the nonintrusive syntax of these languages and their ability to evaluate expressions as they appear in the program text.[2,3] A DSL embedded in a Lisp-like language is based on a macro system—that is, it expands the macros before evaluating the program text. In contrast, the DSL in message-based OO languages such as Ruby and Smalltalk is defined by means of methods, which are evaluated at runtime.

So, to create an embedded DSL in Ruby, we define each DSL keyword as a method, so that the language's interpreter executes a keyword as it reads the corresponding method call. We can use the body of a class as the context for keyword methods and use inheritance to set such methods in context. (We could define DSL keywords as instance methods, but we'd need to evaluate the DSL program text explicitly in the context of an object using the `instance_eval` method.)

Figure 1a shows the definition of a relational metamodel, while figure 1b shows an excerpt of the DSL implementation to define metamodels. We define the metamodel in a class called `Relational` inheriting from `MetamodelKeywords`, the class that actually implements the DSL by defining class methods. The keywords `metaclass`, `attribute`, and `reference` correspond to the class methods of the same names defined in the class `MetamodelKeywords`. (Class methods are defined with `def self.method_name`). Ruby's nonintrusive syntax lets us invoke these methods in the class body, even without using brackets to enclose the arguments, thus providing a clear style for writing the DSL.

Then we map each DSL nested structure to a code block. For instance, a nested structure is the containment relationship between classes and attributes, and a Ruby code block is a piece of code enclosed by `do...end` or by `{}`. The implementation of `metaclass` creates a new metaclass and calls `yield` to evaluate the passed block. The object that receives the message is implicit. Because we've mapped keywords to class methods, the receiving object is the `Relational` class itself.

This approach to implementing a DSL is much less time consuming than using a parser generator: there's no need to deal with either a grammar or an abstract syntax tree. On the other hand, defining an arbitrary syntax is impossible, because we're limited to the underly-

```
toprule 'klass2java' do
  from UML::Class
  to   Java::Class
  mapping do |klass, javaklass|
    javaklass.name     = klass.name
    javaklass.features = klass.fields
  end
end

rule 'field2feature' do
  from UML::Field
  to   Java::Attribute
  mapping do |field, attribute|
    attribute.name = field.name
    attribute.type = field.type

    case field.visibility
    when UML::Vk_private
      attribute.visibility = 'private'
    when UML::Vk_public
      attribute.visibility = 'public'
    else
      raise "Not handled"
    end
  end
end
```

Figure 2. An excerpt of a UML-to-Java transformation. Fields of a UML class are transformed into attributes of a Java class.

ing host language's syntax.[4] For instance, Ruby syntax is best suited to define block-structured DSLs, and Lisp syntax is best suited for defining expression-based DSLs.

A DSL for model transformation

Model transformation is at the heart of MDD because it bridges the gap between domain concepts and implementation technologies.[5] The root of our research into embedded DSLs came from a project we started to define RubyTL, a rule-based model transformation language with the distinctive feature of extensibility.[6] In addition, the language adds a declarative layer on top of Ruby to transform models by means of declarative rules that specify mappings between model elements.

Figure 2 shows an excerpt of a transformation definition between a UML class model and a Java class model. The first rule transforms every UML class into a Java class (as the `from` and `to` keywords specify). The mapping keyword takes a code block in which we've placed special assignments named *bindings* (to specify what is transformed into what). We implement such bindings by redefining set accessor methods to look up a rule that can transform the right part into the left part. For instance, the `field2feature` rule resolves the second binding of the `klass2java` rule.

RubyTL's declarative style provides a clean way of writing transformation definitions. Nevertheless, it might not be enough to express certain complex transformations where imperative constructs are needed. For these cases, because each rule's mapping part is actually a code block, we can write any imperative Ruby code we need. So, an important advantage of using an embedded DSL approach is to have all the power of a general-purpose language for free. For instance, in the `field2feature` rule, we use the `case` Ruby statement to set the visibility attribute, and we even reuse the Ruby exception mechanism (note the use of `raise`).

As we've said, extensibility is a key characteristic of RubyTL. It provides extension points for adding new features to its set of core features, thereby letting us test whether a new feature is useful for solving transformation problems. One of these extension points lets us define new kinds of rules, extending the default rule's behavior. We use this extension point to illustrate how we've applied three common techniques used in dynamic languages:

- *metaprogramming*—the ability to write a program that writes another program,
- *introspection*—the ability to obtain information about an object or class at runtime, and
- *intercession*—a program's ability to both inspect and modify its values at runtime.

Implementing a new kind of rule such as `toprule` (see figure 2) implies both inheriting from the `Rule` class (which implements the default rule's behavior) and defining a new keyword whose implementation only instantiates a new rule object of the proper type. So, the keyword implementation varies only in the `Rule` subclass's name, and we'd like to generate it automatically. To accomplish this, we need a way of knowing when the `Rule` class is inherited and to dynamically add a new keyword (that is, a new method) to the language. We solve the first requirement by using introspection and the second by intercession, which requires a form of metaprogramming.

Figure 3 shows how we implemented it. In Ruby, every class has a set of life-cycle callbacks that notify us when a method is added or removed, a class is inherited, and so on. In particular, when a class is inherited, Ruby triggers the `inherited` class method. We implemented this method in the `Rule` class so that the system

would notify us when the user defines a new kind of rule by inheriting from such a class.

Our implementation of the `inherited` method dynamically adds a custom method to the list of DSL keywords maintained in the `RubyTLKeywords` class. The interpreter invokes the `inherited` method when it evaluates the line "`class TopRule < Rule`." Then, the implementation of such a method dynamically generates the `toprule` method, which will be in charge of instantiating the new kind of rule.

Facilities to define string literals and to manipulate strings are common in dynamic languages, and they enable practical code generation. The code to create the new code is defined as a string, enclosed in `%{...}` (to handle multiline strings easily). The value of any expression enclosed in `#{...}` is substituted into the string.

The call `RubyTLKeywords.class_eval (code)` causes the interpreter to evaluate the code in the context of the class that actually implements the DSL, `RubyTLKeywords`, thus adding a new class method to the class. The white box in figure 3 shows the actual code executed when the DSL is evaluated and the interpreter executes the `toprule` method.

This example shows that such features of dynamic languages are very useful for writing adaptable, extensible code because they enable us to handle code at runtime. In this case, we used a powerful type of introspection to find out how the class hierarchy is built, and metaprogramming to avoid writing repetitive code and to easily modify the DSL at runtime.

```
class Rule
  def self.inherited(subclass)
    code = %{
      def self.#{subclass.name.downcase}(name, &block)
        @rules << #{subclass.name}.new(block)
      end
    }
    RubyTLKeywords.class_eval(code)
  end

  def self.keyword(name = nil)
    @name = name || @name
  end
end

class TopRule < Rule
  ...
end
```

Actual code for `toprule`:

```
def self.toprule(name, &block)
  @rules << TopRule.new(block)
end
```

Figure 3. Implementation of the Rule class automatically generates a DSL keyword when the user defines a new kind of rule.

A validation DSL

We also had to check whether the input models for model transformation and code generation were complete from the transformation point of view. This task is usually performed by writing constraints in OCL. But because we had developed the whole environment in Ruby, we decided to neither build an OCL interpreter nor reuse an interpreter in another language (Java, for instance). Instead, we created another DSL embedded in Ruby to build a constraint language.

Figure 4 shows an invariant on the `Class` metaclass of the UML metamodel ("for each class, if there exists an attribute called id, its type must be integer") written in both OCL and our constraint language. For each instance of the class specified by the context clause, the invariant is checked.

This DSL consists only of two keywords, `context` and `inv`, so implementing it is easy using the technique shown earlier to define language keywords with methods. Moreover, code blocks are practical for implementing iterators, which are similar to those in OCL (for example, `forAll`). However, for the DSL to work as expected, we must

- specify the invariant explicitly in a code block whose default execution context isn't the one at definition time, and
- implement the library of OCL predefined operations such as `forAll`, `implies`, and `oclKindOf` in the appropriate location.

In the implementation excerpt in figure 4c, the `context` method simply stores the metaclass from which instances will be taken to check the invariant. Such instances will be the context in which the `inv` code block should be evaluated. Some dynamic languages can change the environment in which a piece of code is evaluated. For this validation DSL, this feature is useful to change the scope in which the `inv` code block is evaluated. From the DSL user's point of view, the code block should be evaluated in the scope of an instance of the metaclass specified in the context clause, which differs from its default scope. We write the `self` variable in the code block, but such a variable isn't bound to any instance until the block is executed in the proper context. In the `inv` method, we use `instance_eval` to evaluate the block within the scope of each instance of the context. In this way, the instance properties written in the code block (for exam-

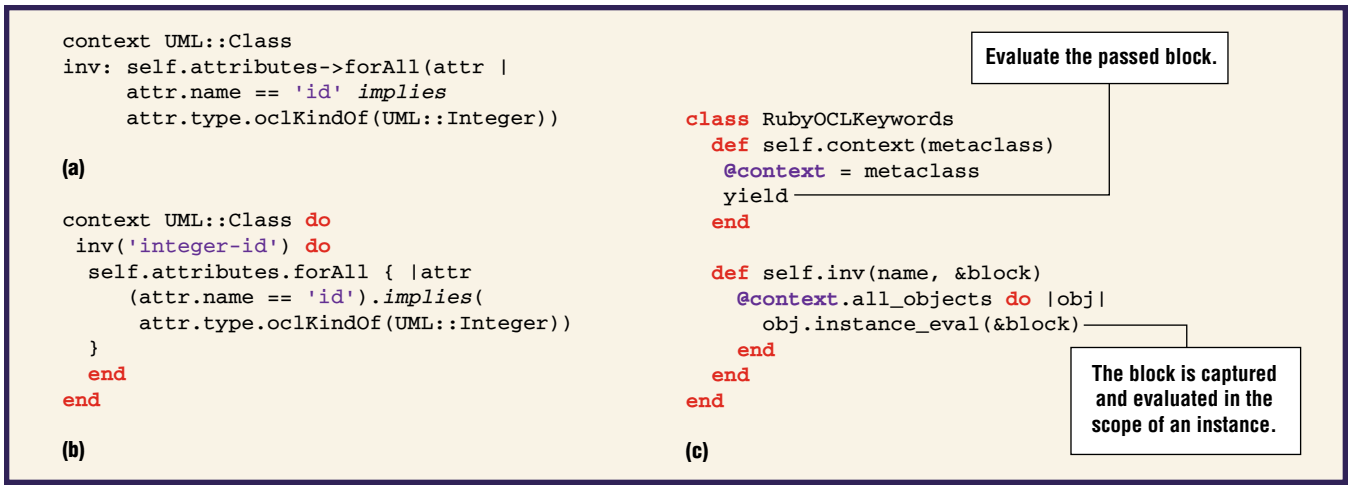

Figure 4. An invariant written in (a) the Object Constraint Language and (b) our constraint language; (c) an excerpt of the embedded DSL implementation.

ple, `self.attributes`) are dynamically bound to the corresponding object.

In Ruby, classes are open in the sense that you can always add or remove methods to an existing class, or even supersede an existing method. This applies even to the classes of the standard library. In our case, we override some classes to provide methods similar to those in the OCL library. For instance, we override the `Array` class to provide methods such as `forAll`, `exists`, and `includes`, which are similar to those in the OCL library. In addition, we added the `implies` method to `TrueClass` and `FalseClass` to implement the OCL keyword `implies`. This feature is more commonly found in a dynamic language where checking whether a method exists is delayed until that method executes. It provides a powerful way to reuse classes, although we must add one caution: breaking a class's contract by modifying an existing method is easy.

This DSL gave us an initial way to validate and navigate models syntactically close to OCL. It's also a good example of how a language's dynamic nature lets us evaluate code in a scope other than the default one.

A DSL for code generation

Usually a transformation chain's final step is a model-to-code transformation, also known as code generation. This step takes an input model and produces one or more text files, such as program files and deployment descriptors.

To provide this feature, we used the ERB template engine, available from the standard Ruby library, along with a small embedded DSL that provided commands to traverse the model, specify target files, and perform other tasks. We use this DSL to show another kind of metaprogramming that relies on defining classes at runtime.

Figure 5 shows an example of the DSL, which traverses all classes of a Java model and generates a `.java` file for each one. The `template` keyword applies a template, while the `indentation` keyword changes the indentation of the next template application. Finally, the `inline` keyword provides a way to output text without using a template file.

The problem involved in creating this DSL is how to make an object of the DSL (that is, a model element) available on a template. For instance, in the template box in figure 5, the *method* variable refers to the object stored in the DSL variable *m*, but such a variable isn't in the scope where the template will be executed. So, we need a way to give the template an execution environment where the names it needs are bound to the proper objects (note that templates have no parameters). In the DSL, the syntax to specify the binding between a name and an object is to pass a hash table containing name-object pairs (for instance, `:method => m`).

We can see in the Implementation box in figure 5 that the `template` method implementation consists of three steps:

1. We dynamically generate an anonymous class. For each name-object pair, we generate a method with the proper name and whose result is such an object. Later, we

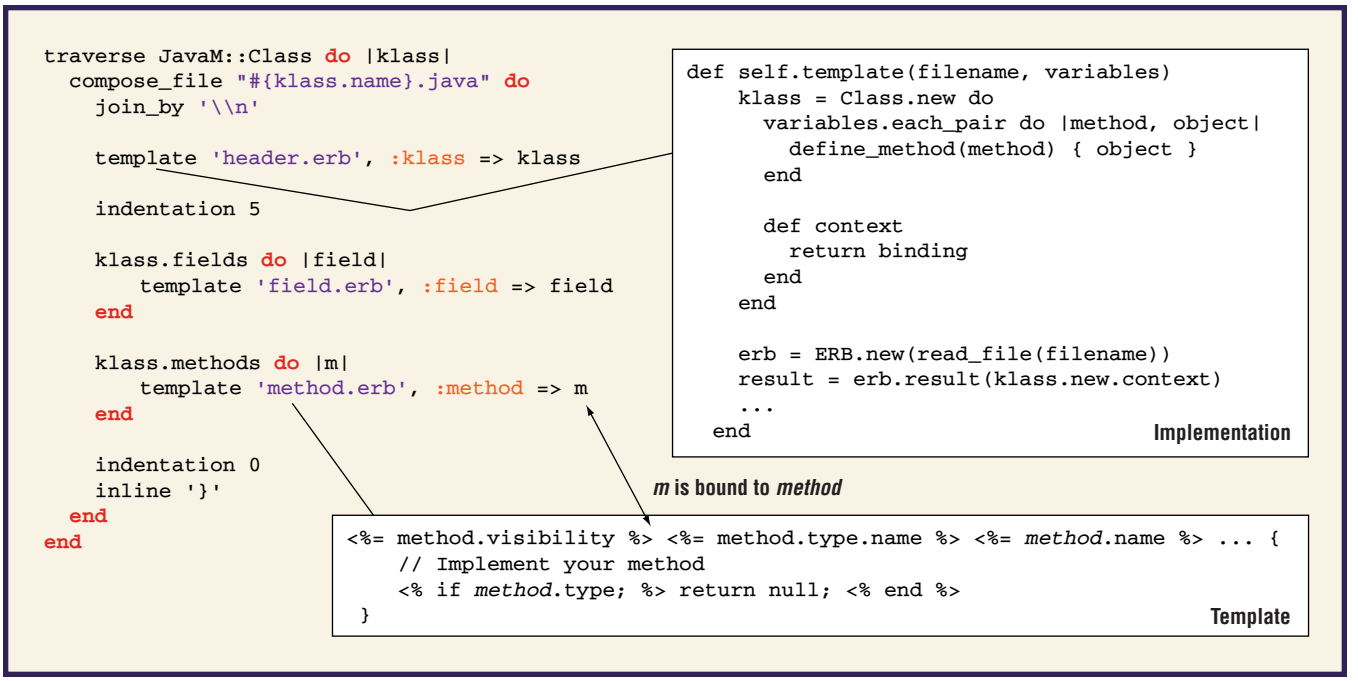

Figure 5. An example DSL for code generation, annotated with an excerpt of the implementation of the `template` keyword and with a template definition.

use an instance of such a class as the execution context for the template.
2. Every Ruby class has a private method called `binding` that returns an object of the `Binding` class encapsulating the object's execution context. Thus, we create a public method called `context` to return such an object for later use.
3. ERB lets us execute the template in a certain context (that is, the template's scope) using a `Binding` object. We use the binding of an instance of the `anonymous` class to execute the template in a context where the proper names are defined (note `klass.new.context`) So, we achieve our purpose: to make a DSL object available in a template.

Dynamic languages' ability to create data structures (classes, in this case) at runtime is useful for creating dynamic proxies easily and, in general, adapting code at runtime. This is one of dynamic languages' great strengths, because we can easily adapt components written by others to connect them to our own code.

Numerical test results

Dynamic languages' flexibility is supposed to shorten development time, but it might also decrease runtime performance. To establish to what extent this is true for our MDD tool, we ran model-to-model and model-to-code transformation tests, comparing some of our embedded DSLs against well-known open source languages written in Java that provide similar features. We also evaluated our model's parsing and handling facility (which lets us handle XML Metadata Interchange files as transformation inputs) against the standard Java solution, the Eclipse Modeling Framework (EMF).

Tables 1 and 2 show our results (our case studies are available for download at http://gts.inf.um.es/age). Table 1 deals with lines of code (without counting comments or automatically generated code); table 2, with Ruby's performance. We compared the ATL (Atlas Transformation Language) model-to-model transformation language against RubyTL, and the MOFScript model-to-code transformation language against our embedded DSL. The implementation of both ATL and MOFScript is based on a grammar that a parser generator translates into an abstract syntax tree, which is then interpreted or compiled. We also compared EMF (used in ATL and MOFScript) against our facility to parse and handle models.

The data show that we wrote many fewer LOC to implement our embedded DSLs and our parsing and model-handling facilities. Although we don't have an estimate of the development time of the Java tools, it's reasonable to as-

Table 1

Number of lines of code in model-to-model and model-to-code transformations using ATL, MOFScript, and our domain-specific languages, plus a comparison of the parsing and model-handling facility

	Model-to-model transformation				Model-to-code transformation				Parsing and model handling	
	ATL grammar	Ruby eDSL	ATL engine	Ruby engine	MOFScript grammar	Ruby eDSL	MOFScript engine	Ruby engine	Eclipse Modeling Framework	Ruby implementation
No. of lines of code	Not freely available	81	11,131	2,392	2,834	151	6,501	140	32,981	1,681

Table 2

Data resulting from model-to-model and model-to-code transformations using ATL, RubyTL, MOFScript, and our embedded DSL in Ruby

	Model-to-model transformation				Model-to-code transformation			
	UML → Java, using an average-size input model		UML → Java, using a big, autogenerated input model		Java → .java		SQL → .java	
	ATL	RubyTL	ATL	RubyTL	MOFScript	Ruby eDSL	MOFScript	Ruby eDSL
Input size (no. of model objects)	310	310	1,296	1,296	1,001	1,001	142	142
Output size	1,001 objects	1,001 objects	3,464 objects	3,464 objects	2,550 LOC	2,550 LOC	1,337 LOC	1,337 LOC
Reading an input metamodel* (sec)	0.804	1.010	0.806	0.998	0.378	0.065	0.383	0.079
Reading an input model* (sec)	0.171	0.242	0.243	0.870	0.241	0.723	0.077	0.102
Reading an output metamodel* (sec)	0.006	0.020	0.006	0.022	–	–	–	–
Writing an output model* (sec)	0.111	0.793	0.181	2.726	–	–	–	–
Execution time (sec)	1.065	0.634	2.865	2.445	2.647	0.070	45.776	0.128
Total time (sec)	**2.157**	**2.699**	**4.101**	**7.061**	**3.266**	**0.858**	**46.236**	**0.309**

** Reading and writing models are operations handled by model-parsing and model-handling facilities*

sume that an increase in code size implies an increase in development time.

Regarding the runtime performance of model-to-model transformations, we tested a UML-to-Java transformation with two different model input sizes. In the one with an average-size model, the execution time of RubyTL is lower than ATL (including the overhead the Ruby interpreter incurs when reading the DSL definition). RubyTL's global performance is determined by the time it takes to read the UML input metamodel and the input model. The second model transformation test shows that the Ruby implementation doesn't scale as well as the Java one when the input size grows. Because the number of new created objects is large, garbage collection is important. If we disable garbage collection, the global time is 4.45 seconds (which is comparable to ATL).

Finally, and surprisingly, the code generation tests showed that our embedded DSL is considerably more efficient than MOFScript. The first test generated one file for each Java class and applied templates to write fields and method stubs. The second test implied traversing an input model several times and performing complex string manipulations. Because Ruby efficiently traverses lists and manipulates strings, implementing this code generation facility as an embedded DSL in Ruby is a good choice from a performance point of view.

Embedding a DSL in a host language is an alternative to constructing a compiler or interpreter. Furthermore, we consider it a general-purpose technique for improving program readability and maintenance. Developers have used this technique in the context of dynamic languages for years, but today's growing interest in DSLs is revitalizing both the embedding approach and dynamic languages. We can implement powerful, readable, maintainable DSLs in a short amount of development time for application domains in which the underlying dynamic language can satisfy the performance restrictions, as is the case for our tool. In fact, in most domains, performance isn't a critical issue, and trading execution time for development time really pays off.

Acknowledgments

This work was partially supported by Fundación Séneca (Murcia, Spain) grant 00648/PI/04. Thanks also to the reviewers and guest editors for their clear and insightful comments.

References

1. T. Stahl and M. Voelter, *Model-Driven Software Development*, John Wiley & Sons, 2006.
2. P. Hudak, "Modular Domain-Specific Languages and Tools," *Proc. 5th Int'l Conf. Software Reuse*, IEEE Press, 1998, pp. 134–142.
3. M. Fowler, "Language Workbenches: The Killer-App for Domain-Specific Languages?" June 2005; www.martinfowler.com/articles/languageWorkbench.html.
4. M. Mernik, J. Heering, and A.M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, Dec. 2005, pp. 316–344.
5. S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20, no. 5, Sept./Oct. 2003, pp. 42–45.
6. J. Sánchez Cuadrado and J. García Molina, "A Plugin-Based Language to Experiment with Model Transformations," *9th Int'l Conf. Model-Driven Eng. Languages and Systems*, LNCS 4199, Springer, 2006, pp. 336–350.

About the Authors

Jesús Sánchez Cuadrado is a PhD candidate at the University of Murcia. His research interests are model-driven development, model transformation languages, and dynamic languages. He received his master's in computer science from the University of Murcia. Contact him at the Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; jesusc@um.es.

Jesús García Molina is a professor of software design at the University of Murcia, where he leads the Software Technology Research Group. His research interests include model-driven development, domain-specific languages, and software processes. He received his PhD in science from the University of Murcia. Contact him at the Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; jmolina@um.es.