Rascal: Language Technology for Model-Driven Engineering

Jeroen van den Bos
CWI & NFI
jeroen@infuse.org

P.R. Griffioen
CWI
p.r.griffioen@cwi.nl

Tijs van der Storm
CWI
storm@cwi.nl

Abstract

Model-Driven Engineering (MDE) promises to increase productivity and quality by generating software systems from high-level, domain specific models. Domain-specific languages (DSLs) are a particular strategy for realizing MDE, where models are stored and processed as source code. Rascal is a DSL for source-code analysis and transformation, and as such represents an excellent tool for realizing a model-driven approach to software development. In this paper we substantiate this claim by discussing the implementation of three DSLs using Rascal. The first is an educational DSL aimed at exercising all aspects of MDE, including IDE support. The second, Derric, and third, Pacioli, are real-life DSLs from the domain of digital forensics and financial auditing respectively.

1 Introduction

MDE [9] is a promising approach for the future of software development. High-level models that describe the essential aspects of a sofware system do not suffer from the "programming as encryption" phenomenon [10] as much as ordinary programming does. Nevertheless, software construction using MDE technology involves a significant up-front investment in modeling tools, model-transformation tools, code generators etc. In this paper we argue that Rascal provides a flexible, low barrier-to-entry environment for implementing the essential components of MDE. Rascal [7, 8][1] is a DSL for source code analysis and transformation. It is a dedicated language for meta programming. This means that the typical input and output of a Rascal (meta-)program are other programs represented as source code. If one considers high-level, domain specific models as a special kind of source code, then Rascal's core application domains—that is, source code analysis and transformation—can be put to good use in the context of MDE. We substantiate our claim by showing how Rascal is currently used to develop three DSLs.

Organization In Section 2 we introduce Rascal. This provides enough background to appreciate the discussion of the three DSL implementations in Section 3. We highlight aspects of each DSL and its implementation and conclude with results on the size of the implementation. Finally, we summarize and address future work in Section 4.

[1] http://www.rascal-mpl.org

2 Rascal at a Glance

Rascal is a new and extensive language. Below we give a quick overview of the most distinguishing language features of Rascal.

Familiar syntax and control flow Although Rascal is a meta programming language, its syntax will appear familiar to anyone who is acquainted with C, Java, JavaScript or C#. Most of the usual suspects (while, if, block, assignment, declaration) obey the same syntactic (and often semantic) rules of curly-based general purpose languages.

Built-in data types and pattern matching Rascal features a large collection of built-in data types: integer, boolean, string, real, tuple, list, set, relation, map, parse trees, source locations (for identifying fragments of source code), date-time. Additionally, Rascal supports user-defined algebraic data types (ADTs). All data types have a syntactic, literal representation and can be used in pattern matching (with the exception of maps).

Domain-specific constructs Although Rascal has most of the features of a general purpose language, it also features a number of constructs tailored to the domain of source code analysis and transformation. One example is the `visit` statement for structure-shy traversal and transformation of arbitrary data, including parse trees. Other domain specific constructs include: comprehensions (for querying and creating sets, relations, maps or lists), regular expressions (for string matching), transitive closure (on binary relations), and a `solve` statement (for fix-point computations).

Arbitrary context-free grammars Context-free grammars are important for any source code analysis and transformation task, including the implementation of

DSLs. Hence, they are an integral part of Rascal. Rascal automatically generates parsers for these grammars using state-of-the-art scannerless, generalized parsing technology. All three DSLs are heavily dependent on this feature.

String templates Since code generation is such a common task in the domain of source code analysis and transformation, Rascal has built-in support for string templates. The evaluation of string templates maintains indentation with respect to user defined margins, enabling the generation of well-indented code. This feature is heavily used in the implementation of Derric (Subsection 3.2). An example is given in Subsection 3.1.

Integration with Java Sometimes a certain task requires a feature that is not provided by Rascal or its standard library, or is better implemented in a general purpose language, for instance for performance reasons, or the availability of third-party library. To cater for such situation, Rascal has a Foreign Function Interface (FFI) to Java. It is possible to interface to the Java world simply by annotating a Rascal function header with the qualified name of a Java method. This feature is currently used in Pacioli (Subsection 3.3).

IDE integration with Eclipse Programming without IDE support is hard to imagine these days. For this reason, Rascal can be installed as an Eclipse-based IDE plugin, featuring syntax highlighting, outlining, interactive visualization, and a Read-Eval-Print-Loop (REPL). Additionally, you can dynamically extend this IDE for your own languages. You may register parsers, outliners, hyperlinkers, type checkers etc.—all developed in Rascal—to get IDE support for your own languages without having to recompile and/or restart Eclipse.

3 MDE in Rascal: Three Examples

We describe three realistic applications of Rascal to model-driven software development:

- A DSL for Entity modeling. This educational example is based on our submission to the Language Workbench Competition 2011 [12][2] and illustrates syntax definition, code generation and modular language extension.
- Derric is a real-world DSL for describing binary file formats and is used in the digital forensics domain to generate data recovery tools. Despite the small size of their implementation, the Derric-based tools are comparable in functionality and performance to their industrial-strength counterparts currently used in practice.
- Pacioli is a DSL for "querying enterprises". The goal of Pacioli is to enable auditors to formulate relevant questions against heterogeneous data sets available within an enterprise. Pacioli is still in prototype stadium;—we discuss the current design.

[2] `http://www.languageworkbenches.net`

3.1 A simple DSL: Entities

3.1.1 Concrete and abstract syntax

Listing 1 Syntax definition of Entity models

```
import lang::entities::syntax::Layout;
import lang::entities::syntax::Ident;
import lang::entities::syntax::Types;

start syntax Entities = entities: Entity* entities;
syntax Entity =
    entity: "entity" Name name "{" Field* "}";
syntax Field = field: Type Ident name;
```

The Entities DSL allows you to declare entity types which model business objects. An entity has named fields, which are either primitively typed (integer, string, boolean), or contain a reference to another entity. An excerpt of the syntax definition of entity models using context-free grammars, is shown in Listing 1. First, auxiliary (syntax) modules are imported for defining Layout, Identifiers and Types. An Entity model then consists of a sequence of zero or more `Entity`s. An `Entity` starts with the keyword `entity`, followed by a name and a sequence of zero or more `Field`s. Finally, a `Field` consists of a Type (integer, string, boolean or Entity reference) and a name. The `Entities` nonterminal is the start symbol of the Entities grammar as indicated by the `start` keyword.

The syntax definition is used to parse textual entity models into parse trees. These parse trees can be automatically converted to abstract syntax trees (ASTs) for further processing. For every syntax production in the grammar for entities, there should be a corresponding constructor in the user-provided algebraic data type (ADT) of the abstract syntax. Lexical tokens are mapped to Rascal primitive types (e.g., `int`, `str` etc.).

3.1.2 Java code generation

In order to generate code from entity models, the code generator takes ASTs as input and uses *string templates* to produce textual code. The code generator is shown in Listing 2. It is defined using ordinary Rascal functions that produce string values using Rascal's built-in string templates. Consider, for example, the function `entity2java`. The string value returned by `entity2java` uses string interpolation in two ways. First, the name of the `Entity e` is directly spliced into the string via the interpolated expression `e.name.name`

Listing 2 Functions to generate Java source code from Entity models

```
public str entity2java(Entity e) {
  return "public class <e.name.name> {
         '<for (f <- e.fields) {>
         '  <field2java(f)>
         '<}>
         '}";
}
public str field2java(field(typ, n)) {
    <t, cn> = <type2java(typ), capitalize(n)>;
    return "private <t> <n>;
           'public <t> get<cn>() {
           '  return this.<n>;
           '}
           'public void set<cn>(<t> <n>) {
           '  this.<n> = <n>;
           '}";
}
```

between < and >. Next the body of the class is produced using an interpolated for-loop. This for-loop evaluates its body (a string template again) and concatenates the result of each iteration. For each field, the function `field2java` is called to generate a field, and getter and setter declarations. The single quote (') acts as margin: all white space to its left is discarded. Furthermore, every interpolated value is indented automatically relative to this margin. As a result the output of each consecutive call to `field2java` is nicely indented in the class definition. For many cases, this obviates the need for grammar-based formatters to generate readable code.

The function `field2java` is an example of pattern-based dispatch: the function signatures consists of an AST pattern, not just a list of (typed) formal parameters. This technique is a powerful tool for implementing languages in a modular fashion. For instance, the entity language could be extended so that entities support computed attributes. This will involve adding new production rules to the grammar, and new field constructors to the abstract syntax. Finally, the code generator would have to be extended. Using pattern-based dispatch this can be achieved by adding additional `field2java` declarations that match on the new AST constructors. No part of the original code generator has to be modified.

3.1.3 IDE support

No language can do without IDE support, and this includes DSLs. RASCAL exposes hooks into the Eclipse-based IMP [2, 3] framework for dynamically creating IDE support from within RASCAL. These hooks allow the dynamic registration of, for instance, parsers, type checkers, outliners and reference resolvers. A screen-shot of the generated IDE for the Entities language is shown in Figure 1. In the middle you see an editor containing

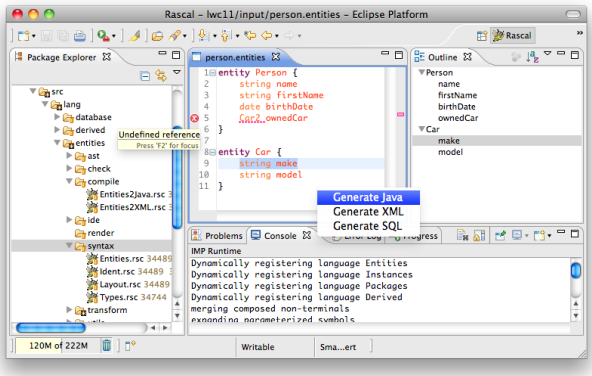

Figure 1: Screen-shot of the dynamically generated IDE for the Entities DSL

	Synt.	AST	Check	Gen.	IDE	*Total*
ENTITIES	45	23	27	32	72	199
INSTANCES	40	25	58	–	56	179
PACKAGES	+11	+11	114	+28	62	226
DERIVED	+28	+19	–	+32	+36	115
Total	124	78	199	92	226	719

Table 1: Non-comment lines-of-code (SLOC) per language, per aspect

a simple entity model. It has syntax highlighting and folding which are both based on the context-free grammar (cf. above). As you can see, there is an error: entity Person references an undefined entity Car2. On the right an outline is shown detailing the structure of this entity model. Clicking on an outline element highlights the corresponding source fragment. At the bottom of the editor pane, (a fragment of) the context-menu is shown, including entries to invoke various code generators.

3.1.4 Results

Table 1 shows an overview of the number of non-comment, non-blank lines of code of the complete suite of Entities-based languages. The INSTANCES languages is used to represent instances of entities. The PACKAGES extension adds name-spaces to base entities. Finally, DERIVED extends the entities language with support for derived attributes. The figures are shown per language and per aspect (Syntax, AST, Checker, Generator and IDE). A +-sign indicates that the number describes an extension of the base entities language.

3.2 A DSL for Digital Forensics: Derric

Another MDE application of RASCAL is in the domain of digital forensics. Investigations in this area are often related to recovery of deleted, obfuscated, hidden or

otherwise difficult to access data. The software tools to recover such data require lots of modifications to deal with different variants of file formats, file systems, encodings etc. Additionally they are also required to return a result within a reasonable amount of time on data sets in the terabyte range. We are investigating a model-driven approach to this problem by designing a DSL, DERRIC, to easily express the data structures of interest [1,11]. From these descriptions we generate high performance tools for specific forensic applications.

DERRIC is a declarative data description language used to describe complex and large-scale binary file formats, such as video codecs and embedded memory layouts. It is essentially a very fine-grained grammar formalism to precisely capture the way files are stored. For example, it is possible to define a component of a file format to be a 21-bit unsigned integer that is always stored in big endian byte order. Figure 3 shows an excerpt of a DERRIC specification for the JPEG image file format.

Listing 3 Excerpt of JPEG in DERRIC

```
structures
APP0JFIF {
  marker: 0xFF, 0xE0;
  length: lengthOf(rgb) + (offset(rgb)
          - offset(identifier)) size 2;
  identifier: "JFIF", 0;
  version: expected 1, 2;
  units: 0 | 1 | 2;
  xthumbnail: size 1;
  ythumbnail: size 1;
  rgb: size xthumbnail * ythumbnail * 3;
}

DHT {
  marker: 0xFF, 0xC4;
  length: size 2;
  data: size length - lengthOf(marker);
}

SOS = DHT {
  marker: 0xFF, 0xDA;
  compressedData:
    unknown terminatedBefore 0xFF, !0x00;
}
```

DERRIC is a language that can be used for many digital forensics applications. Currently, we have implemented DERRIC in RASCAL and have used it to develop a digital forensics data recovery tool called EXCAVATOR (see Figure 2). EXCAVATOR is used for *file carving*: recovering files from storage devices without using file system meta-data (these are often unavailable or incomplete). EXCAVATOR is implemented as a code generator. It generates a validator that checks whether a series of bytes conforms (or might conform) to a certain file format.

The steps in the implementation of EXCAVATOR are

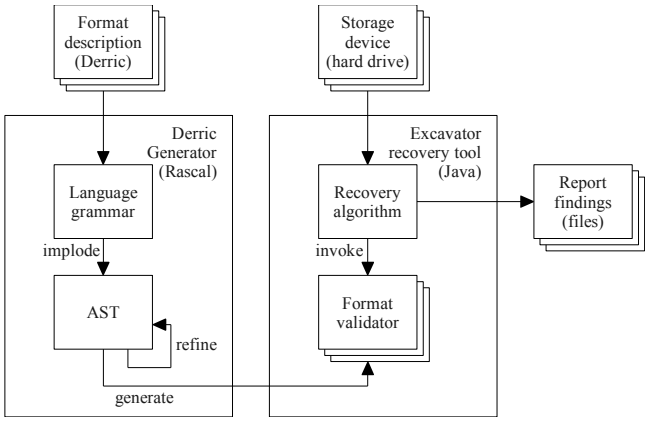

Figure 2: Use of DERRIC in EXCAVATOR

shown in Figure 2. The first step consists of parsing the DERRIC source text and converting the parse tree to an AST (implode). This AST is the starting point of a series of refinements where each step takes a complete AST as input and produces a modified AST (of the same type) that is better suited to the final goal of generating a validator for the described format.

One refinement consists of annotating the AST with derived values required in later stages. For instance, such values could include size and offset information, used by compile-time expressions in the descriptions (e.g., `lengthOf` and `offset` in the `length` field of the `APP0JFIF` structure in Figure 3).

Another refinement consists of simplifying the AST by performing compiler optimizations such as constant folding and propagation. For instance, string constants (e.g., the definition of `identifier` of `APP0JFIF` in Figure 3) are desugared to a list of bytes corresponding to the string in the defined encoding, so that the generated code only has to do simple byte comparisons.

Finally, the DERRIC implementation supports a number of optional refinements, which can be executed on demand. An example is to replace parts of the AST with alternatives that result in code that is either faster or more precise. As such, this allows users to configure the trade-off between accuracy and runtime performance on a case-by-case basis.

The final step of Figure 2 consists of generating code. We currently have a code generator that generates Java code and as a result, all DERRIC types are annotated with a target type that maps cleanly onto Java types. For instance, a 32-bit unsigned type will be stored in a 64-bit signed type since Java does not support unsigned types. The resulting code is then loaded by the EXCAVATOR runtime system to recover files from disk images.

Component	Lang	Size (SLOC)
Syntax	RASCAL	58
Code generator	RASCAL	971
Recovery Code	Java	480
Base library	Java	1081
Total		2590

Table 2: Sizes of the DERRIC-based EXCAVATOR components

3.2.1 Results

To evaluate EXCAVATOR, we have compared it to three industrial-strength carving tools on a set of standard benchmarks. Our evaluation shows that even though the implementation is very small (see Table 2), it performs as good as the competing tools both in terms of functionality and runtime performance, while providing a much higher level of flexibility to the user. For more details we refer the reader to [11].

3.3 Querying the Enterprise: Pacioli

Earlier work in computational auditing showed that enterprise process models, known as *value cycles*, are essential in answering important auditing questions, pertaining to segregation of duties and spanning reconciliation[3] [4, 5]. Figure 3 shows a simple, but realistic, example of such a value cycle for a small maintenance company. The circles represent accounts (+ means asset, − means liability) and the boxes are transactions. The numbers on the edges represent the effect on the linked accounts. For instance, the buy transaction adds 1000 units of supplies to the Supplies account and adds 5 units of money to the creditors account.

Although the value cycles are a prime example of graphical models (a kind of petri-nets), it turns out that they can be conveniently formalized as matrices. Moreover, various other data sets (e.g. journal files, inventory listings, sales records) that are relevant for auditors are amenable to a similar formalization. For this reason, PACIOLI is based on linear algebra. Both value cycles and financial data sets can be represented as matrices. The ultimate goal of PACIOLI is to provide auditors with the means to conveniently analyze these enterprise data in efficient and consistent way.

PACIOLI is currently in prototype stadium. We have defined a kernel version of PACIOLI which basically is a lambda-calculus for linear algebra. We expect to desugar the high-level syntax to this lower level language. Instead of directly interpreting or generating code, the kernel version of PACIOLI is translated to virtual machine instructions, which are then executed by a virtual machine

[3]This is the normative relationship between an enterprise's input and output; it is equivalent to the presence of a so-called *value jump* in the value cycle.

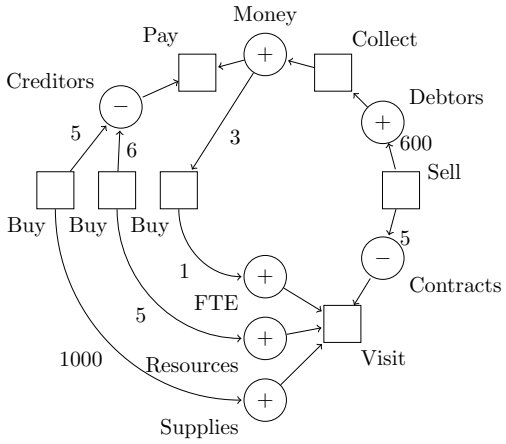

Figure 3: Value-cycle for a maintenance company

for matrices, called MVM. MVM features primitive operations on matrices (transposition, multiplication, etc.), deals with register allocation and accesses external data sources. MVM is currently implemented in Java. The front-end of PACIOLI, however, is fully implemented in RASCAL. It currently includes a novel type-inference algorithm to infer the units of measure of PACIOLI expressions [6].

The high-level design of PACIOLI is to implement the heavy lifting of the front-end and static analyses in RASCAL, whereas the dynamic execution of the code is done by an interpreter (MVM) which is implemented in a general purpose language. MVM is integrated into RASCAL through RASCAL's FFI.

3.3.1 Questionnaires

The typical auditing process is well supported by filling out large questionnaires about an enterprise. Such questionnaires consist of forms containing input elements (e.g., checkboxes, text fields, dropdown boxes etc.) which may be conditionally enabled. This conditional logic gives these questionairres a highly interactive nature. Currently, we are working on a DSL complementary to PACIOLI for describing such questionnaires. It allows the declaration of inputs and outputs and their representation. The expressions used in this language, will be based on PACIOLI.

3.3.2 Initial Results

The PACIOLI prototype is very much work in progress. Nevertheless, we have displayed the sizes of the different components of the PACIOLI implementation in Table 3. The virtual machine component is currently the largest: this includes runtimes support for units of measure, register allocation, and a simple parser for byte code instructions. For the matrix computations a third-party library

Component	Language	Size (SLOC)
Syntax	Rascal	25
Type inference	Rascal	451
Code generator	Rascal	59
Virtual machine	Java	1525
Total		2060

Table 3: Size of the implementation of KernelPacioli

is used[4]. The type inferencer includes data types for describing matrix types, vector types and unit systems, and a unification procedure. The code generator simply converts λ-expressions to virtual machine instructions.

4 Conclusion

Rascal is a DSL for developing source-code analysis and transformation tools. If models are represented as source code, which is typically the case in DSL development, Rascal can be used to implement the essential components of MDE: the analysis and transformation of high-level models. We have shown three elaborate examples of (families of) DSLs developed using Rascal: an educational DSL for entities, and two real-life DSLs from the domain of digital forensics (Derric) and financial auditing (Pacioli). In each of these cases, the results are promising: the implementations are very small and easy to modify. Yet, both Derric and Pacioli are non-trivial languages. Derric generates high-performance file validators from high-level file formats. Pacioli performs state-of-the-art type inference on linear algebra expressions, supporting matrices with heterogeneous units.

A number of key features of Rascal play an important role in obtaining these results:

- Declarative syntax definition using arbitrary context-free grammars: this poses no restriction on a DSL's syntax and supports modular syntax definition.
- As seen in the entities family of DSLs, Rascal supports modular language extension. Grammars, ADTs and functions can be extended in a modular fashion.
- Productivity is improved through Rascal's dedicated features for querying models (comprehensions), transforming models (pattern matching and traversal), and code generation (string templates).
- IDE support comes virtually for free through Rascal's IDE hooks into Eclipse.

Rascal is open source software and can be downloaded from `http://www.rascal-mpl.org`. The source code of the entities languages can be found at `http://svn.rascal-mpl.org/lwc/trunk/lwc11`. The current prototype of Pacioli can be found at `http://svn.rascal-mpl.org/pacioli`. The source code of Derric can found at `http://svn.rascal-mpl.org/derricbase`.

[4] `http://commons.apache.org/math`

References

[1] L. Aronson and J. van den Bos. Towards an engineering approach to file carver construction. In *COMPSACW'11*, pages 368–373. IEEE, 2011.

[2] Ph. Charles, R.M. Fuhrer, S.M. Sutton, Jr., E. Duesterwald, and J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. In *OOPSLA'09*, pages 191–206. ACM, 2009.

[3] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton. IMP: A Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse. In *ASE'07*, pages 485–488. ACM, 2007.

[4] Philip I. Elsas. *Computational Auditing*. PhD thesis, Vrij Universiteit Amsterdam, 1996.

[5] P. R. Griffioen, P. I. Elsas, and Reind P. van de Riet. Analysing enterprises: The value cycle approach. In *DEXA'00*. Springer-Verlag, 2000.

[6] Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996.

[7] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM'09*, pages 168–177. IEEE, 2009.

[8] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In *GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[9] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[10] Charles Simonyi. Is programming a form of encryption? `http://blog.intentsoft.com/intentional_software/2005/04/dummy_post_1.html`, April 2008.

[11] J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *ICSE'11*, pages 671–680. ACM Press, 2011.

[12] T. van der Storm. The Rascal Language Workbench. CWI Technical Report SEN-1111, CWI, May 2011.