RLSRunner: Linking Rascal with K for Program Analysis

Mark Hills[1,2], Paul Klint[1,2], and Jurgen J. Vinju[1,2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

Abstract. The Rascal meta-programming language provides a number of features supporting the development of program analysis tools. However, sometimes the analysis to be developed is already implemented by another system. In this case, Rascal can provide a useful front-end for this system, handling the parsing of the input program, any transformation (if needed) of this program into individual analysis tasks, and the display of the results generated by the analysis. In this paper we describe a tool, RLSRunner, which provides this integration with static analysis tools defined using the K framework, a rewriting-based framework for defining the semantics of programming languages.

1 Introduction

The Rascal meta-programming language [13,12] provides a number of features supporting the development of program analysis tools. This includes support for processing the input program with a generalized parser and pattern matching over concrete syntax; developing the code for the analysis using flexible built-in types (e.g., sets, relations, lists, and tuples), pattern matching, user-defined algebraic data types, and higher-order functions; and displaying the analysis results interactively to the user through visualization libraries and through integration with the Eclipse IDE via IMP [4,5].

However, sometimes the analysis to be developed already exists, and there may be compelling reasons to use this analysis instead of rewriting it in Rascal. The existing analysis may be trusted, very complicated, or highly optimized, or may provide features not already available in Rascal. In these cases, instead of requiring the user to rewrite the analysis, Rascal can be used for its front-end capabilities (parsing, concrete syntax matching, Eclipse integration) while handing off analysis tasks to existing analysis tools.

This paper describes a tool, RLSRunner, which provides this integration with program analyses written using the K framework [10,16]. K is a rewriting-based, tool-supported notation for defining the semantics of programming languages. It is based on techniques developed as part of the rewriting logic semantics (RLS) project [15,14]. RLSRunner links languages defined in Rascal with formal language definitions written either directly in K and compiled to Maude [6], or with K-style definitions written directly in Maude.

Contributions. The direct contribution described here is the RLSRunner tool itself, providing a method to take advantage of K language definitions while building program analysis tools using Rascal. Indirectly, we believe that RLSRunner highlights the

flexibility of Rascal, showing that there is no requirement for Rascal-only solutions. It also mitigates one of the limitations of the current K suite of tools, which are focused on abstract syntax-based program representations but provide only limited support for working with concrete syntax.

Roadmap. In Section 2 we describe the Rascal support created for interacting with external tools (in general) and K definitions in Maude (in particular). We then describe extensions to K language definitions needed to support interaction with Rascal in Section 3. This is put together in a case study in Section 4, presenting the pairing of Rascal with an existing analysis framework used to search for type and unit errors in programs written in a paradigmatic imperative language. Section 5 then briefly discusses related work, while Section 6 presents some thoughts for future work and concludes.

2 Supporting K Program Analysis in Rascal

Figure 1 provides an overview of the process of integrating languages defined in Rascal with program analysis semantics defined in K. The first step in this integration is to define the front-end components: a grammar for the language; a generated parser based on this grammar; and a program (the "Maude-ifier") that will take the generated parse tree for a program and transform it into a Maude-readable form. Section 4 provides a concrete example of this process. As a side-effect, defining the grammar also provides an IDE for the language, which can be further extended with additional features as needed.

Given the Maude-ified code, the Rascal RLSRunner library then provides the functionality, in conjunction with language-specific user code, to: generate the analysis tasks; send them to Maude; read the results; and parse these results to extract the overall results of the analysis. The main driver function for this process is the `runRLSTask` function, shown in Figure 2. `runRLSTask` provides the functionality needed to start, communicate with, and stop Maude. To do this it makes use of the ShellExec library. ShellExec includes a number of routines to start, read from, write to, and terminate processes running outside Rascal.

The execution of `runRLSTask` is customized for specific languages and analysis tasks using an `RLSRunner` value (in this case, `RLSRunner` is the name of a Rascal

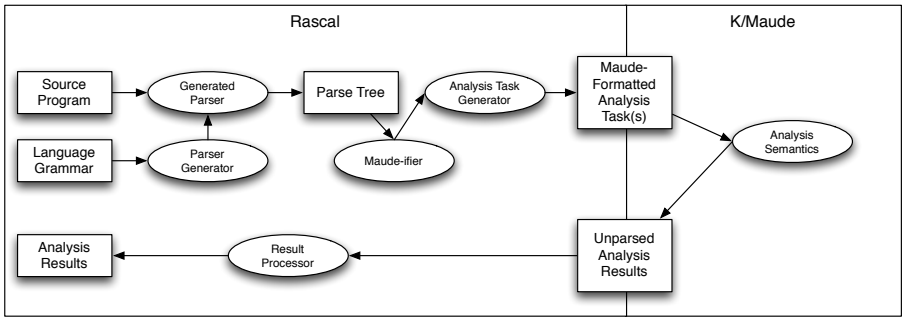

Fig. 1. Integrating Rascal and K

```
public RLSResult runRLSTask(loc ml, RLSRunner runner,
                            str input...)
{
  PID pid = startMaude(ml,runner.maudeFile);

  str inputStr = input[0];
  list[str] inputArgs = [ ];
  if (size(input) > 1)
    inputArgs = [ input[n] | n <- index(input)-0 ];

  str toRun = (runner.pre)(inputStr,inputArgs);

  writeTo(pid, toRun);
  str res = readFrom(pid);
  bool continueReading = true;
  while (continueReading) {
    if (/rewrites:\s*\d+/ !:= res && /Maude\>\s+$/ !:= res)
      res = res + readFrom(pid);
    else
      continueReading = false;
  }

  RLSResult rlsRes = (runner.post)(res);
  stopMaude(pid);
  return rlsRes;
}
```

Fig. 2. The `runRLSTask` Function, in Rascal

algebraic data type, not the library) passed as a parameter (`runner` in Figure 2). The `RLSRunner` includes the Maude definition of the K semantics to use for the analysis, and also includes pre- and post-processing functions, referred to in Figure 1 as "Analysis Task Generator" and "Result Processor", respectively (and in the code as `pre` and `post`). The pre-processing function takes the Maude-ified term, passed to `runRLSTask` as the first element of the `inputs` parameter, and pre-processes it. This generates the analysis task, or tasks, in the form of a Maude term (named `toRun` in the code). This term is written to the Maude process's input stream, where it will then be rewritten by Maude using the analysis semantics to perform the analysis. When Maude writes the final result of the analysis, this is read by `runRLSTask` on the process output stream.

Once the entire result is read, `runRLSTask` invokes the post-processing function. This function checks to see if the result format is one it can parse; if so, it returns the analysis results using a user-defined, analysis-specific constructor that extends type `RLSResult`. Two output formats are currently supported by the RLSRunner library, while more can be added as needed. Both default formats include performance information from Maude: total rewrites, rewrites per second, etc. In the first format, the result of the analysis is provided just as a string containing all the generated output (for instance, all error messages). This string is processed by the result processor (see Figure 1) to extract the analysis results. In the second format, error information is returned using a delimited format, made up of the location of the error, the severity of the error, and the

error message. In conjunction with the `createMessages` function in the RLSRunner library, the result processor can extract the location, severity, and error message information, returning the analysis results as a set of items of Rascal type `Message`:

```
data Message = error(str msg, loc at)
             | warning(str msg, loc at)
             | info(str msg, loc at);
```

Using the `addMessageMarkers` function in the Rascal ResourceMarkers library, these messages can be added in Eclipse as `Problem` markers to the indicated locations, with a severity (`Error`, `Warning`, or `Info`) based on the results; these markers appear both in the editor (as red "squiggly" underlines for errors, for instance) and in the `Problems` view. Location information from the parse tree, passed in as part of the analysis task to Maude, is used in the messages to ensure that markers are added to the correct locations. Section 4 shows examples of this output marking in action.

If, instead, the post-processor cannot parse the result, it returns all the output from the analysis using a predefined `RLSResult` named `NoResultHandler`, indicating that some error occurred. Then, after any results (parsable or not) have been computed by `post`, `runRLSTask` stops the Maude process and returns the `RLSResult` item.

3 Rascal Support in K

To work with Rascal, K definitions need to meet two requirements. First, the output of a run of the semantics should be in a format parsable by the RLSRunner library. Second, the semantics should support Rascal source locations, allowing messages generated by the analysis to be tied back to specific source locations in the file being analyzed.

To support Rascal source locations in K semantics, an algebraic definition of Rascal locations is provided as an operator `sl` (for source location) that defines a value of sort `RLocation`. `sl` keeps track of the URI as well as the position information: offset, length, and starting and ending rows and columns (set to -1 if not available). `sl` is defined (using Maude notation) as follows:

```
fmod RASCAL-LOCATION is
  including STRING .
  including INT .
  sort RLocation .
  op sl : String Int Int Int Int Int Int -> RLocation .
endfm
```

These locations are then used by extending the abstract syntax for various language constructs, such as declarations, expressions, etc, adding new "located" versions of these constructs. For each construct `C` that we wish to extend, we define a new operator written as `locatedC`, taking both a `C` and an `RLocation`. For instance, assuming we have a sort for declarations named `Decl`, the located version is defined in Maude as:

```
op locatedDecl : Decl RLocation -> Decl .
```

To use these in the semantics, a new K cell, `currLoc`, is defined, holding the current source location. This cell is updated to the location given in a located construct when one is encountered during evaluation:

```
op currLoc : RLocation -> State .
eq k(decl(locatedDecl(D, RL)) -> K) currLoc(RL') =
   k(decl(D) -> rloc(RL') -> K) currLoc(RL) .
```

The equation shown above does the following: if we are in a state where a located declaration `D`, with source location `RL`, is the next evaluation step, and `RL'` is the current location, we change the current location to `RL`. We also set the next two computational steps, first evaluating the declaration `D` using whatever logic was already present, and then resetting the location back to `RL'`. This processes the declaration in the context of the source location given with the declaration, and then recovers the current location in case it is needed. The rule to handle `rloc`, which recovers a source location (while discarding the current location in the `currLoc` cell), is shown below:

```
op rloc : RLocation -> ComputationItem .
eq k(rloc(RL) -> K) currLoc(RL') = k(K) currLoc(RL) .
```

The need for, in essence, creating a location stack can be seen with constructs such as loops or conditionals, where the correctness of the (located) construct may depend on the correctness of (located) children. Without a mechanism to recover prior locations, the error message would instead have to be given in terms of the most recent location, which may not be the correct one.

4 Linking Rascal with the SILF Analysis Framework

SILF [10], the Simple Imperative Language with Functions, is a standard imperative language with functions, arrays, and global variables. Originally designed as a dynamically typed language, it has since been adapted to support experiments in defining program analysis frameworks, including type systems, using an abstract K semantics, with analysis information given in programs using function contracts and type annotations [11].

As mentioned in Section 2, the first step in linking Rascal with a K-defined analysis tool is to define the front-end components: a grammar for the language, the generated parser, and the Maude-ifier. As a running example, two rules in the grammar for SILF, defining the productions for addition (`Plus`) and for a function call (`CallExp`), are given as follows (`{Exp ","}*` represents a comma-separated list of expressions):

```
Exp = Plus: Exp "+" Exp
    | CallExp: Ident "(" {Exp ","}* ")";
```

In Maude, the abstract syntax is defined by defining algebraic operators, with each _ character representing a placeholder for a value of the given sort. The Maude versions of the two productions given above are shown below, with the first operator defining `Plus` and the second and third defining `CallExp`, one with parameters and one without:

```
op _+_ : Exp Exp -> Exp .
op _`(_`) : Id ExpList -> Exp .
op _`(`) : Id -> Exp .
```

The string generated by the Maude-ifier, created with Rascal code like the following, uses a prefix form of these operators, for instance using _+_ for the plus operation with the operands following in parens:

```
case (Exp)`<Exp el> + <Exp er>` :
  return located(exp,"Exp","_+_(<toMaude(el)>,<toMaude(er)>)");
```

Function `located` then builds the located version of the construct, discussed in Section 3, using the location associated with `exp` (the subtree representing the plus expression) to generate the correct values for the `sl` operator. An example of the Maude generated for the expression `1 + 2` is shown below (with the file URI replaced by `t.silf` for conciseness, and with the expression located on line 13 between columns 7 and 12):

```
locatedExp(_+_(
    locatedExp(#(1),sl("t.silf",166,1,13,7,13,8)),
    locatedExp(#(2),sl("t.silf",170,1,13,11,13,12))),
  sl("t.silf",166,5,13,7,13,12)))
```

Once we have the Maude-ified program with embedded location information, we can use the RLSRunner library, as described in Section 2, to run the analysis tasks generated for the language. In this case the pre-processing function is very simple, inserting the Maude-ified version of the program into a use of `eval` (with `nil` indicating that no input parameters are provided – they are only used in executions of the dynamic semantics). `red` is the Maude command to reduce a term, i.e., to perform the analysis:

```
public str preCheckSILF(str pgm, list[str] params) {
  return "red eval((<pgm>),nil) .\n";
}
```

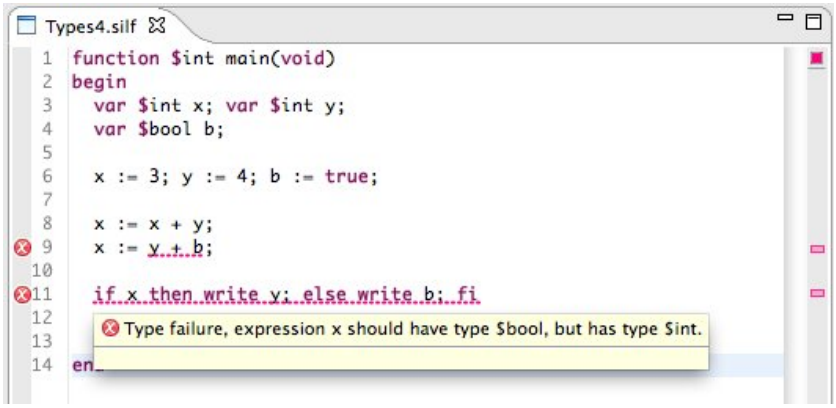

Fig. 3. Type Error Markers in SILF

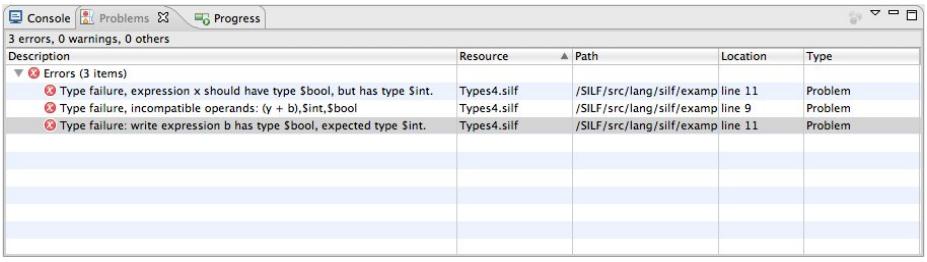

Fig. 4. Problems View with SILF Type Errors

The results are handled by the post-processing function in conjunction with functions in the RLSRunner library, extracting the analysis results as Rascal `Messages` as discussed in Section 3.

Two types of analysis are currently supported. The first is a standard type checking policy, with types provided using type annotations such as `$int`. Figure 3 provides an example of the results of a run of this analysis on a program with several type errors. As can be seen in the Eclipse Problems view, shown for the same file in Figure 4, there are three errors in this file. The first marked location in Figure 3 contains one of these errors, an attempt to add `y`, which is an integer, to `b`, which is a boolean. The second marked location actually has two errors. The guard of the `if` should be a boolean, not an integer; and the argument to `write` should be an integer, not a boolean. The actual output of Maude, parsed to extract this error information, is the following string (formatted to fit in the provided space, and with the actual URI replaced with `t.silf`):

```
Type checking found errors:
||1:::|t.silf::124::5::9::7::9::12|:::
  Type failure, incompatible operands: (y + b),$int,$bool||
||1:::|t.silf::134::35::11::2::11::37|:::
  Type failure, expression x should have type $bool,
  but has type $int.||
||1:::|t.silf::158::8::11::26::11::34|:::
  Type failure: write expression b has type $bool,
  expected type $int.||
```

The second analysis is a units analysis, which includes unit type annotations, assert and assume statements, loop invariants, and function contracts with preconditions, postconditions, and modifies clauses (useful because SILF supports global variables). An example of a run of this analysis, including a tooltip showing a detected error, is shown in Figure 5. The error is in the addition: variable `projectileWeight` is declared with pounds as the unit, while function `lb2kg` converts an input value in pounds (as specified in the precondition) to an output value in kilograms (as specified in the postcondition). When the units analysis examines the function call, it checks that the input satisfies the precondition (true) and then assumes the output result matches the postcondition. Because of this, the two operands have a different unit, so they cannot be added. Another example is shown in Figure 6, where the invariant fails to hold because of a mistake on line 10, where `y * x` was accidentally written instead of `y * y`, causing the units of

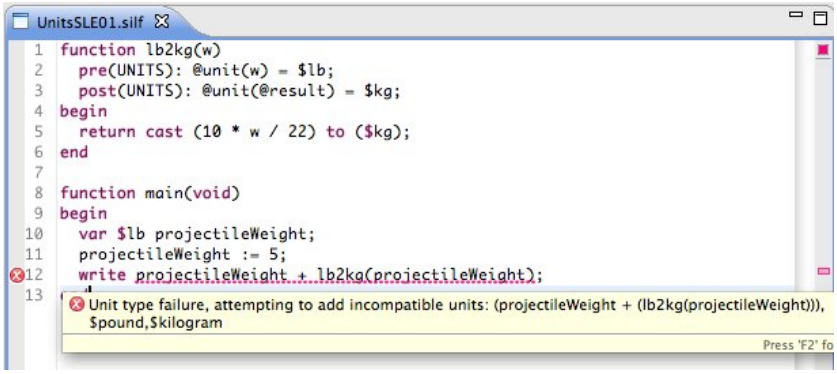

Fig. 5. Units Arithmetic Error in SILF Units Policy

`x` and `y` to be out of sync. This error is detected after one abstract loop iteration, with the detected inconsistency between the units shown in the error message.

All the code for the integration between Rascal and K shown in this section is available in the Rascal subversion repository: links can be found at `http://homepages.cwi.nl/~hills/RascalK`.

5 Related Work

Many tools exist for working with, and potentially executing, formal definitions of programming languages. Generally these tools focus on defining a standard dynamic (evaluation) and maybe static (type) semantics, but, by choosing the appropriate domain, they could also be used for program analysis. Tools with support for some form of development environment include Centaur [2] (for operational semantics) and the Action Environment [17] (for action semantics). A number of approaches for defining semantics and program analyses using term rewriting have also been proposed [7]. For instance, the Meta-Environment [18] provides an open environment for designing and implementing term rewriting environments; one instantiation of this, the ASF+SDF Meta-Environment [19], provides a graphical environment for defining the syntax of a language in SDF, editing programs in the defined language, and performing reductions

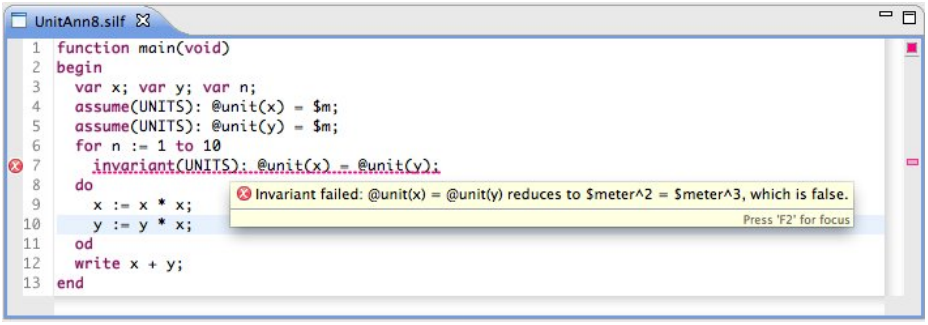

Fig. 6. An Invariant Failure in SILF Units Policy

or transformations with ASF equations [20]. Much like in K, these equations can also be (and have been) used for program analysis.

Along with these, many static analysis tools, including those with some sort of graphical interface for the user, have also been developed, including Frama-C [1] and a number of tools that support JML notation [3].

The main distinctions between these tools and the work presented here are: the use of K for defining the program analysis semantics; the lack of a standard development environment for K (especially as compared to tools such as ASF+SDF); and the lack of a specific source language to analyze (as compared to tools such as JML, which focuses on Java). The work here attempts to address some of the limitations caused by the second point, providing a method to link IDEs developed in Rascal with an analysis semantics given in K, leading to a better experience for the analysis user. On the other hand, since we are focusing specifically on the second point, this work takes the use of K, and the interest in supporting multiple source languages, as a given. K has already been compared extensively to other semantic notations [8,16], and this approach towards program analysis has already been compared to other program analysis approaches [8,11]; due to space concerns, we do not repeat these comparison here.

6 Summary and Future Work

In this paper we have presented a library-based approach to integrating Rascal with analysis tools written using K and running in Maude. This approach allows standard K specifications to be written with only slight modifications to account for the use of source locations and the format of the output. In Rascal, most of the code needed has been encapsulated into a reusable library, requiring only the Maude-ifier and a small amount of "glue" code to be written. The RLSRunner library, including the ScriptExec and ResourceMarkers code, consists of[1] 156 lines of Java code, 114 lines of Rascal code, and 9 lines of Maude code. For SILF, the Maude-ifier is 244 lines of Rascal code – essentially 2 lines per language construct (1 for the match, 1 for the generated string), plus several lines to handle lists. The glue code is 42 lines of Rascal in a single module, plus two lines to add the required menu items to run the analyses. 12 equations were added to the SILF specification to handle the source locations, totaling 25 lines. The definition of source locations is reusable in other K analysis specifications, while all added equations are reusable across other SILF analyses (or analyses in other languages with the same abstract language constructs). As a point of comparison, the total size of the SILF specification, including both analyses used here, is 4428 lines.

In the future we would like to expand the RLSRunner to provide for more execution options, including those related to Maude's search and model checking features (useful for concurrent languages), with appropriate visualizations of the results. We would also like to investigate the automatic generation of the Maude-ifier, and potentially the Maude syntax operators, from the Rascal and Maude language specifications. Finally, once the Rascal C grammar is complete, we plan to use RLSRunner to integrate the generated C environment with CPF [9], an existing analysis framework for C defined using K.

[1] These are just counts of the total number of lines in the file exclusive of blank lines and comments.

References

1. Frama-C. `http://frama-c.cea.fr`.
2. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of SDE 3*, pages 14–24. ACM Press, 1988.
3. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Proceedings of FMICS'03*, volume 80 of *ENTCS*, pages 75–91, 2003.
4. P. Charles, R. M. Fuhrer, and S. M. S. Jr. IMP: A Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse. In *Proceedings of ASE'07*, pages 485–488. ACM, 2007.
5. P. Charles, R. M. Fuhrer, S. M. S. Jr., E. Duesterwald, and J. J. Vinju. Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. In *Proceedings of OOPSLA'09*, pages 191–206. ACM, 2009.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
7. J. Heering and P. Klint. *Rewriting-based Languages and Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*, chapter 15, pages 776–789. Cambridge University Press, 2003.
8. M. Hills. *A Modular Rewriting Approach to Language Design, Evolution and Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 2009.
9. M. Hills, F. Chen, and G. Roşu. Pluggable Policies for C. Technical Report UIUCDCS-R-2008-2931, Department of Computer Science, University of Illinois at Urbana-Champaign, 2008.
10. M. Hills, T. F. Şerbănuţă, and G. Roşu. A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. In *Proceedings of WRLA'06*, volume 176 of *ENTCS*, pages 215–231. Elsevier, 2007.
11. M. Hills and G. Roşu. A Rewriting Logic Semantics Approach To Modular Program Analysis. In *Proceedings of RTA'10*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 151 – 160. Schloss Dagstuhl - Leibniz Center of Informatics, 2010.
12. P. Klint, T. van der Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, volume 0, pages 168–177, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
13. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.
14. J. Meseguer and G. Roşu. The rewriting logic semantics project. In *Proceedings of SOS'05*, volume 156 of *ENTCS*, pages 27–56. Elsevier, 2006.
15. J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
16. G. Roşu and T. F. Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
17. M. van den Brand, J. Iversen, and P. D. Mosses. An Action Environment. *Science of Computer Programming*, 61(3):245–264, 2006.
18. M. van den Brand, P.-E. Moreau, and J. J. Vinju. Environments for Term Rewriting Engines for Free! In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 424–435. Springer, 2003.
19. M. 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, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Proceedings of CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.
20. 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.