The Rascal meta-programming language — a lab for software analysis, transformation, generation & visualization

Mark Hills Anastasia Izmaylova Paul Klint Atze van der Ploeg Tijs van der Storm
Jurgen Vinju
Centrum Wiskunde & Informatica — SEN1
INRIA Lille — ATEAMS

Abstract

This paper summarizes the goals and features of a domain specific programming language called Rascal. On the one hand it is designed to facilitate software research — research about software in general. On the other hand Rascal is applied to specific software portfolios as well, as a means to improve them and as a means to learn to understand them.

Specifically, Rascal is used create tools that analyze, transform, generate or visualize source code of software products. Such tools are motivated by the need to improve quality of existing software or the need to lower its cost-of-ownership. More generally such tools are created to build laboratory experiments that observe and measure quality, or try and improve software quality, etc.

In this paper we provide an overview of Rascal as a "domain specific language for meta programming". We first explain its goals and then its features. We end by highlighting some example applications in the area of software analysis and transformation.

1 Introduction

We are generally interested in high-quality source code and means to assess quality and means to improve quality of source code. One example aspect of software quality is maintainability, which is governed mostly by understandability. A relevant question to any (manager of a) software engineer is: "how easy to understand is this particular source code?". After having asked this question many times for many different products, a different question arises: "how can we quickly and accurately assess the understandability of a large piece of software?". Another question would be: "what design decisions could have been made to prevent this low understandability?".

Our research mission is to produce both new tools that help answering such questions about specific software tools, and to produce insights (theory) about software in general. We state that there usually is no general answer, one that goes for any software product in any kind of circumstances. Rather, we want to help ourselves and other software engineers to answer such questions over and over again in different industrial, academic and governmental contexts.

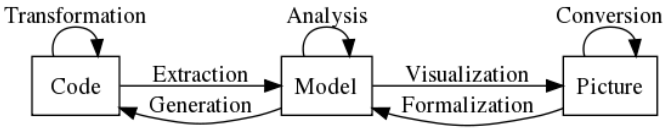

Figure 1: Domain of meta-programming.

2 Goals

Our research strategy in this field of software quality is to use the Rascal language as a means to rapidly construct analysis tools, software transformation and generation tools and software visualization tools. Each tool is a contribution in itself, if the goal is to analyze, transform, generate or visualize complex software systems. Each tool might also be applied to many different (kinds of) software systems, to try and learn lessons about software in general.

The goal of Rascal is to be a programming language that allows to rapidly construct *any* kinds of analysis, transformation, generation or visualization of source code that is written in *any* kind of programming language or combination of programming languages. Figure 1 summarizes this domain. The challenges that come with this goal are:

- Scope and abstraction level — how specific should Rascal be, what should remain open and what needs automation?
- Concepts and language — what is the audience of users of Rascal, and how does it influence the shape of this language?
- Variation — software is written in many kinds of programming languages, using many kinds of frameworks and libraries and written by many kinds of different people, deployed on many different kinds of computer hardware.

- Integration — software analysis, transformation, generation and visualization are classically separated, both conceptually as well as technically. We want to integrate all of these into a single programming language that allows their seamless integration.
- Speed — any analysis suffers from the speed versus accuracy trade-off, and we need to be able to process large portfolios. Transformations have similar trade-offs, where we trade simple and speedy but "noisy" transformation tools (all indentation is lost), for high-fidelity but complex tools that keep all indentation and comments.

These five challenges are inter-related in an intrinsic manner. For example, the need to balance the speed to precision trade-off requires a level of abstraction that still allows programming customized algorithms. The alternative would be to build specific algorithms into the language as primitive concepts. On the other hand, there are standard text-book algorithms in software analysis that should be available to Rascal users (using libraries). Another example is that to be able to accept a wide range of programming languages, we need powerful but slightly slower parsing algorithms.

In this paper we do not evaluate whether or not Rascal has met the above challenges. Rather we present some of its features and explain how we applied it. This should provide the reader with information to help decide to investigate it further. This paper also does not provide source code examples of Rascal, but it describes Rascal code from a birds-eye perspective.

3 Features

This section lists the basic design decisions of Rascal. This information is interesting for programmers to be able to decide whether or not they can understand and use this language.

3.1 Procedures and control-flow

The core of Rascal is a normal programming language with procedures, functions, and structured control-flow with exceptions. You will find foreach loops, conditional statements, switch blocks, etc. Their basic use is no different from a language like C or Java, which allows programmers to start with Rascal using their knowledge of programming in general[1].

However, these core control features do provide the fundamental syntax and semantics for the kind of features needed for meta programming algorithms in specific, like pattern matching, backtracking and traversal over large data-structures (see below).

More advanced users will appreciate the generic and higher-order features or Rascal as well as closures. Such features help in creating reusable libraries of data-structures and algorithms.

3.2 Immutable abstract and concrete data

From functional programming we borrowed that all data is immutable. The kinds of (collections of) algorithms and inter-mediate data-structures that go with interesting software analyses and transformations are complicated. Well known issues with side-effects and referential integrity in mutable data-structures are avoided in Rascal. By not allowing one part of a program to affect the data that another part is working on, we hope to provide a safe environment to experiment, as well as an environment to construct stable tools.

By concrete data we mean that Rascal allows to define the syntax of programming languages. It generates a parser for any context-free grammar, allows to disambiguate the syntax using any information and provides ways to match and transform parse trees. This is all meant to allow analysis and transformation to be expressed at the level of abstraction of the programming language that is analyzed. An expert in the object programming language at hand (say Java), should be able to read a Rascal program that manipulates programs in that language (say, a Java refactoring tool) quite easily.

By abstract data we mean the basic data types, such as integers and strings. Also, we provide *algebraic data-types*, to construct hierarchical models of software and sets and *n-ary relations* to construct graph-like models of software. You should think of abstract syntax trees, control-flow graphs, data-dependency graphs, etc. Rascal provides a library of reusable tools that can analyze and transform such models, and it is very common to write highly specific code that manipulates these.

Most Rascal programs follow the "EASY paradigm": Extract, Analyze, SYnthesize. Source code is first parsed, or tokenized, then basic facts are extracted and a model or models of it are constructed using the data constructs mentioned above. Then we search and query this data to extend it with aggregated or derived results and finally we traverse, substitute and expand the data to synthesize a new result (Figure 2).

3.3 Search and query

Any data in Rascal can be searched using pattern matching expressions that include:

- First-order constructor matching, as in term rewriting

[1] This is opposed to our previous meta-programming system, the ASF+SDF Meta-Environment [19], which was appreciated most by programmers with a background in theoretical computer science.

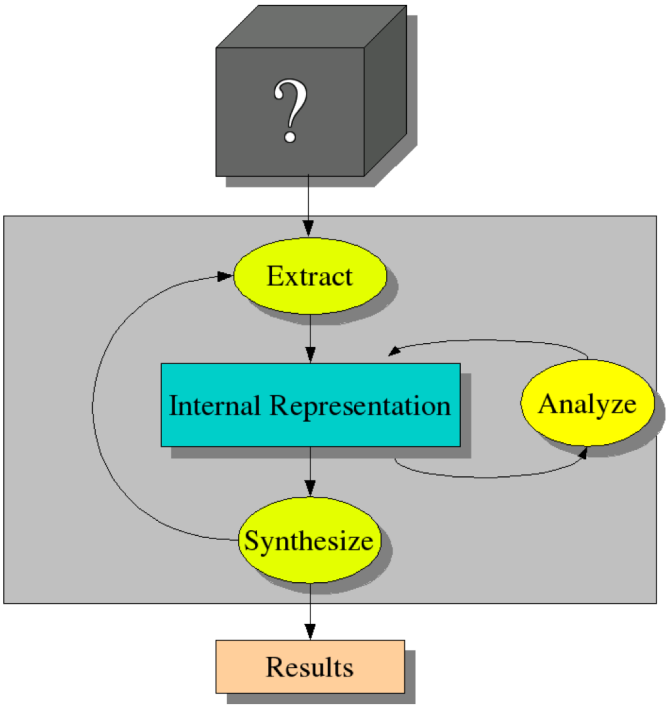

Figure 2: The EASY paradigm [15], from black box system, via internal models, to improved source code, metrics or visualizations, using Extraction, Analysis and Synthesis primitives.

- List patterns (matching modulo associativity), and set patterns (matching modulo associativity, commutativity and idempotence).
- Path (deep) matching, like in XPath[2].
- Negative matching and alternative matching

Such patterns can be used in any control flow construct, like if-then-else and switch. In these constructs the patterns control the decision which control path to take: if and only if a pattern matches it returns true.

In the function definitions the patterns guide which function body is executed (dynamic dispatch) — when all parameter patterns match the function body is executed. In both cases patterns bind variables that can be used in the body of the conditional construct or the body of an overloaded function. These feature makes Rascal as powerful as any term rewriting language, such as ASF+SDF [9].

Backtracking is an essential property of patterns, to allow searching in a large data-space. All of Rascal's control flow and dispatch constructs allow explicit control of backtracking behavior without having to write boilerplate code.

[2] `http://www.w3.org/TR/xpath`

A different way of query is the manipulation of sets and relations using calculus. Rascal provides the basic relational calculus operators as primitives as well as list, set, and map comprehensions. This expression languages allows programmers to start from text-book explanations of software analyses (e.g. from [1]), and gradually expand them to more specific or more advanced versions. The application of relational calculus is a proven method in software reverse engineering [12, 3].

3.4 Traversal, substitution and expansion

In general purpose programming languages the recursive traversal of complex heterogeneous data-structures is hard work. There are design patterns, like the Visitor design pattern, to guide such solutions, but it remains a lot of boilerplate work to implement them. Rascal, like other meta programming systems, automates the traversal of any data (abstract syntax trees, relational models) in a type safe manner using a control-flow statement called "visit". Visit allows the programmer to travel to any part of a data-structure using a pattern to match it. Then she may trigger some code, or substitute the part by a new part.

To be able to synthesize new source code or new models Rascal provides string templates (php-like), concrete syntax templates and the aforementioned comprehensions. The string templates have an auto-indent feature, simply because generated code often needs to be read by programmers. The concrete syntax templates ensure that any code that is generated will be syntactically correct.

3.5 Modularity and extensibility

There is a strong need for modular and extensible in meta programs: There are dialects of programming languages, variants of algorithms, layered models of representation etc. To be able to manage such variation, Rascal has a module system. Modules may depend on each other or extend each other.

Inside these modules all data and function definitions are extensible (concrete syntax, abstract data, and function definitions), by simply adding alternative definitions. When modules extend each-other definitions can also be removed or overridden.

The net result of these modularity features is that tools written in Rascal can be written to be easily adaptable to deal with different dialects of Cobol, Java or C.

4 Example applications

4.1 Refactoring to use generics

Contrary to the current paper, in [14] we did try and evaluate the fitness of Rascal, in this case for con-

structing refactoring tools. The experiment was to reconstruct an existing refactoring tool for Featherweight Generic Java [14] in Rascal and to compare its implementation to published language specifications and pseudocode. The goal of the tool is to find uses of generic classes (such as `List<X>`), which do not bind the type parameters. Then, the surrounding code is analyzed to figure out what the type parameter should be and finally the code is changed to use that type.

The use of generics makes code safer (less bugs), but the move from code that does not use generics to code that does is costly and perhaps not worth the effort. Constructing such a refactoring is therefore one of the ways to lower cost-of-ownership of software products.

This prototype was a proof-of-concept that is now being generalized and extended to cover interesting Java refactoring tools.

4.2 Refactoring from visitor design pattern to interpreter design pattern

In this study we were interested to learn the implications of a design choice: to use the interpreter design pattern, or to use the visitor design pattern? The book on design patterns [10] explains us why in general a design pattern should be good, but not how to pick between several patterns that appear to be good but in different ways.

We constructed a tool to transform a large Java program, and remove most applications of Visitor to replace them later by Interpreter. The tool is semi-automatic and requires an extensive analysis of the existing system to compute preconditions and locations of change.

The result of the study showed, counter-intuitively, that for longer-lived projects the Visitor design pattern is probably the safest bet if you want more maintainable software. This is surprising because conceptually Visitor is quite a bit more complex than the Interpreter design pattern.

4.3 A compiler for Oberon-0

We joined in a competition between several meta programming systems[3]. The goal is to construct a highly modular and full-fledged compiler for the Oberon-0 language. Using Rascal we also "generated" a reasonably feature full Eclipse IDE plugin for Oberon-0 including syntax highlighting, reference resolving, outline, folding, etc.

A team of 7 researchers worked on the compiler and the IDE for approximately two weeks, each working on a separate part to construct this exercise. The result is indeed a modular implementation. What is interesting to note here is that some of these parts are clearly different in style, as they are written by programmers with different backgrounds.

Compilers are not per sé the intended application domain of Rascal, but compilers do analysis and transformation of code and models just as any other meta programming application does.

4.4 Domain specific languages

Rascal is also used to construct code generators and other meta tooling for domain specific languages. We did not emphasize this in the current paper, as there is other material on that aspect of Rascal[4]. Example domains that have been covered are Digital Forensics [18] and Computational Auditing. We also participated in the Language Workbench Competition 2011, to construct and IDE for a simple textual modeling DSL[5].

4.5 Extracting domain concepts from source code

A question in the application of domain specific languages to software construction is: "which language should we make?" One answer lies in analyzing existing software to see which concepts are represented and how they relate. Especially older and successful software may be trusted to "cover a domain". In this study we used techniques from information retrieval to rank and filter identifiers that occur in source code. The goal is to separate the merely technical concepts from the concepts that are relevant from the problem domain.

Rascal is used in this exercise to manipulate the models for the input and the output of a tool called Formal Concept Analysis (FCA). FCA can be used to detect latent (implicit) relationships. Using a number of heuristics that should be indicators for domain concepts the output of FCA is ranked and filtered to get high accuracy in domain concept retrieval. The results are published in a masters thesis [16].

From these results, which produce the names of the most important domain concepts rather accurately, we now need to continue to identify the relations between them. A fully automatically extracted domain model would indeed be a good starting point for the domain analysis for new domain specific languages.

4.6 The quality of method names

In this study the relation between the contents of method bodies (i.e. procedures) and their name was studied. First we extract facts about the bodies and their names from the source code using Rascal. Then we use Formal Concept Analysis again. After post-processing the output of FCA, the results can be used

[3] For the international workshop on Language Descriptions Tools and Applications (LDTA 2011)

[4] Another paper was submitted to ICT.OPEN to highlight this application area of Rascal.

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

to estimate whether a name of method is "strange" as compared to other methods which have similar bodies.

Even if the results of such a study are hard to validate in-the-large, it is worth investigating the usefulness of such a tool. Our results show promising precision in detecting bad names. Bad names have shown to be an important factor in misunderstanding source code, which can lead to subtle and expensive bugs in complex software products. The results will be published in a master thesis by the end of 2011.

4.7 Connecting to an existing program analysis framework

In [11] we described how Rascal can be used to create a front-end and an IDE for a programming language that is otherwise interpreted by another meta programming framework.

Using Maude [6] and the K framework one can construct program analyses in a modular and formal fashion. Rascal is strong in parsing and extracting facts from real programs. To be able to link existing tools with Rascal instead of having to redo things in Rascal is an important design consideration that trades purity for pragmatism.

4.8 Analyzing traces using attribute grammars

We constructed a tool [8] that allows the specification of attribute grammars to define assertions on Java call traces. The tool processes the specifications and generates code to implement the assertions using ANTLR and weaves in the appropriate calls to tracing code into Java programs.

This prototype is a parser, aspect weaver, and code generator in a few hundred lines of code, which shows the diversity of applications and Rascal's ability of integrating different technical domains into a single application.

4.9 A DSL for visualization

This DSL, called FIGURE, is actually a standard library component of Rascal. Its intended use is to construct software and metric visualizations and reports, that can be composed modularly and browsed inter-actively. The library is written partly in Rascal and partly in Java to integrate into SWT and Eclipse.

4.10 Observations

From the above applications we noticed that some of the goals have been reached, while others are still far away.

- Master students learn basic Rascal in a day or less, so it is accessible to programmers from a broad range of backgrounds.
- Some programmers write in ten lines what others write in hundreds. Rascal provides very high-level abstractions for pattern matching, backtracking, traversal and search, but since it has general primitives as well they can be ignored by the novice programmer.
- Our initial implementation as an interpreter in Rascal is flexible for maintenance, but is not efficient enough. We need to invest in optimization and re-implementation.
- Pseudocode of algorithms from text-books and academic publications can be expressed almost literally in Rascal, or at least just as short [14].
- Applications written in Rascal are surprisingly short, typically a number of pages, as compared to their counter-parts written in Java or C. Nevertheless, we also observe that constructing Rascal programs requires a comparable intellectual effort: the problem you are approaching using Rascal does not become easier to solve, but it is much easier to oversee and to experiment with on this level of abstraction.
- We have experimented with a wide range of different kinds of meta programs, which is a good sign.

5 Acknowledgements

Rascal was inspired by our study and use of many other (open-source) meta programming systems: ASF+SDF [9], Stratego [20], ELAN [4], Grok [12], Crocopat [3], RScript [13], Maude [6], TXL [7], DMS [2], Refine [17], etc.

We also thank the many students of the Master Software Engineering at Universiteit van Amsterdam, who have applied Rascal in their course on Software Evolution and in some of their master projects. Christian Köppe [16] and Jouke Stoel have worked on the applications for name analysis and domain concept recovery.

Two important caveats in the application of Rascal are now that it is slow in some applications and that its type-checker has not been integrated. Nevertheless, we consider the current implementation of Rascal to be of enough quality for proof-of-concept kind of applications.

Rascal is open-source software. Its IDE makes use of the Eclipse IMP [5] platform[6]. The source code of Rascal itself has been accepted for contribution to `Eclipse.org` in 2011.

6 Conclusion

This paper provided a motivation, an overview and some showcases of Rascal. It is a programming language

[6] `http://www.eclipse.org/imp`

to help in software engineering and software engineering research. We hope to have given enough detail to help deciding to investigate it further or not.

We are interested in running case studies and constructing tools that are specific to industrial, governmental or academic software. Specifically internships for analyzing, transforming or visualizing software systems are welcome. Please contact us if you are interested: `http://www.rascal-mpl.org`.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] I. Baxter, P. Pidgeon, and M. Mehlich. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proc. ICSE'04*, pages 625–634. IEEE, 2004.

[3] D. Beyer. Relational programming with CrocoPat. In *Proc. ICSE'06*, pages 807–810. ACM Press, 2006.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[5] Ph. Charles, R.M. Fuhrer, S.M. Sutton, Jr., E. Duesterwald, and J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. *SIGPLAN Not.*, 44:191–206, October 2009.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[7] J.R. Cordy. Excerpts from the TXL cookbook. In *Post-Proc. GTTSE'09*, volume 6491 of *LNCS*, pages 27–91. Springer, 2011.

[8] S. de Gouw, F. de Boer, and J.J. Vinju. Prototyping a tool environment for run-time assertion checking in jml with communication histories. In *12th Workshop on Formal Techniques for Java-like Programs*, 2010.

[9] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

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

[11] M. Hills, P. Klint, and J.J. Vinju. RLSRunner: Linking Rascal with K for Program Analysis. In *Proceedings of the 4th International Conference on Software Language Engineering (SLE'11)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To Appear.

[12] R.C. Holt. Grokking Software Architecture. In *Proc. WCRE'08*, pages 5–14. IEEE, 2008.

[13] P. Klint. Using Rscript for software analysis. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.

[14] P. Klint, T. van der Storm, and J.J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. SCAM'09*, pages 168–177. IEEE, 2009.

[15] P. Klint, T. van der Storm, and J.J. Vinju. EASY Meta-programming with Rascal. In *Post-Proc. GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[16] C. Köppe. Automated domain knowledge recovery from source code using information retrieval techniques. Master's thesis, Universiteit van Amsterdam, August 2011.

[17] Reasoning. Refine User's Guide. Reasoning Systems Incorporated, 1994.

[18] J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 671–680, 2011.

[19] M.G.J. van den Brand, M. Bruntink, G.R. Economopoulos, H.A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J.J. Vinju. Using The Meta-environment for Maintenance and Renovation. In *Proc. CSMR'07*, pages 331–332. IEEE, 2007.

[20] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.