Meta-Programming and MDE with Rascal

Jeroen van den Bos, Mark Hills, Paul Klint, Tijs van der Storm
and Jurgen J. Vinju

CWI, INRIA ATEAMS, NFI

2nd Workshop on Algebraic Methods in Model-Based Software
Engineering (AMMSE 2011)
Zurich, Switzerland, 30 June 2011
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
Rascal: From Algebraic Specification to Meta-Programming

Or...
Lessons learned: ASF to ASF+SDF to Rascal
Some background: design principles of Rascal
Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications
Lessons learned: ASF to ASF+SDF to Rascal

Some background: design principles of Rascal

Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications
A Plug for the Paper

- Lessons learned: ASF to ASF+SDF to Rascal
- Some background: design principles of Rascal
- Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications
Lessons learned: ASF to ASF+SDF to Rascal

Some background: design principles of Rascal

Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
What’s Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.
What’s Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.
What’s Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.
What’s Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.
What’s Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.
Rascal Features

- Familiar, C or Java-like syntax
- Immutable data
- Rich built-in data types and pattern matching
- Domain-specific constructs (traversals, comprehensions, regular expressions, fixed-point computation)
- Arbitrary context-free grammars with generalized parsing
- String templates
- Java and Eclipse integration
Extract, Analyze, SYnthesize

System Under Investigation (SUI)

EASY Paradigm

Extract

Internal Representation

Analyze

Synthesize

Results
Rascal is EASY

Rascal follows the EASY paradigm:

- Information is *Extracted* from the program, such as the program’s abstract syntax
- This information is then used to *Analyze* the program, for instance to check consistency, generate a control flow graph, or bind names to definitions
- Finally, the extracted information and the analysis results are used to *Synthesize* the desired results, such as by transforming the code or generating visualizations
Domain Analysis for Rascal: Meta-Programming
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
PROCEDURE Swap(VAR x, y: INTEGER);
VAR
temp: INTEGER;
BEGIN
  temp := x;
  x := y;
  y := temp
END Swap;
Why Look at a Standard Programming Language?

- Similar challenges across standard PLs, DSLs, modelling languages, etc
- Similar desired functionality: IDEs, consistency checking, program analysis, code generation, etc
Why Look at Oberon-0?

- Part of work done for tools competition at this year’s LDTA
- Focused on features as a showcase for Rascal – shows what one could do for a language defined in Rascal
- Features include checkers, code generation, visualization, IDE menu links
- Not too Oberon specific: features shown are ones you could use for your own language
Goals of Oberon-0 Implementation

- Modular
- Functional
- Visual
Goals of Oberon-0 Implementation

- Modular
- Functional
- Visual
Goals of Oberon-0 Implementation

- Modular
- Functional
- Visual
**Parsing in Rascal**

- Grammars defined using Rascal grammar definition notation
- A Rascal program then builds a Java-based parser for the grammar
- Parser is GLL with filtering rules used to remove ambiguities
Example: Oberon-0 Grammar

```rascal
syntax Statement
    = assign: Ident var "::=" Expression exp

| ifThen: "IF" Expression condition "THEN"
    {Statement ";"}+ body
    ElsIfPart*
    ElsePart?
    "END"

| whileDo: "WHILE" Expression condition "DO"
    {Statement ";"}+ body
    "END"
```

Van den Bos, Hills, Klint, Van der Storm, Vinju
Rascal Meta-Programming Architecture

- Grammar Source Code
- Parser Generator
- AST Generator
- AST Source Code
- Parser Source Code
- AST Builder Source Code
- Interpreter Source Code
- Rascal Programs
- Rascal Parser
- Parse Tree
- AST Builder
- AST
- Rascal Interpreter
- Input Output

Legend:
- Build-time flow
- Run-time flow
- Data
- Operation
Code Outlining Example: Java in Eclipse

```java
public void print(IValue arg, IEvaluatorContext eval)
{
    PrintWriter currentOutStream = eval.getStdOut();

    synchronized(currentOutStream)
    {
        try{
            if(arg.getType().isStringType())[
                currentOutStream.print(((IString) arg).getValue().toString());
            }else if(arg.getType().isSubtypeOf(Factory.Tree))
            {
                currentOutStream.print(((IValueFactory) arg).getTree().toString());
            }
        }
    }
}
```
Outlining Support in Rascal: Building the Outline

- Outlines are built over the concrete syntax of a language
- Labels indicate the display name in the outline view
- Locations allow the user to jump to the outlined item
- Once the outliner is registered, the runtime keeps the view up to date as the source is edited
Code Outlining Example: Oberon-0 in Rascal

```
PROCEDURE Multiply;
  VAR x, y, z: INTEGER;
BEGIN
  Read(x);
  Read(y);
  z := 0;
  WHILE x > 0 DO
    IF x MOD 2 = 1 THEN
      z := z + y
    END;
    y := 2*y;
    (* Dog *)
    x := x DIV 2 END;
  Write(x);
  Write(y);
  Write(z);
  WriteLn
END Multiply;

PROCEDURE Divide;
  VAR x, (* Q *) y, r, q, w: INTEGER;
BEGIN
```

Van den Bos, Hills, Klint, Van der Storm, Vinju
Annotators

- Annotators allow annotations to be added to language constructs and displayed in the editor.
- Typical examples: name resolution, type checking – want errors to be displayed graphically to users, marking error locations.

```rascal
public Module checkModule(Module x) {
    m = implode(x);
    <m, st> = resolve(m);
    errors = { error(l, s) | <l, s> <- st.scopeErrors ];
    if (errors == {}) {
        errors = check(m, st.symbolTable);
    }
    return x[@messages = errors];
}

registerAnnotator("l4", checkModule);
```
Annotator Example: Type Checking Oberon-0

```plaintext
MODULE Collatz;

VAR even, odd : INTEGER;

PROCEDURE doCollatz();
VAR current : INTEGER;
    currentEven : BOOLEAN;

PROCEDURE computeEven();
    BEGIN
        IF current MOD 2 = 0 THEN
            currentEven := even
        ELSE
            currentEven := false
        END
    END computeEven;
```

Error message: Cannot assign value of type INTEGER, expected type BOOLEAN.
Contributors provide a way to add more advanced functionality
Each contribution is a menu item – execution is triggered by the user
Examples: interaction with external tools, compilation, visualization
An Example Contributors Menu
Visualization Contribution: Control Flow Graph

```rascal
BEGIN
swap2(w, x);
swap2(x, y);
swap2(y, z);
swap2(z, w);
END swap4;

PROCEDURE swap3Twice(VAR x, y, z : INTEGER)
BEGIN
swap3(x, y, z);
swap3(x, y, z);
END swap3Twice;

BEGIN
a := 1;
b := 2;
c := 3;
d := 4;
Write(a); Write(b); Write(c); Write(d);
swap4(a, b, c, a); swap3Twice(a, b, c, a);
Write(a); Write(b); Write(c); Write(d);
END Swap.
```
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
Goals

- First, define a language, with support tools, for entities
- Then, extend this to support packages for modularity
- Next, extend this language to support entity instances
- Finally, add modular extensions to the language

Note: Work by Tijs van der Storm, presented at LWC’11 by Jurgen J. Vinju
Goals

- First, define a language, with support tools, for entities
- Then, extend this to support packages for modularity
- Next, extend this language to support entity instances
- Finally, add modular extensions to the language

Note: Work by Tijs van der Storm, presented at LWC’11 by Jurgen J. Vinju
Entities and Instances

- Immediate IDE: highlighting, folding, error marking, etc
- Java and SQL generation
- Online checking and error marking
An IDE for Entities

```
1. entity Person {
   2.   string name
   3.   string firstName
   4.   date birthDate
   5.   Car2 ownedCar
  }

6. entity Car {
   7.   string make
   8.   string model
   9.
  10. }
```

Van den Bos, Hills, Klint, Van der Storm, Vinju
Generating Java Code

```java
public class Person {
    private java.lang.String name;
    public java.lang.String getName() {
        return this.name;
    }
    public void setName(java.lang.String name) {
        this.name = name;
    }
}
```

```java
public class Car {
    private java.lang.String make;
    public java.lang.String getMake() {
        return this.make;
    }
    public void setMake(java.lang.String make) {
        this.make = make;
    }
}
```

`rascal>import Plugin; ok
rascal>main(); ok`
Generating SQL Code

Rascal code for generating SQL code:

```rascal
create table Person (  
id int primary key,  
name varchar,  
firstName varchar,  
birthDate date,  
ownedCar int foreign key references Car(_id)  
);

create table Car (  
id int primary key,  
make varchar,  
model varchar  
);
```

Rascal code to import the plugin:

```rascal
rascal>import Plugin;
ok
rascal>main();
ok
```
Defining Entity Concrete Syntax

```rascal
module lang::entities::syntax::Entities
import lang::entities::syntax::Layout;
import lang::entities::syntax::Ident;
import lang::entities::syntax::Types;

start syntax Entities
    = entities: Entity* entities;

syntax Entity
    = @Foldable entity: "entity" Name name "{" Field* "}";

syntax Field
    = field: Type Ident name;

rascal> import lang::entities::syntax::Types;
ok
rascal> `entity X { }
`entity X { }
Entity: appl(prod([lit("entity"),layouts("LAYOUTLIST"),label("name",sort("N
```
Defining Entity Abstract Syntax

```rascal
module lang::entities::ast::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);

int: 2
rascal>2 + 2
int: 4
rascal>3 + 3
int: 6
rascal>
```
String Template-based Code Generation

```rascal
public str entity2java(Entity e) {
    return "public class <.name>.name {  
    \foreach (f <- e.fields) {  
    \field2java(f) 
    }";
}

public str field2java(String::field(typ, n)) {
    let cn = \type2java(typ), capitalize(n)>
    return "private <t> <n>;  
    public <t> get<cn>() {  
    return this.<cn>;  
    }  
    public void set<cn>(<t> <n>) {  
    this.<n> = <n>;  
    }";
}
```
Checking Entities

```rascal
// entities check.rsc

// Imports
import lang::entities::ast::Entities;
import Message;
import Relation;

// Function to check entities
public list[Message] check(Entities es) {
  defs = {};
  errors = for (e <- es.entities) {
    if (e.name in defs) {
      append error("Redeclared entity", e.name@location);
    }
    defs += {e.name};
  }

  return (errors | it + checkEntity(e, defs) | e <- es.entities);
}

// Function to check a single entity
public list[Message] checkEntity(Entity e, set[Name] defs) {
  fs = {};
  return (e.fields | it + checkField(e, fs, defs));
}
```

Van den Bos, Hills, Klint, Van der Storm, Vinju

Meta-Programming and MDE with Rascal
Registering the Contributors
Adding Packages: Package Concrete Syntax

```rascal
module lang::packages::syntax::Packages

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

start syntax Package
  = package: "package" Ident name "{" Import* imports Entities entities ""};

syntax Import
  = imp: "import" Ident name;

// Extension

```

Resolving Names

```rascal
// entities are declared out of order and may have cyclic deps
// if a package imports 2 packages that export the same name it is an error
// (here we assume this has already been checked for)
// (resolve will actually resolve them, hiding the error)

public WorkingSet resolve(WorkingSet pkgs) {
    return { <k, success(l, resolvePkg(pkg, pkgs))> | <k, success(l, pkg)> <- pkgs }
}

private Package resolvePkg(Package pkg, WorkingSet pkgs) {
   imps = imports(pkg) + {pkg.name};
    return visit(pkg) {
        case Name n:name(str x) => qualified(ip.name, x)[@location=n@location]
        when i <- imps, <i, success(_, Package ip)> <- pkgs, x in exports
    }
}
```
Generating Java Code

```java
import List;
import lang::packages::ast::Packages;
extend lang::entities::compile::Entities2Java;

public rel[str, str] package2java(Package pkg) {
    return "<" + pkg.name + ", " + e.name + "", packagedEntity2Java(pkg, e) + " >";
}

public str packagedEntity2Java(Package pkg, Entity e) {
    return "package " + pkg.name + ";
    'import "" + i.name + "";
    ';
    " + entity2java(e) + "";
}

public str type2java(String qualified(String pkg, String name)) = "<" + pkg + ", " + name + "";
```
Results

- 4 languages defined in total
- 5 total IDEs (1 for Rascal, plus 4 more)
- 3 checkers defined
- 3 Java code generators created
- 1 SQL code generator created
- 2 XML code generators created
- Total SLOC: 950
Derric

- Developed for digital forensics
- Allows specification of file formats using a DSL
- Compiled to optimized Java code
- Total code size: 1871 SLOC
- Highly competitive in both speed and precision
- Developed by a PhD student working at the NFI (Dutch Forensics Institute)
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
Contributors in Rascal-based IDEs are not limited to those written in Rascal

Example: linking a Rascal-based front-end with a Maude-based analysis framework
What is Needed to Make the Link (Rascal)?

- Grammar for the language
- *Maudeifier* to generate Maude-readable form of program
- Support for starting, reading from, writing to, and stopping Maude
- Support for preparing individual tasks and reading back results
- Eclipse interaction to display results
What is Needed to Make the Link (Maude)?

- Specification support for Rascal source locations (if used)
- Result generation in parsable format (not necessarily human readable)
Information from the external tool can be used to set up annotations...
... and to add other information, such as entries in an Eclipse Problems view.
Outline

1. Introduction to Rascal
2. Rascal for Language Development
3. Developing Modelling Languages
4. Tying into Existing Specifications
5. Rascal: Future Development Plans
Syntax: features and documentation finished by the end of September for an early October release

Performance: ongoing work on improving the performance of program evaluation, with special focus on function call and pattern match performance

Type Checking: currently uses a runtime type system, switching over to static system – work mostly done, but integrating into Rascal more closely, improving performance so it can run constantly as files are edited
Rascal Development: A Rough Future Timeline

- Syntax: features and documentation finished by the end of September for an early October release
- Performance: ongoing work on improving the performance of program evaluation, with special focus on function call and pattern match performance
- Type Checking: currently uses a runtime type system, switching over to static system – work mostly done, but integrating into Rascal more closely, improving performance so it can run constantly as files are edited
Syntax: features and documentation finished by the end of September for an early October release

Performance: ongoing work on improving the performance of program evaluation, with special focus on function call and pattern match performance

Type Checking: currently uses a runtime type system, switching over to static system – work mostly done, but integrating into Rascal more closely, improving performance so it can run constantly as files are edited
Rascal in Moving to Eclipse.org!
Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications
Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications
Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
  - MDE: DERRIC
- Linking to Existing Specifications
Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications
Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications
For More Information...

- Rascal: http://www.rascal-mpl.org
- IMP: http://www.eclipse.org/imp
- CWI SEN1: http://www.cwi.nl/sen1