Rascal: From Algebraic Specification to Meta-Programming

Jeroen van den Bos[1,3] Mark Hills[1,2] Paul Klint[1,2] Tijs van der Storm[1,2]
Jurgen J. Vinju[1,2]
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands[1]
INRIA Nord-Europe, Lille, France[2]
Netherlands Forensic Institute, The Hague, The Netherlands[3]
{Jeroen.van.den.Bos,Mark.Hills,Paul.Klint,Tijs.van.der.Storm,Jurgen.Vinju}@cwi.nl

Algebraic specification has a long tradition in bridging the gap between specification and programming by making specifications executable. Building on extensive experience in designing, implementing and using *specification formalisms* that are based on algebraic specification and term rewriting (namely ASF and ASF+SDF), we are now focusing on using the best concepts from algebraic specification and integrating these into a new *programming language*: RASCAL. This language is easy to learn by non-experts but is also scalable to very large meta-programming applications.

We explain the algebraic roots of RASCAL and its main application areas: software analysis, software transformation, and design and implementation of domain-specific languages. Some example applications in the domain of Model-Driven Engineering (MDE) are described to illustrate this.

1 Introduction

Algebraic specification has a long tradition in bridging the gap between specification and programming [22, 18]. There has always been a tension between algebraic specifications as mathematical objects with certain properties and algebraic specifications as executable objects. This tension is nicely summarized by the label "algebraic programming".

Experience has taught us that when a formalism is made executable it effectively becomes a programming language. Even if the language operates on a higher level of abstraction, common engineering issues arise when developing and maintaining specifications. Like programs, executable specifications have bugs and thus require debugging; they are slow and thus need to be optimized; they are complex and thus need to be analyzed in order to be understood; and finally their life extends beyond the first version and thus they need to be maintained to accommodate new requirements. The nature of algebraic specification exacerbates the difficulty of some of these common software engineering tasks. This is due to the inherent non-deterministic nature of algebraic specification and the complexity of highly optimized execution platforms (term rewriters). What actually happens at run-time, and why and when, is conceptually far removed from what is specified.

We describe the language RASCAL we are currently working on. It is a dedicated language for meta-programming (Figure 1). This means that programs can be the input and output of RASCAL programs. RASCAL's primary applications are in software analysis, software transformation and design and implementation of domain-specific languages. The word "software" should here be interpreted in a broad sense: subjects for analysis and transformation include source code, models and meta-data such as documentation, version histories, bug trackers, log files, execution traces, and more. RASCAL is rooted in algebraic programming and is targeted at solving large, real-life, problems.

The goal of this paper is to explain why and how RASCAL is not an algebraic specification formalism with programming language features, but rather a programming language with algebraic specification

Submitted to:
AMMSE 2011

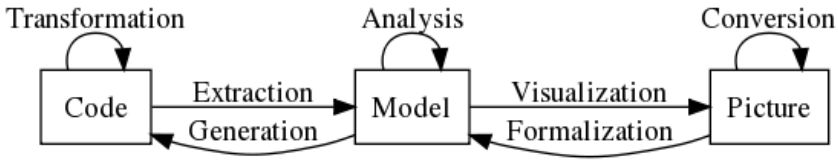

Figure 1: The meta-programming domain: three layers of software representation with transitions.

features. The plan of the paper is to first summarize in Section 2 the lessons we have learned in designing and applying several languages for algebraic programming. These lessons form the starting point for RASCAL's design requirements. We sketch the resulting language and explain where it deviates from the purely algebraic paradigm in Section 3. In Section 4 we illustrate RASCAL by detailing three applications in the domain of Model-Driven Engineering (MDE), one of the prime application areas of RASCAL, as well as one linking RASCAL with existing algebraic specifications. We conclude in Section 5.

2 An Algebraic Perspective to Meta-Programming

RASCAL succeeds ASF+SDF [7, 6] as our platform for experimenting with and implementing language definitions and other meta-programs. ASF+SDF consists of two parts: ASF, the Algebraic Specification Formalism (used for rewriting terms), and SDF, the Syntax Definition Formalism (for specifying grammars). Below we summarize, with hind-sight, our experiences with ASF+SDF that have motivated the design of RASCAL. We first focus on ASF and after that specifically address the lessons we have learned from its combination with SDF.

2.1 Experience with ASF+SDF: the case of ASF

Our first focal point is ASF, which originally was a standalone formalism (ASF, [2]):

- An ASF specification is modular. Modules import each other, while optionally instantiating sort parameters and/or renaming sorts.
- Each module contains function signatures that declare (first order) typed functions. Functions can be either constructors (i.e. function names that can occur in normal forms) or defined functions (i.e. functions that will be eliminated by applying equations). Originally, the *add* function on natural numbers (represented by the sort `NAT`) was declared in ASF as: `add: NAT # NAT -> NAT`. The latest incarnation of of ASF+SDF uses SDF [19, 33] to define signatures (cf. below).
- An equation in ASF consists of an equality between two terms that respects the declared many sorted function signatures. Optionally, this equality may be preceded by one or more conditions that can be equalities or inequalities between terms. A conditional equality does not imply full unification: only one side of a positive condition may introduce variables and inequalities may not introduce variables at all.
- Equations may be marked as "default" equations that apply if no other relevant equations apply.
- Equations can use list matching—pattern matching modulo associativity of list construction—facilitating handling programming language constructs like statement and parameter lists.

The *add* function mentioned above could be defined in ASF using the following equations (assuming appropriate definitions for the `NAT` constant `0` and the successor function `succ`):

```
[add1] add(X, 0)       = X
[add2] add(X, succ(Y)) = succ(add(X, Y))
```

The design and use of ASF have always been focused on executability by way of (left-most innermost) term rewriting. Initial implementations of ASF compiled to Prolog, later ones (in the context of ASF+SDF) to highly efficient C code. As part of ASF+SDF, ASF has been successfully used for the analysis and transformation of multi-million line software systems and for the implementation of industrial domain-specific languages [9]. These experiences have lead to the following observations:

- Although ASF allows arbitrary rewrite rules, programmers almost without exception write strongly confluent and terminating sets of rules. They do that by introducing enough intermediate function symbols and by strictly using default rules when not all cases need to be matched by a function symbol. In other words, a locally non-confluent specification is almost always considered to be buggy rather than simply declarative.
- Programmers in the meta-programming domain write specifications under the assumption of left-most innermost reduction. By doing this they use ASF as a first-order functional programming language with advanced pattern matching features.
- Practically all bugs in ASF specifications are caused by non-matching terms in conditions and therefore non-reducing terms that were supposed to be reduced. Debugging a specification amounts to carefully simplifying an input term to the smallest possible term that triggers a bug, then running the specification and locating the offending rewrite rule.
- Term rewriting can be implemented extremely efficiently and scales to big applications in meta-programming. Much of the efficiency is caused by maximally sharing sub-terms which allows small memory footprints and equality checking in $O(1)$ [11, 10]. This also implies immutability of data values at run-time.
- For large-scale meta-programming, which implies signatures of hundreds of constructor functions for the abstract syntax trees of programming languages, simple recursion over deep terms needs to be automated. Many meta-programs are "structure shy": they only apply to some node types of the abstract syntax and such nodes may be buried deep in a term. We have extended ASF with so-called traversal functions [8] to facilitate automatic type-safe traversal. This feature commonly reduces the size of an ASF application, sometimes up to 95% depending on the size of the language.
- Rule-based programming is not for everyone: it requires special training and experience to use effectively. Programmers with a formal computer science background have no trouble using ASF, while programmers without such background have difficulty adapting to this paradigm. They are surprised by the fact that simple things, like iteration over a list, may require two or more non-trivial rewrite rules or the use of a complicated list pattern, while other operations that are complex in a normal programming language are suddenly completely trivial. As a result, their previously acquired engineering skills seem useless.
- Text-book algorithms for static analysis and program optimization are not easily translated into the algebraic paradigm. Instead one must re-think these algorithms and visualize their effect in run-time and memory consumption as the execution platform executes them. This implies that to start analyzing and transforming programs, one must first re-think the basics. Although this is interesting from an academic perspective, from the software engineering perspective this represents a negative investment.

- Sets of rewrite rules and algebraic signatures are open for extension. One can add a new function to a certain sort and simply add alternatives for all the functions that process that sort. Example: we add a "do-while" construct to a language and then we add new rewrite rules for the extended definition of a control-flow graph extractor. This is called *open extensibility*: without changing existing code functionality can be extended in a modular fashion.

We have used ASF as a high-level programming language, applying it to different forms of meta-programming. We had to extend algebraic specification with default rules and traversal functions to obviate the need for large amounts of boilerplate rewrite rules. Unfortunately, as our student influx became less formally educated, we could not keep using ASF as a vehicle for education in software analysis and transformation.

2.2 Experience with ASF+SDF: the case of SDF

Since the original goal of ASF+SDF was describing programming languages, it includes a built-in facility for describing syntax: the Syntax Definition Formalism (SDF [19]). Naturally, this combination of parsing and rewriting makes ASF+SDF specifically apt for the domain of meta-programming. Parsing the input source code enables all further analysis and transformation. The essential characteristics of ASF+SDF derived from SDF are:

- Function signatures correspond to context-free grammar rules, similar in semantics to EBNF. For example, the *add* function is now declared as `NAT "+" NAT -> NAT`. Appropriate priorities and associativity can also be defined. A non-terminal is a sort, a grammar rule is a function. Furthermore, SDF supports *variable* definitions, a class of non-terminals specifically tagged to be meta-variables.
- In ASF+SDF equations we write concrete syntax patterns instead of prefix term patterns. Any production rule in a context-free grammar is a term constructor. The syntax of terms is completely user-defined.
- SDF integrates lexical and context-free syntax definitions to generate scannerless parsers [33]. This helps in broadening the scope of programming languages that can be accepted, for example to allow the analysis and transformation of languages that do not have a separate tokenizer or to allow parsing of embedded languages that have different sets of reserved keywords in different contexts.
- When modules are combined by way of import, the signatures that they declare are merged. In the case of SDF this means composition of complete grammars. Because only the full class of context-free grammars is closed under composition, SDF is supported by a parser which supports all context-free grammars, rather than a subset like LALR(1) or LL(1).
- List matching is implemented for regular expressions in SDF notation, e.g., terms of type `Statement*`, which corresponds to the language of possibly empty lists of statements, may be matched using arbitrary list patterns.

Using SDF as a front-end to ASF, the *add* example can now be written as follows:

```
[add1]  X + 0       = X
[add2]  X + succ(Y) = succ(X + Y)
```

Note both how the concrete syntax of the *add* function (+) can be used in the equations without quotation, and the use of `X` and `Y` as meta-variables ranging over the non-terminal for recognizing natural numbers.

Work on ASF+SDF has been documented in [17]. Writing and executing ASF+SDF specifications is supported by the ASF+SDF Meta-Environment [24, 7, 6]. The following list of observations summarize our experience in using ASF+SDF, this time focusing on the consequences of using SDF:

- Parsing and term rewriting are the only main features of ASF+SDF, and they are inseparable. This is both a strength and a weakness. The strength is conceptual simplicity and expressivity. The weakness is that one must understand both together and this makes the learning curve steep. New users struggle to conceptually separate parsing from rewriting when confronted with a bug or unexpected behavior.
- The upside of modular grammars is their unlimited composability. The downside is that no guarantee can be given as to whether the composed grammar is unambiguous. Solving ambiguities is difficult and requires expert knowledge [1].
- Since a signature is defined by a context-free grammar in ASF+SDF, any *type errors* — providing the wrong type of argument to a function — result in *parse errors*. This is rather uninformative, and can be especially confusing to new users.
- Some typical programming languages have non-context-free syntaxes. COBOL files for example have "margins" that are line based, while inside the margins the syntax is not line based. Consequently, it is impossible to parse such languages using just context-free grammars. Users of ASF+SDF have written preprocessors in scripting languages such as Perl and Python to remove margins or indentation and put them back later.
- The meta-programming paradigm requires "high fidelity" in rewriting source code to source code. Otherwise unimportant details such as whitespace and source code comments need to be retained in meta programs that transform existing software systems. To facilitate high fidelity source-to-source transformation, we have extended the ASF execution engine to rewrite full parse trees instead of abstract syntax trees.
- All data that is processed must first be specified as a context-free grammar. For example, the ASF+SDF to C compiler contains a grammar for a subset of ANSI C and of an intermediate pattern matching automaton. The COBOL control flow visualization tool contains a grammar of graphviz's dot formalism. The ASF+SDF library even contains several definitions of XML. Defining good grammars is hard work, but in ASF+SDF there is no way around it.
- Program analyses frequently require the representation of graphs, such as control flow graphs or data dependency graphs. These can be easily *encoded* as parse trees (which contain tree nodes and lists), but this representation induces a significant loss of efficiency. Furthermore, operations on sets, relations and graphs are encoded as traversals over lists, with similar loss of efficiency in computation.
- What you see is what you get. Since in ASF+SDF all data is a parse tree, simply unparsing it renders a complete and readable representation of all input, output and even intermediate data structures. This helps in making complex algorithms debuggable.

The combination of context-free grammars and term rewriting is very powerful, but is not without its drawbacks. All data is required to be described by a context-free grammar, hence parsing is a first and unavoidable step in creating any meta-programming tool. Input, output and intermediate representations are limited by context-free grammars: efficient representations of data that is more complex or simpler are not available.

2.3 Lessons learned from other formalisms

The strategic programming language Stratego [34] was motivated by similar experiences with ASF+SDF. Its design aims to keep the intention of rewrite rules as algebraic equalities, but on top of that introduces expressive rewrite strategies to compose them. In other words, Stratego extended algebraic programming with higher-order parameterized rule application. Stratego's strategies, which derived from the rewriting strategies in ELAN [4], are a true first-class programmable feature rather than a conservative extension of algebraic specifications. In Stratego, the strategies drive the computation, not the rewrite rules. We noticed that in most meta-programming applications of Stratego the strategies rather than the rewrite rules do the heavy lifting. This again emphasizes the programming rather than the specification features.

TXL [16] is a functional programming language intended to implement language extensions and software transformations. It has surprising similarities to ASF+SDF, but does not have its roots in algebraic specification. This is an eye opener. Although somewhat different, pattern matching, substitution, traversal, and BNF rules are all features of TXL, while it does not feature algebraic equations that are applied in a non-deterministic fashion. TXL is also known to be very successful in the area of meta-programming, so we may hypothesize that key components of algebraic programming are success factors in the meta-programming domain, rather than the whole integrated concept of algebraic specification.

Among other things, from ELAN [4] and Maude [14] we have learned how ACI matching (associative and commutative matching with defined identities) can be used to express sets and relations. Maude is known to be strong in analysis algorithms, such as model checking, while ASF+SDF is naturally stronger in transformation intensive applications. We have also experimented with a language called RScript [25], inspired by relational analysis tools such as Grok [23] and Crocopat [3], to verify that explicit set and relation operators would match the software analysis application domain in combination with fact extraction implemented directly in ASF+SDF.

Another source of inspiration is ANTLR [30]. Although ANTLR does not support all context-free grammars, it shines in its applicability and popularity among meta-programmers. This is caused by the perspective that meta-programs need to be included in a bigger software engineering context. ANTLR ensures by the design and implementation of the code it generates that ANTLR-based tools can be seamlessly integrated in ordinary software projects. A reason for this is that the code that is generated is almost what the programmer would write manually. ANTLR connects to the background of advanced software engineers rather than to the background of computer scientists. This is another eye-opener.

3 Rationale for the Design of RASCAL

RASCAL's goal is to cover the domain of meta-programming as a whole (see Figure 1). We now first enumerate which ASF+SDF features are desirable to keep and which to avoid. Then we present an overview of the features of the language. Based on our experience, we came to the conclusion that the following ASF+SDF features are desirable:

- Context-free grammars (and scannerless parsing) for the modular definition of the syntax of real programming languages.
- Pattern matching (and list matching), for finding patterns in programs.
- Pattern matching for dispatch over language constructs to obtain open extensibility.
- Default definitions to prevent boilerplate completion of alternatives.
- Automated traversal to support structure-shy applications.

- Concrete syntax for matching and construction of source code fragments.
- Immutability of data to facilitate efficient rewriting and for a safe programming environment.
- "what you see is what you get". Similar to the parse trees of ASF+SDF, all data should have a standard, complete and human-readable serialized representation. This notation should coincide exactly with the notation for expressions in RASCAL.

We also concluded that the following features are undesirable:

- Non-determinism in dispatch over language constructs. Meta-programs are mostly deterministic. So, simple rewrite rule semantics should be restricted.
- The necessity of defining a context-free grammar for every kind of data. We want to re-introduce abstract data-types as separate feature and have the possibility to compute directly with basic data-types such as strings, reals and integers.
- The paradigm of sets of rewrite rules is too exotic for many software engineers.
- The all-or-nothing experience of ASF+SDF. We need a language that can be introduced feature-by-feature, starting from a simple and understandable (procedural) basis.
- Type-checking by parsing is confusing.

We have now explained why RASCAL should be different. Now we explain how it is different. Starting from the aforementioned features of ASF+SDF we reorganized them into separate, independent layers and have added features if we considered them missing. These ingredients were then synthesized into the language design by an iterative design process, in which we reviewed a number of key use cases. These were static analysis algorithms, source-to-source transformations, type-checkers and source-code generators. The resulting language is RASCAL [27, 26][1].

RASCAL is organized in a core layer which contains basic data-types (booleans, integers, reals, source locations, date-time, lists, sets, maps, relations), structured control flow (if, while, switch, for) and exception handling (try, catch). To use the core you must understand that all data is immutable and that all code is statically typed. From this point of view, RASCAL looks like a simple general purpose programming language with built-in, immutable data structures.

The following is a list of features that can be learned on a "need to know" basis. The layers are progressively more domain specific to the meta-programming domain:

- List, set, and map comprehensions for the construction of SQL-like queries and analyses. This includes the `<-` element generation operator, which can enumerate the elements of all container data-types, like lists, sets, maps, and trees. The same operator is used in for loops.
- Algebraic data type definitions for the definition of (intermediate) abstract data-types. These are similar to the data type facilities in functional programming languages.
- Advanced pattern matching operators, like deep match (`/`), negative match (`!`), set matching and list matching. These can, for instance, be used in switch cases, for loops and comprehensions.
- String templates with margins and an auto-indent feature. The margins of strings allow one to indent a template with the nesting depth of the RASCAL program while embedding a multi-line source code template. This enhances the formatting of RASCAL template-based code generators.

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

- A `visit` statement, which is an extension of `switch` that traverses arbitrarily nested data in order to perform structure-shy analysis and transformation. Cases of a visit may substitute in place and/or have side effects on (local) variables by executing arbitrary RASCAL code. Visit is parameterized by a traversal strategy to allow different traversal orders.
- A `solve` statement for fixed-point computation.
- Syntax definitions using an EBNF-like notation for generating parsers. This includes disambiguation facilities.

Additionally, RASCAL is designed specifically to help the programmer create safe, modular and generic meta-programs in the following ways:

- Type inferencing for local variables in functions. Formal parameters and return types of functions must be explicitly typed. This prevents typing errors from leaking between function definitions.
- There is no down-cast operator. Instead all down conversions are done by matching. To use a variable that is bound by a match, the programmer must include the match in a conditional context, such as an if, for, switch, visit or comprehension to ensure that in the body of that construct the variables are bound. As a result, RASCAL programs have no "ClassCastException"-like run-time exceptions.
- Lexically scoped backtracking. Each body of a conditional statement or expression that uses non-deterministic pattern matching may use the `fail` statement to undo the effects of the current scope and jump to the next available match.
- The formal parameters of a function may also be arbitrary patterns, like the left-hand sides of rewrite rules. Each alternative for a certain function name must have mutually exclusive patterns. If this can not be realized, one of the alternatives must have the **`default`** modifier to indicate that it will be tried only after the other patterns fail. This gives us open extensibility: add a rule in a syntax definition or a data definition and you can add an alternative definition for any function that operates on that type.
- Rascal's sub-typing lattice supports a number of layers that allow algorithms to work on different levels of generality. This is complementary to having type parameters for generic functions and type-parameterized abstract data-types. The `value` type is the top type. Algorithms that do not assume anything about a value use this type. The `void` type is the bottom type. This is an example of a longest possible sub-type chain: `void < Statement < Tree < node < value`. In this chain `value`, `node` and `void` are built-in, while the others are defined in Rascal. The `node` type represents the common super-type of all abstract data-types, allowing access to and modification of the names and children of constructors. `Tree` is a library definition of an ADT for all parse trees and `Statement` is a defined non-terminal from a syntax definition for some programming language like C or Java. Functions may operate on each of these 5 levels, with the level chosen based on the amount of detail about the parameter that is needed. Another benefit is that the implementation of parse trees, which are central to meta programming, is completely transparent to the programmer.

To summarize, Rascal is a value-based procedural programming language with high-level domain specific built-in operators. These operators come from algebraic programming and relational calculus. We realize that this is not a formal definition nor a full explanation of the language. It should, however, be a good starting point for getting an impression of RASCAL and its design rationale. Details of Rascal may change as we get more feedback, and it may be extended, but this design will not change.

Example Getting back to our running example, there are many ways in which the *add* example can be rewritten in RASCAL[2]. One is to use a switch statement for case distinction:[3]

```
NAT add1(NAT x, NAT y) {
  switch (y) {
   case z(): return x;
   case succ(NAT y): return succ(add1(x, y));
  }
}
```

This is a traditional (mostly imperative) programming style which is fully supported. Another way of writing this same example in RASCAL is to write a function for each case. Note that RASCAL generalizes the notion of a function signature from a list of typed variables to a list of patterns that may contain (possibly deeply nested) variables. Pattern-matching at the call site determines which version of the function is actually called (*pattern-directed invocation*). The two functions for defining *add* are then written as:

```
NAT add2(NAT x, z()) { return x; }
NAT add2(NAT x, succ(NAT y)) { return succ(add2(x , y)); }
```

We can approach algebraic equations even further, since functions that return a single expression can be abbreviated as follows:

```
NAT add2(NAT x, z()) = x;
NAT add2(NAT x, succ(NAT y)) = succ(add2(x , y));
```

This example illustrates that one can stay close to algebraic specifications, but that mixtures of algebraic style and imperative style are supported as well. This is very convenient when mixing, for instance, axiom-based simplification rules with more imperative symbol table handling.

RASCAL provides lists (with associative matching) and sets (with associative and commutative matching) further strengthening the algebraic flavor. Although RASCAL remains true to its algebraic roots, the overall feeling of the language is that of a programming language rather than a specification language. This is not only because we opted for a Java-like notation, but also because we have packaged concepts differently and have introduced some non-algebraic concepts like, most notably, global and local variables, comprehensions, and standard control flow. A very simple example can illustrate this. We define binary trees with integers as leaves and composite nodes that specify the color of the node and two subtrees. This can be defined as follows:

```
data ColoredTree = leaf(int n)
                 | composite(str color, ColoredTree left, ColoredTree right);
```

Next we want to analyze a `ColoredTree` and compute a frequency distribution of the colors used in composite nodes. We use a map from integers to strings to maintain the frequencies and a local variable `counts` to maintain this map. The automatically inferred type of `counts` is `map[str, int]`. A visit statement is used that traverses an arbitrary data structure, matches the patterns for the cases to all substructures, and executes the case when a match is found. The statement `counts[color]?0 += 1` increments the current frequency count for the given color if it exists or it increments 0 otherwise. Note how this affects the value of the local variable `counts`. The RASCAL code is shown in Listing 1.

[2]This example is for comparison only and is a-typical since, unlike ASF+SDF, RASCAL has built-in, arbitrary length, integers and reals.

[3]We use the constructor `z()` to represent `0`.

Listing 1 Counting frequencies of colors in a ColoredTree

```
public map[str, int] colorDistribution(ColoredTree t) {
   counts = (); // initialize an empty map
   visit(t) {   // all leaves and composite nodes in the tree
     case composite(str color, _, _):
                // for each composite node: increment count for color
                // (use 0 as default when not yet in table)
                counts[color] ? 0 += 1;
   }
   return counts;
}
```

4 Applications

We describe three realistic applications of RASCAL to model-driven software development here and one example of connecting RASCAL to an existing executable specification.

- A DSL for Entity modeling (Section 4.1). This educational example is based on our submission to the Language Workbench Competition 2011[4] and illustrates modularity, syntax definition, AST types and code generation.
- ECore [12] (Section 4.2) is a well-known class-based meta-model used in many Eclipse-based modeling tools such as Kermeta, ATL and XText. This example shows an encoding of ECore using RASCAL types, in particular the use of relations for DAG-like and cyclic structures.
- DERRIC (Section 4.3) 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.
- RLS-RUNNER (Section 4.4) is a library plug-in for RASCAL that enables the execution of existing Maude specifications using a combination of higher-order functions and a co-routine implemented using pipes. This enables us to reuse existing program analysis specifications instead of requiring them to be rewritten in RASCAL.

4.1 A simple DSL: Entities

4.1.1 Concrete and abstract syntax

Listing 2 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 in order to model business objects. An entity has named fields, which are either primitively typed (integer, string, boolean), or contain a reference to

[4] http://www.languageworkbenches.net

another entity. An excerpt of the syntax definition of entity models is shown in Listing 2. 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` non-terminal is the start symbol of the Entities grammar as indicated by the `start` keyword.

Listing 3 Abstract syntax of entities

```
data Entities = entities(list[Entity] entities);
data Entity = entity(Name name, list[Field] fields);
data Field = field(Type \type, str name);
data Type = primitive(PrimitiveType primitive) | reference(Name name);
data Name = name(str name);
data PrimitiveType = string() | date() | integer() | boolean() | currency();
```

Whereas ASF+SDF allowed only rewriting of concrete syntax trees, RASCAL supports the automatic mapping of parse trees to ASTs, using the library function `implode`. This function converts a parse tree to an AST that conforms to a RASCAL ADT describing the abstract syntax. Listing 3 shows an ADT describing the abstract syntax of Entity models. For every syntax production in the grammar for entities, there is a corresponding constructor in this ADT. Every constructor has the same number of arguments as the number of symbols in the production (modulo keywords and layout). Lexical tokens are mapped to RASCAL primitive types.

Transformation of concrete syntax trees is useful in cases where layout preservation is essential, such as refactoring or legacy renovation. However, for MDE applications this is often less important. As already discussed earlier, RASCAL addresses this issue by providing light weight string templates, next to full-blown source-to-source transformations. Below we describe a simple, template-based, Java code generator for entity models.

4.1.2 Java code generation

The code generator is shown in Listing 4. It is defined using ordinary RASCAL functions that produce string values using RASCAL's built-in string templates. As an example, consider 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` 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 with 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. Again, 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, as introduced in Section 3. The `field2java` function is implemented in a style reminiscent of term rewriting. The `field2java` function thus matches its first parameter against the pattern `field(typ, n)`. 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.

Listing 4 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>;
           '}";
}
```

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.

4.1.3 IDE support

No language can do without IDE support, and this includes DSLs. RASCAL exposes hooks into the Eclipse-based IMP [13] 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 2. The generated IDE runs within the RASCAL Eclipse IDE, so the package explorer on the left actually shows the source code of the implementation of the Entities DSL. In the middle you see an editor containing a simple entity model. It has syntax highlighting and folding which are both based on the context-free grammar. 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.

4.2 Relational meta-modeling: ECore

Algebraic data types generally do not support expressing structures with sharing and/or cycles. Nevertheless, in MDE such graph-like structures, especially class-based models, are very common. RASCAL is a functional programming language in that all the data is immutable. To deal with graph structures (e.g., control-flow graphs, call-graphs, automata, work-flow models etc.) RASCAL provides *relations*. Relations are basically sets of tuples which can be queried using comprehensions. Additionally, RASCAL provides built-in support for computing the transitive closure of a binary relation.

As an example of how one might encode a class-based model as a RASCAL data type, Listing 5 shows an excerpt of the ECore meta meta-model. We have used this ADT to successfully import around 300 ECore models. The top-level constructor `ecore` contains a set of `Classifiers`, a subtype relation between `Classifiers` and a typing relation from `Items` (scoped in `Classifiers`) to `Types`. The last two

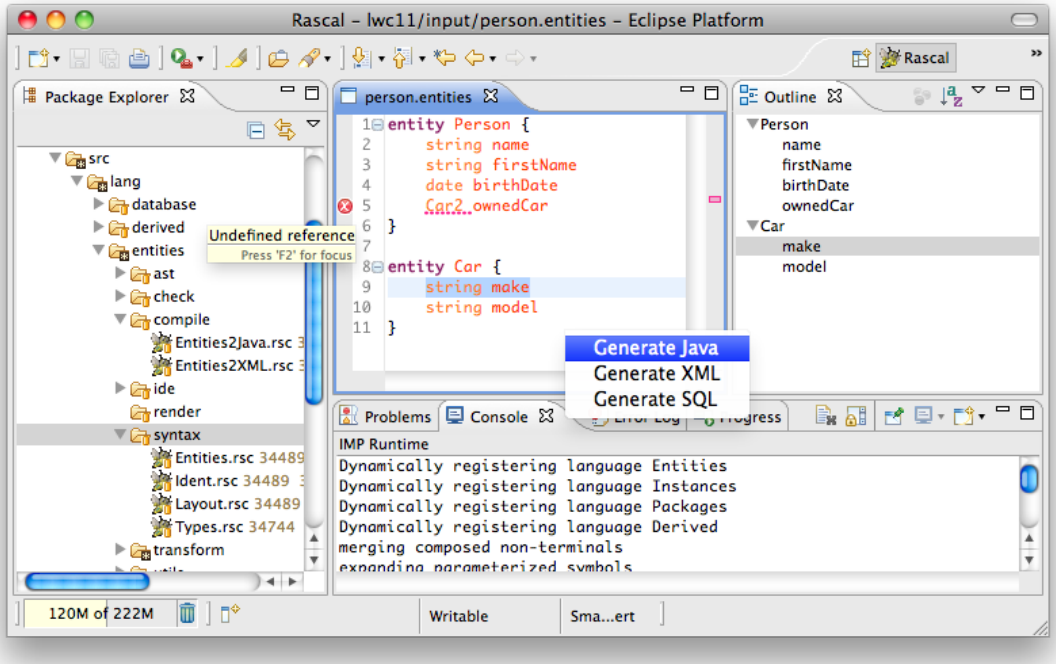

Figure 2: Screen-shot of the dynamically generated IDE for the Entities DSL

constructor arguments capture the essential sharing and/or cyclicity that may be present in a class model. Since classifiers are identified by a package qualification (`Package`, not shown) and a name, such values can be used as indices into the `subtype` and `typing` relations.

For instance, assume we have variable `class` containing a `Class` value representing a Person class. One could then find all super classes of this class using the transitive closure of the `subtype` relation of an ECore model `e`:

```
class = concrete("Person", [attribute("name", {}, string())]);

for (sup <- e.subtype+[class])

  print("Super: <sup>");
```

The encoding shown here is non-trivial and it makes a number of trade-offs and short-cuts with respect to accurate typing of class models. For instance, the type of the `subtype` relation allows primitive types to be sub- and super-types because they are in fact classifiers; technically this is incorrect. Nevertheless, making the encoding more strict would also introduce more indirections and hence, introduce more case distinctions when processing ECore models.

Another observation is that the encoding is very convenient for querying, but less than optimal for transformation. Model transformation of Ecore models encoded this way would entail creating new `ecore` values every time a single element is changed, at arbitrary depth in the constructor. Apart from not being very efficient, a problem with this approach is the lack of locality: every transformation, no matter how small and localized, has to have knowledge of the complete value. We consider this to be an area of further research.

Listing 5 ADT for ECore (excerpt)

```
data ECore = ecore(set[Classifier] classifiers,
                   rel[Classifier, Classifier] subtype,
                   rel[Classifier, Item, Type] typing);
data Classifier = dataType(Package package, str name) // Package omitted
                | class(Package package, Class class);
data Class = concrete(str name, list[Item] items)
           | interface(str name, list[Item] items)
           | abstract(str name, list[Item] items);
data Item = operation(str name, set[Option] options)
          | parameter(str operation, str name, set[Option] options)
          | attribute(str name, set[Option] options, Type dataType)
          | reference(str name, set[Option] options);
data Type = classifier(Classifier classifier);
```

4.3 A model-based approach to 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. 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.

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 4). 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 shown in Figure 4. 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` on lines 4 and 5 in Figure 3).

Another refinement consists of simplifying the AST by performing compiler optimizations such as constant folding and propagation (e.g., replacing string constants such as on line 6 in Figure 3 by a list of bytes corresponding to the string in the defined encoding, so that the generated code only has to do simple byte comparisons).

```
1 structures
2 APP0JFIF {
3   marker: 0xFF, 0xE0;
4   length: lengthOf(rgb) + (offset(rgb)
5           - offset(identifier)) size 2;
6   identifier: "JFIF", 0;
7   version: expected 1, 2;
8   units: 0 | 1 | 2;
9   xthumbnail: size 1;
10  ythumbnail: size 1;
11  rgb: size xthumbnail * ythumbnail * 3;
12 }
13
14 DHT {
15   marker: 0xFF, 0xC4;
16   length: size 2;
17   data: size length - lengthOf(marker);
18 }
19
20 SOS = DHT {
21   marker: 0xFF, 0xDA;
22   compressedData:
23     unknown terminatedBefore 0xFF, !0x00;
24 }
```

Figure 3: Excerpt of JPEG in DERRIC

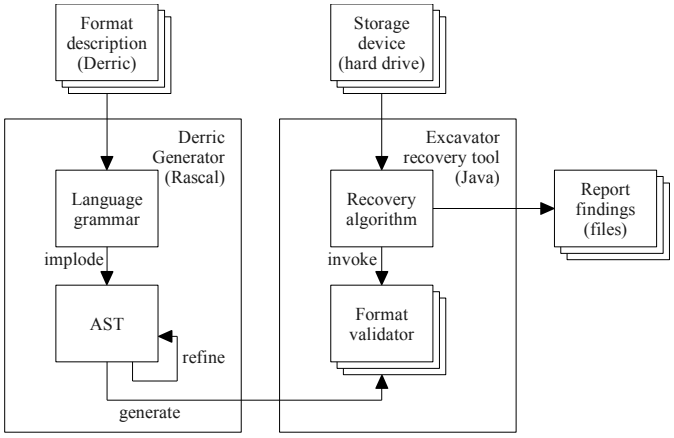

Figure 4: Use of DERRIC in EXCAVATOR

Component	Lang	Size (SLOC) (total **1871**)
Grammar	RASCAL	57
JPEG def	DERRIC	58
PNG def	DERRIC	89
GIF def	DERRIC	101
Code generator	RASCAL	510
Recovery Code	Java	316
Base library	Java	740

Figure 5: Sizes of the EXCAVATOR components

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 4 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.

To evaluate EXCAVATOR, we have compared it to three industrial-strength carving tools on a set of standard benchmarks [5]. Our evaluation shows that even though the implementation is very small (see Figure 5), 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.

4.4 RASCAL front-ends for K program analysis semantics

The K [32] semantics framework is an executable framework for defining the semantics of programming languages. Semantics for program analysis can be defined similarly to those for standard evaluation, with rules evaluating program constructs over abstract value domains. Work in this area includes matching logic [31], a logic with similar goals to separation logic [29]; and analysis policies, where a policy is defined as a combination of a generic analysis semantics for a language, specific extensions for each

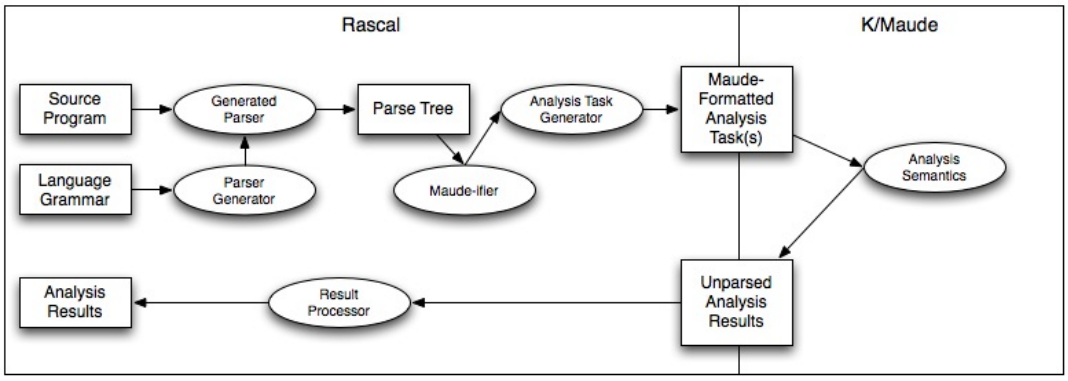

Figure 6: Integrating RASCAL with K Definitions in Maude

specific analysis, and an annotation language tailored to each analysis [21].

One limitation of this work is that it has focused on the semantics, but not on the entire tool chain. This means that the process of transforming the input program into something that can be evaluated in a K semantics, or of taking the results and providing them to the user of the analysis in some useful form, has always been approached in an ad-hoc fashion. While not a theoretical problem, this makes the analyses much less useful in practice.

The RLSRunner tool [20] provides a solution to this for K definitions compiled to run in Maude [15], a language and engine for defining, evaluating, and reasoning about rewriting logic [28] specifications. An overview of the RLSRunner integration with Maude is shown in Figure 6. First, using RASCAL, one defines the grammar for the language being analyzed, which is used to generate a parser for the language. As shown earlier, this automatically provides for a basic IDE for the language. A *maudeifier* is then defined, allowing the parse tree of the program to be analyzed to be transformed into a prefix form easily consumable by Maude. This prefix form also includes location information, encoded using a K definition for locations, which can be used to tag errors with the location of the offending construct. RLSRunner library functions are then used to register handlers both for preparing the term to be given to Maude and for parsing the term resulting from evaluation. Other RLSRunner functions allow RASCAL to interact with Maude and with the Eclipse environment, allowing error information returned as a result of the analysis to be shown in the IDE and in the Problems view. An example of showing this information in Eclipse is provided in Figure 7 for a units of measurement analysis in a simple imperative language.

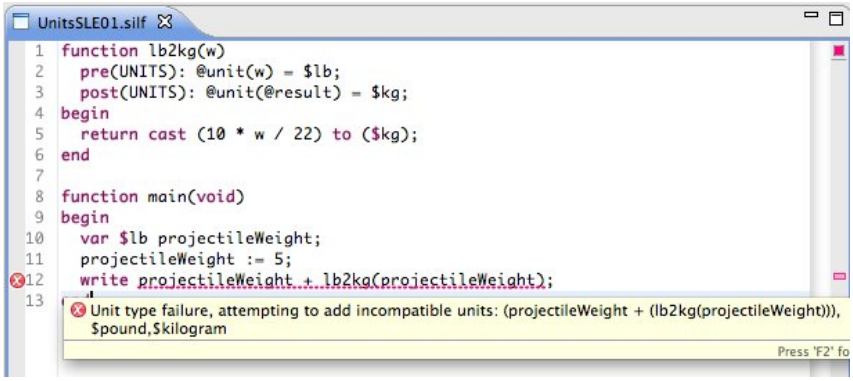

Figure 7: Units Arithmetic Error, Shown in Eclipse

5 Concluding Remarks

We have presented the lessons we learned going from algebraic *specification* languages to algebraic *programming* languages. These lessons were input for the design rationale for the RASCAL language: a domain-specific programming language for meta-programming. RASCAL is easy to teach, learn and use in the domain of meta programming, but is still true to its algebraic specification roots.

References

[1] H.J.S. Basten (2010): *Tracking Down the Origins of Ambiguity in Context-Free Grammars*. In: *Proceedings of ICTAC'10*, *LNCS* 6255, Springer-Verlag, pp. 76–90, doi:10.1007/978-3-642-14808-8_6.

[2] J.A. Bergstra, J. Heering & P. Klint, editors (1989): *Algebraic Specification*. ACM Press/Addison-Wesley.

[3] D. Beyer (2006): *Relational programming with CrocoPat*. In: *Proceedings of ICSE'06*, ACM Press, pp. 807–810, doi:10.1145/1134285.1134420.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau & C. Ringeissen (1998): *An Overview of ELAN*. In: *Proceedings of WRLA'98*, *ENTCS* 15, Elsevier, doi:10.1016/S1571-0661(05)82552-6.

[5] J. van den Bos & T. van der Storm (2011): *Bringing Domain-Specific Languages to Digital Forensics*. In: *Proceedings of ICSE'11*, ACM Press, pp. 671–680, doi:10.1145/1985793.1985887.

[6] M.G.J. van den Brand, M. Bruntink, G.R. Economopoulos, H.A. de Jong, P. Klint, T. Kooiker, T. van der Storm & J.J. Vinju (2007): *Using The Meta-Environment for Maintenance and Renovation*. In: *Proceedings of CSMR'07*, IEEE, pp. 331–332, doi:10.1109/CSMR.2007.52.

[7] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser & J. Visser (2001): *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment*. In: *Proceedings of CC '01*, *LNCS* 2027, Springer-Verlag, pp. 365–370, doi:10.1007/3-540-45306-7_26.

[8] M.G.J. van den Brand, P. Klint & J.J. Vinju (2003): *Term rewriting with traversal functions*. *ACM Transactions on Software Engineering and Methodology* 12(2), pp. 152–190, doi:10.1145/941566.941568.

[9] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener & E.A. van der Meulen (1996): *Industrial Applications of ASF+SDF*. In: *Proceedings of AMAST'96*, *LNCS* 1101, Springer-Verlag, pp. 9–18, doi:10.1007/BFb0014303.

[10] M.G.J. van den Brand, J. Heering, P. Klint & P.A. Olivier (2002): *Compiling language definitions: The ASF+SDF compiler*. *ACM Transactions on Programming Languages and Systems* 24(4), pp. 334–368, doi:10.1145/567097.567099.

[11] M.G.J. van den Brand, H.A. de Jong, P. Klint & P.A. Olivier (2000): *Efficient Annotated Terms*. *Software, Practice & Experience* 30(3), pp. 259–291, doi:10.1002/(SICI)1097-024X(200003)30:3%3C259::AID-SPE298%3E3.0.CO;2-Y.

[12] F. Budinsky, S.A. Brodsky & E. Merks (2003): *Eclipse Modeling Framework*. Pearson Education.

[13] Ph. Charles, R.M. Fuhrer, S.M. Sutton, Jr., E. Duesterwald & J. Vinju (2009): *Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse*. In: *Proceedings of OOPSLA'09*, ACM Press, pp. 191–206, doi:10.1145/1640089.1640104.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & J. F. Quesada (2002): *Maude: Specification and Programming in Rewriting Logic*. *Theoretical Computer Science* 285(2), pp. 187–243, doi:10.1016/S0304-3975(01)00359-0.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C.L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *LNCS* 4350, Springer-Verlag, doi:10.1007/978-3-540-71999-1.

[16] J.R. Cordy (2011): *Excerpts from the TXL cookbook*. In: *Post-Proceedings of GTTSE'09*, *LNCS* 6491, Springer-Verlag, pp. 27–91, doi:10.1007/978-3-642-18023-1_2.

[17] A. van Deursen, J. Heering & P. Klint, editors (1996): *Language Prototyping: An Algebraic Specification Approach*. *AMAST Series in Computing* 5, World Scientific.

[18] J. Goguen (1979): *Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs*. In: *Mathematical Studies of Information Processing*, *LNCS* 75, pp. 425–473, doi:10.1007/3-540-09541-1_36.

[19] J. Heering, P.R.H. Hendriks, P. Klint & J. Rekers (1989): *The syntax definition formalism SDF - reference manual*. *SIGPLAN Notices* 24(11), pp. 43–75, doi:10.1145/71605.71607.

[20] M. Hills, P. Klint & J.J. Vinju (2011): *RLSRunner: Linking Rascal with K for Program Analysis*. In: *Proceedings of SLE'11*, LNCS, Springer-Verlag. To Appear.

[21] M. Hills & G. Roşu (2010): *A Rewriting Logic Semantics Approach To Modular Program Analysis*. In: *Proceedings of RTA'10*, *Leibniz International Proceedings in Informatics* 6, Schloss Dagstuhl - Leibniz Center of Informatics, pp. 151 – 160, doi:10.4230/LIPIcs.RTA.2010.151.

[22] C.M. Hoffmann & M.J. O'Donnell (1982): *Programming with Equations*. *ACM Transactions on Programming Languages and Systems* 4(1), pp. 83–112, doi:10.1145/357153.357158.

[23] R.C. Holt (2008): *Grokking Software Architecture*. In: *Proceedings of WCRE'08*, IEEE, pp. 5–14, doi:10.1109/WCRE.2008.34.

[24] P. Klint (1993): *A Meta-Environment for Generating Programming Environments*. *ACM Transactions on Software Engineering and Methodology* 2(2), pp. 176–201, doi:10.1145/151257.151260.

[25] P. Klint (2008): *Using Rscript for Software Analysis*. In: *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*.

[26] P. Klint, T. van der Storm & J.J. Vinju (2009): *RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation*. In: *Proceedings of SCAM'09*, IEEE, pp. 168–177, doi:10.1109/SCAM.2009.28.

[27] P. Klint, T. van der Storm & J.J. Vinju (2011): *EASY Meta-programming with Rascal*. In: *Post-Proceedings of GTTSE'09*, *LNCS* 6491, Springer-Verlag, pp. 222–289, doi:10.1007/978-3-642-18023-1_6.

[28] J. Meseguer (1992): *Conditional rewriting logic as a unified model of concurrency*. *Theoretical Computer Science* 96(1), pp. 73–155, doi:10.1016/0304-3975(92)90182-F.

[29] P.W. O'Hearn, J.C. Reynolds & H. Yang (2001): *Local Reasoning about Programs that Alter Data Structures*. In: *Proceedings of CSL'01*, *LNCS* 2142, Springer-Verlag, pp. 1–19, doi:10.1007/3-540-44802-0_1.

[30] T. Parr & K.S. Fisher (2011): *LL(*): The Foundation of the ANTLR Parser Generator*. In: *Proceedings of PLDI'11*, ACM Press, pp. 425–436, doi:10.1145/1993498.1993548.

[31] G. Roşu, C. Ellison & W. Schulte (2011): *Matching Logic: An Alternative to Hoare/Floyd Logic*. In: *Proceedings of AMAST'10*, *LNCS* 6486, Springer-Verlag, pp. 142–162, doi:10.1007/978-3-642-17796-5_9.

[32] G. Roşu & T.F. Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[33] E. Visser (1997): *Syntax Definition for Language Prototyping*. Ph.D. thesis, University of Amsterdam.

[34] E. Visser (2004): *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9*. In: *Domain-Specific Program Generation*, *LNCS* 3016, Spinger-Verlag, pp. 216–238, doi:10.1007/978-3-540-25935-0_13.