

The RISC ProgramExplorer Tutorial and Manual¹

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
<http://www.risc.jku.at>

April 8, 2010

¹The hypertext version of this document is available at <http://www.risc.jku.at/research/formal/ProgramExplorer/manual>.

Abstract

This document describes the use of the RISC ProgramExplorer, an interactive program reasoning environment that is under development at the Research Institute for Symbolic Computation (RISC). The current version is a first demonstrator skeleton that incorporates the overall technological and semantic framework (programming language and formal specification language) and integrates the RISC ProofNavigator as an interactive proving assistant. Work is going on to provide this skeleton with the envisioned program reasoning capabilities. The software runs on computers with x86-compatible processors under the GNU/Linux operating system; it is freely available under the terms of the GNU GPL.

Contents

1	Introduction	4
2	User Interface	6
3	Examples	14
3.1	Computing Factorial Numbers	14
3.2	Searching for Records	24
3.3	Failed Tasks and Interactive Proofs	31
A	Programming Language	36
B	Specification Language	39
B.1	Logic Language	39
B.1.1	Declarations	39
B.1.2	Types	40
B.1.3	Mapping Program Types to Logical Types	41
B.1.4	Program Variables	42
B.1.5	Program States	43
B.1.6	State Functions	44
B.2	Theory Definitions	46
B.3	Class Specifications	47
B.4	Method Specifications	48
B.5	Loop Specifications	50
B.6	Statement Specifications	51

CONTENTS **3**

C Program Invocation **52**

D Program Installation **55**

D.1 README 55

D.2 INSTALL 58

E Task Directories **62**

F Grammars **64**

F.1 Programming Language 64

F.2 Specification Language 69

Chapter 1

Introduction

This document describes the current state and the use of the RISC ProgramExplorer, an interactive program reasoning environment that is under development at the Research Institute for Symbolic Computation (RISC). The work reported in this document is based on prior work on the RISC ProofNavigator [8, 5], an interactive proving assistant that is fully integrated into the RISC ProgramExplorer. Eventually the environment shall provide advanced program analysis and reasoning capabilities based on a calculus elaborated in [6, 7]. The current version should be mainly considered as a first demonstrator skeleton that incorporates the overall technological and semantic framework (programming language and formal specification language) in an elaborated graphical user interface. The actual analysis and reasoning capabilities will be integrated in a future edition.

The system is freely available under the GNU Public License at the URL

```
http://www.risc.jku.at  
/research/formal/software/ProgramExplorer
```

It has been reasonably well tested with small examples but is certainly not free of bugs; the author is glad to receive error reports at

```
Wolfgang.Schreiner@risc.jku.at
```

The remainder of the document is split in two parts:

- **Chapters 2–3** essentially represent a tutorial for the RISC ProgramExplorer based on examples contained in the software distribution; for learning to use the system, we recommend to study this material in sequence.

- **Appendices A–F** essentially represent a reference manual with an explanation of the software’s programming and specification language; this material can be studied on demand.

The RISC ProgramExplorer uses the following third party software; detailed references can be found in the README file of the distribution listed on page 55:

- CVC Lite
- RIACA OpenMath Library
- General Purpose Hash Function Algorithms Library
- ANTLR
- Eclipse Standard Widget Toolkit
- Mozilla Firefox
- GIMP Toolkit GTK+
- Sun JDK
- Tango Icon Library

Many thanks to the respective authors for their great work.

Chapter 2

User Interface

In the following we explain the main points of interaction with the user interface of the RISC ProgramExplorer. We assume that the system is appropriately installed (see Appendix D) and that the current working directory is the subdirectory `examples` of the installation directory with write permission enabled (respectively a writable copy of that directory). After typing on the command line

```
ProgramExplorer &
```

a window pops up that displays the startup screen shown in Figure 2.1.

This window has three menus at the top:

File The menu entry “New File” creates a new file; files with extension `.java` are considered as program files, files with extension `.theory` are considered as specification files. The menu entry “Open File” opens such a file. The menu entry “Close file” closes the currently selected open file. The menu entry “Close all files” closes all open files. The menu entry “Save file” saves the currently selected open file to disk.

The menu entry “Workspace...” displays the window shown in Figure 2.2. This window displays those directories that together represent the root of the package hierarchy for the RISC ProgramExplorer. The default is the list of those directories set in the environment variable `PE_CLASSPATH` (see Section C) respectively, if the variable is not set, the current working directory. The buttons “Add Directory” and “Remove Directory” modify the list, the button “Restore Directories” restores the original setting. The button “Okay” activates the current selection, the button “Cancel” discards it.

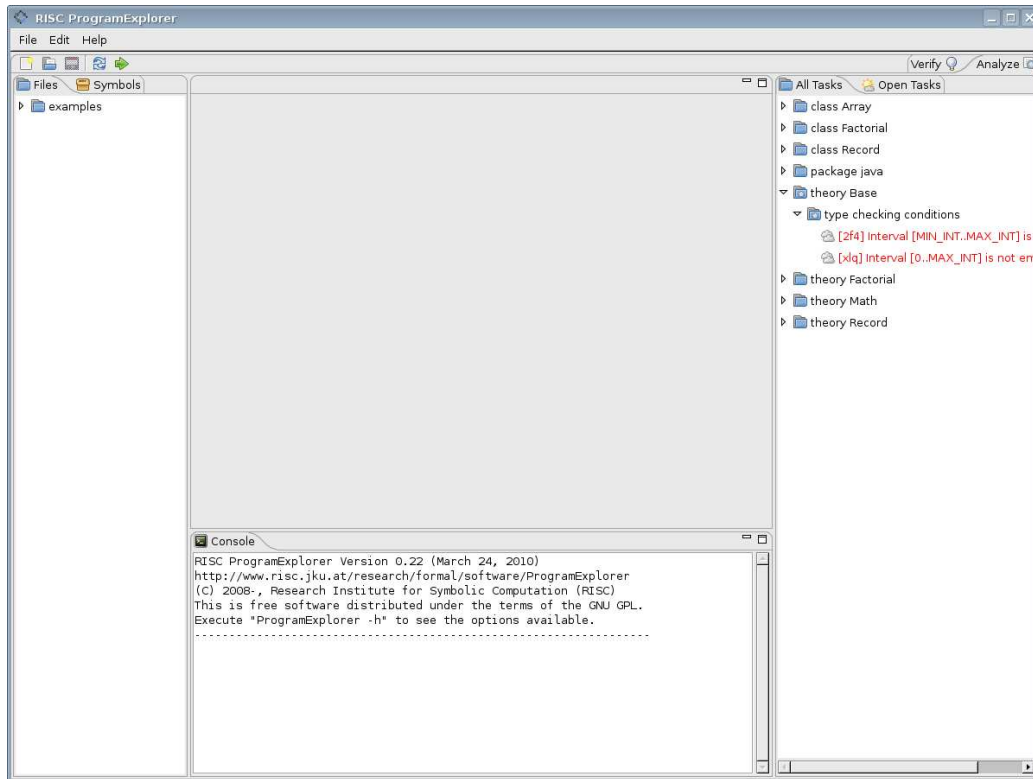


Figure 2.1: Startup Window

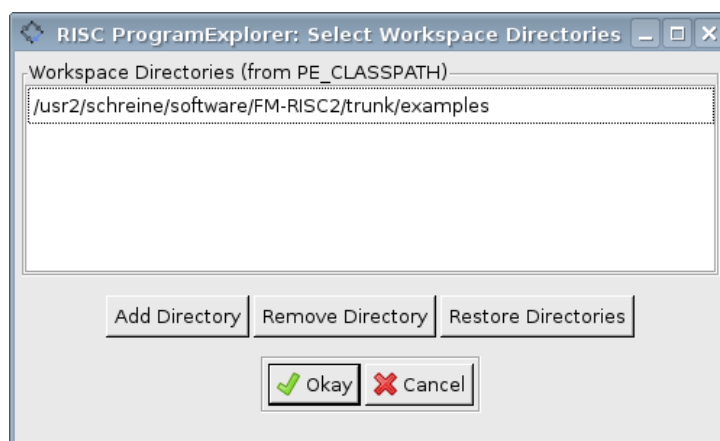


Figure 2.2: Workspace Configuration

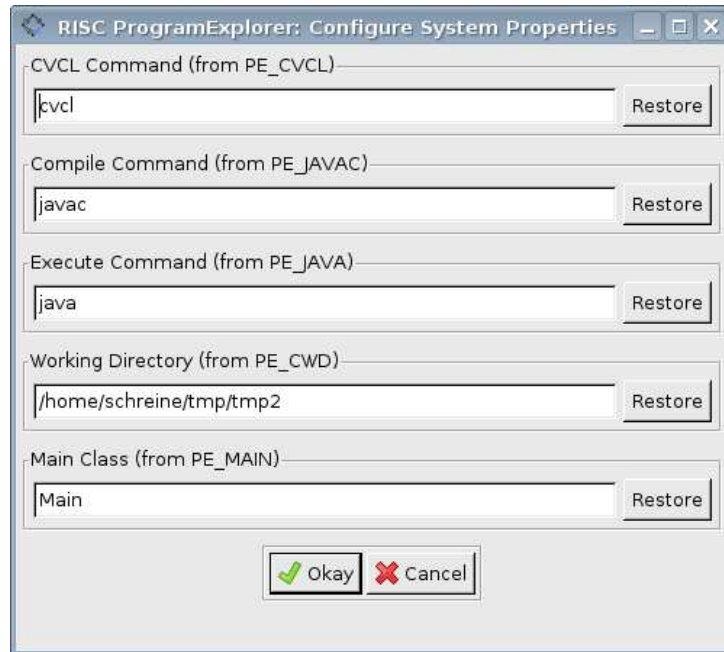


Figure 2.3: Properties Configuration

The menu entry “Properties...” displays the window shown in Figure 2.3. The window allows to configure various properties: the path to the executable of the Cooperating Validity Checker Lite (CVCL) version 2.0, the path to the Java compiler, the path to the Java application launcher, the path of the working directory (used e.g. for creating new files), the path for the main class of the program (the class containing method `main`). The values of these variables can be configured by various environment variables (see Section C). The button “Okay” confirms any modifications, the button “Cancel” discards them.

The menu entry “Quit” terminates the program.


Edit The menu entry “Undo” undoes the last change in the file currently being edited, the menu entry “Bigger/Smaller font” allows to change the size of the font of the editor and of the console.


Help The entry “Online Manual” displays the hypertext version of this document; the entry “About RISC ProgramExplorer” displays a copyright message.


Below the menu, a row of buttons is displayed as shown in Figure 2.4.





Figure 2.4: Analyze Buttons

New File  Like the menu option “New File”, this button creates a new file and opens it in the editing area.

Open File  Like the menu option “Open File”, this button opens an already existing file.

Save File  Like the menu option “Save File”, this button saves an open file that was modified in the editing area.

Refresh View  This button removes from the view all information (symbols and tasks) that was created by processing a class or theory.

Run Program  This button calls the Java compiler to compile the “Main” class indicated in the “Properties” configuration and calls the Java application launcher to execute it; the output is displayed in the console window (currently no input is possible).

The main area of the window is split into four areas (whose borders may be dragged by the mouse pointer). The central area (which is initially empty) is the “editing” area where program and specification files may be displayed and edited. The other three areas are:

Console This area displays textual output of the RISC ProgramExplorer, initially a copyright message. When program/specification files are processed, this area displays the success status respectively error messages, if something went wrong.

Files/Symbols In this area, the tabs “Files” and “Symbols” display the directory respectively symbol structure of the workspace as shown in Figure 2.5. By moving the mouse pointer over a directory/file, a yellow “tip” window pops up that displays the path of the corresponding directory/file respectively information on the corresponding symbol.

Double-clicking on a file opens the corresponding file in the central editing area. Right-clicking on a directory opens a pop-up menu with an option “Refresh” to refresh the display of the directory content and an option

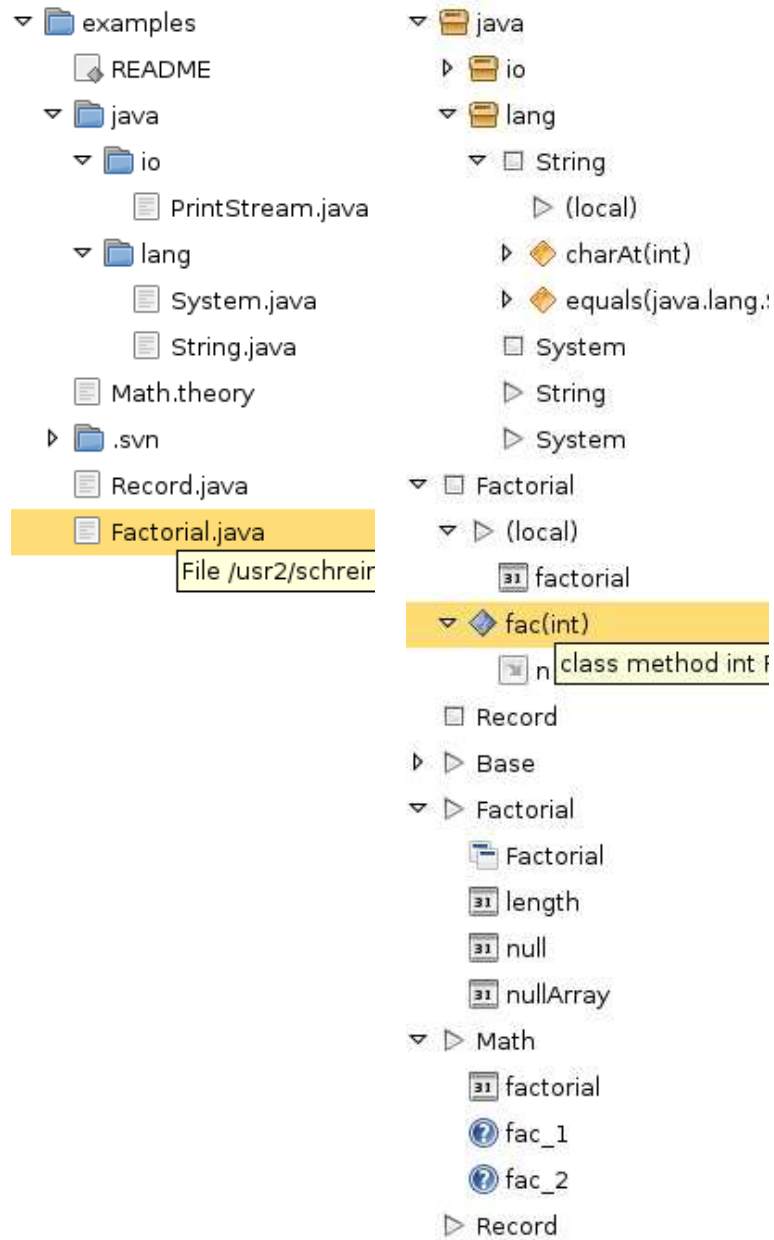




Figure 2.5: Workspace Files/Symbols


“Delete” to delete the directory (after a confirmation). Right-clicking on a file opens a pop-up menu with an option “Open” to open the file in the editing area and an option “Delete” to delete the file (after a confirmation).


Double-clicking on a symbol (e.g. a class symbol or a theory symbol) also opens the corresponding source file (a `.java` file or a `.theory` file) in the editor but also immediately processes it; the success of the operation is displayed in the “Console” area. There are the following kinds of symbols:


Package  A symbol denoted by a `package` declaration.


Class  A symbol introduced by a `class` declaration.


Class Variable  A symbol introduced by the declaration of a `static` variable in a class.


Object Variable  A symbol introduced by the declaration of a non-`static` variable in a class.


Class Method  A symbol introduced by the declaration of a `static` method in a class.


Object Method  A symbol introduced by the declaration of a non-`static` method in a class.


Constructor  A symbol introduced by the declaration of a constructor in a class.

Method Parameter  A symbol introduced by the declaration of a parameter in a method header in a class.


Theory  A symbol introduced by a `theory` declaration.


Type  A symbol introduced by a `TYPE` declaration in a theory.

Value  A symbol introduced by a value declaration in a theory.

Formula/Axiom  A symbol introduced by a `FORMULA/AXIOM` declaration in a theory.

All Tasks/Open Tasks In this area, the tabs “All Tasks” and “Open Tasks” display the tree of all tasks organized in task folders respectively the list of all open tasks as shown in Figure 2.6. The status of the task is indicated by in icon and the color of the description:

New Task  This task (described in red color) is *new* i.e. it has not yet been attempted to solve it.

Open Task  This task is (described in red color) *open* i.e. it has been attempted but not yet solved.

The screenshot displays a project tree on the left and a list of tasks on the right.

Project Tree (Left):

- class Array
- class Factorial
 - method fac
 - type checking conditions
 - [twk] value is natural number
 - [hx3] value is natural number
 - theory (local)
 - type checking conditions
- class Record
 - method Record
 - method equals
 - method main
 - method search
 - type checking conditions
 - [qtt] value is in interval
 - theory (local)
 - type checking conditions
 - [s3f] value is in interval
 - type checking conditions
 - package java
 - theory Base
 - type checking conditions
 - [2f4] Interval [MIN_INT..MAX_INT] is
 - [x] Task: [2f4] Interval [MIN_INT..MAX
 - theory
 - Task: [2f4] Interval [MIN_INT..MAX
 - Status: new
 - Type: verify type checking conditi
 - Goal formula: MIN_INT <= MAX_I
 - theory
 - Task: [2f4] Interval [MIN_INT..MAX
 - Status: new
 - Type: verify type checking conditi
 - Goal formula: MIN_INT <= MAX_I
 - theory Record


Task List (Right):


- [qtt] value is in interval
- [s3f] value is in interval
- [2f4] Interval [MIN_INT..MAX_INT] is not e
- [xlq] Interval [0..MAX_INT] is not empty

Task details for [xlq] Interval [0..MAX_INT] is not empty:

- Task: [xlq] Interval [0..MAX_INT] is not
- Status: new
- Type: verify type checking condition
- Goal formula: MIN_INT <= MAX_INT =

Figure 2.6: All/Open Tasks

Closed Task  This task is *closed*, i.e. it has been successfully solved. The task is typically described in blue; if the description is in violet, the task was solved by a proof in a previous invocation of the RISC ProgramExplorer. The corresponding proof may be then replayed in the current invocation.

Failed Task  This task (described in red) is *failed*, i.e. the task is impossible to solve (which indicates a program/specification error).

By moving the mouse pointer over a task, a yellow “tip” window pops up that displays information on the task such as the kind of task and its status. By double-clicking on the task, the position in the source code of the program or theory is displayed that triggered the creation of the task.

By right-clicking on the task a pop-up menu shows various options depending on the kind of task: “Execute Task” attempts to solve the task e.g. by an automatic proof or, if that fails, by a computer-assisted interactive proof; “Print Task” prints information on the task (the content of the “tip” window) in the “Console” area. “Print State Proving Problem” prints a translation of the task into a proving problem in an extended logic that involves reasoning about program states. “Print Classical Proving Problem” prints a translation of the problem into a classical predicate logic proving problem. “Print Status Evidence (Proof)” shows an associated proof; “Reset Task” resets the task into the “new” state (and deletes any associated proof).

By right clicking on a task folder, a pop-up menu shows up whose option “Execute Task” attempts to solve all tasks in the folder by an automatic proof (interactive proofs have to be individually triggered as shown above). By right the tab “All Tasks/Open Tasks” itself, a menu pops up whose option “Execute all tasks” attempts to solve all tasks by automatic proofs; task folders that only contain closed proofs are then closed as well.

Chapter 3

Examples

In this chapter, we are going to illustrate the current features of the RISC Program-Explorer by some small examples (that are included in the software distribution). The software currently represents in essence a demonstrator skeleton which allows

- to write programs in a subset of Java called “MiniJava” (see Appendix A) and have them parsed and type-checked;
- to use a logic language (see Appendix B.1) in order to write logical theories (see Appendix B.2) and have them parsed and type-checked;
- annotate programs with program specifications (see Appendix B) and have them parsed and type-checked;
- prove the generated *type-checking conditions*, either by automatic proofs (using the integrated Cooperating Validity Checker Lite CVCL [3, 2]), or, if this should not succeed, by a computer-assisted interactive proof (using the integrated RISC ProofNavigator [8, 5]).

A future edition of the RISC ProgramExplorer will support true *reasoning/verification tasks* derived from the formal program specifications.

3.1 Computing Factorial Numbers

This example is about the specification of the following program

```

public class Factorial
{
    public static int fac(int n)
    {
        int i=1;
        int p=1;
        while (i <= n)
        {
            p = p*i;
            i = i+1;
        }
        return p;
    }
}

```

The program is written in Java-syntax; it introduces a method `fac` which is supposed to return the factorial of its argument `n`.

The specification of the program is to be based on the mathematical function $factorial: \mathbb{N} \rightarrow \mathbb{N}$ which is uniquely characterized by the axioms

$$\begin{aligned}
 factorial(0) &= 1 \\
 \forall n \in \mathbb{N}: factorial(n+1) &= (n+1) \cdot factorial(n)
 \end{aligned}$$


First we describe how to define the corresponding mathematical theory, next we describe how to specify the program with the help of this theory.

Theory We define a theory *Math* which introduces a function *factorial* on the natural numbers and constrains its behavior by two axioms as discussed above:

```

theory Math
{
    // an axiomatic specification of
    // the factorial function
    factorial: NAT -> NAT;
    fac_1: AXIOM factorial(0) = 1;
    fac_2: AXIOM FORALL(n: NAT):
        factorial(n+1) = (n+1)*factorial(n);
}

```

We can use the RISC ProgramExplorer to write this theory in a file *Math.theory* in the unnamed top-level package as follows: we select the button *New File* ,

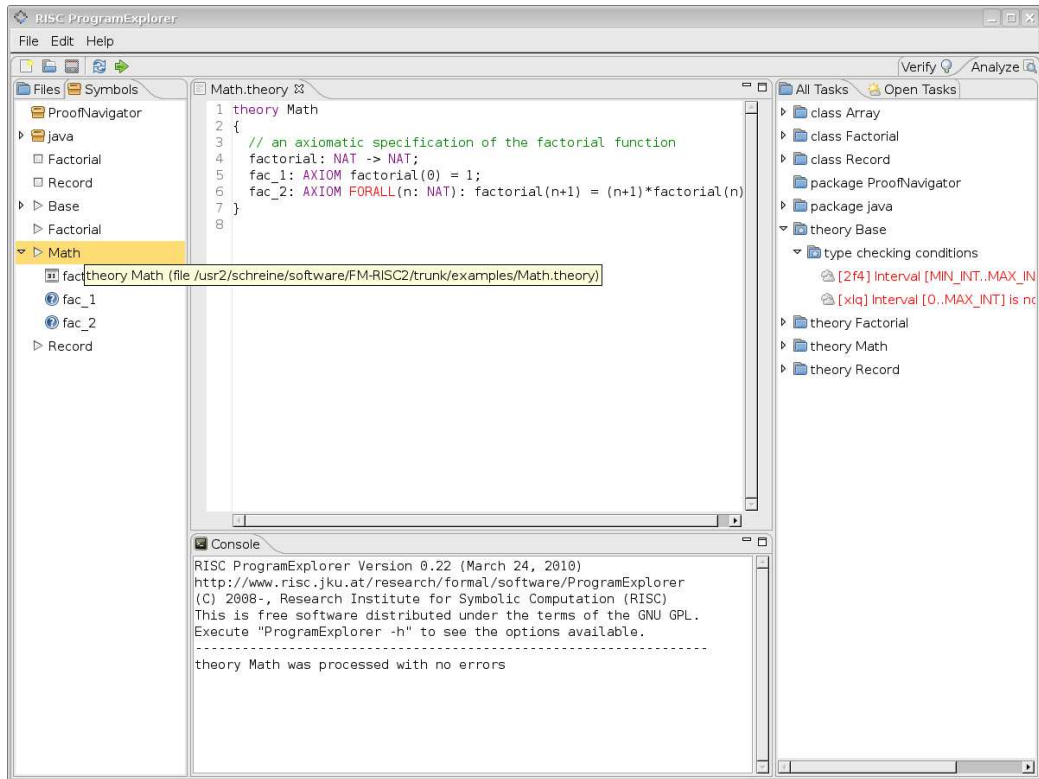

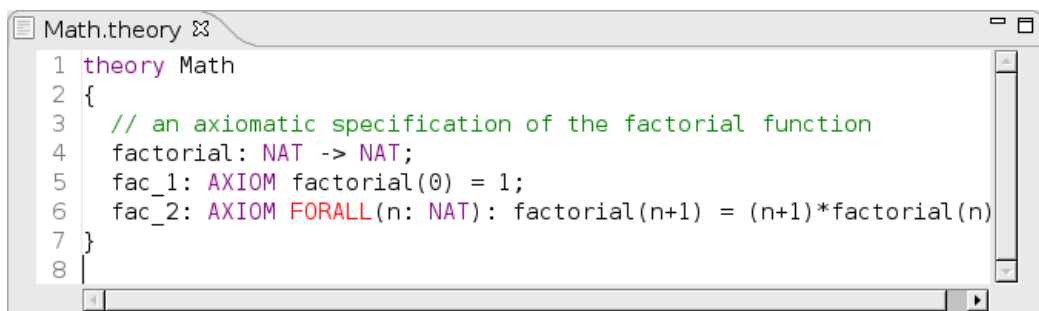


Figure 3.1: A Logic Theory

enter the file name `Math.theory`, and press *Okay*. In the central region a new editing area titled `Math.theory` opens; we enter above theory declaration and press the button *Save File* . The RISC ProgramExplorer window has then the state shown in Figure 3.1. The theory is displayed as



with colors indicating keywords of the specification language. Identifiers are *active*, e.g. by double-clicking on *factorial* the identifier is highlighted, the tab *Symbol* on the left side of the window highlights the corresponding symbol and the *Console* area displays

```
value factorial: (NAT) -> NAT
factorial: (NAT) -> NAT
```

(likewise the identifiers *Math*, *fac_1*, and *fac_2* can be double-clicked). Moving the mouse pointer over the symbol on the left tab displays a corresponding yellow “tip” window, clicking with the right mouse-button allows to choose between *Print Symbol* and *Print Declaration*. Choosing the symbol *Math* and selecting *Print Symbol* displays in the console area output similar to

```
theory Math (file ../../examples/Math.theory)
```

Selecting *Print Declaration* displays

```
theory Math
{
  factorial: (NAT) -> NAT;
  fac_1: AXIOM factorial(0) = 1;
  fac_2: AXIOM FORALL(n: NAT): factorial(n+1) = ...;
}
```



If we introduce in the declaration an error, e.g. by mistyping the function name as *factorials* in axiom

```
fac_1: AXIOM factorials(0) = 1;
```

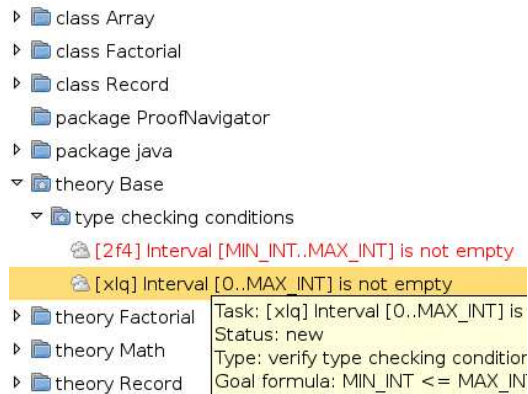
the *Console* area shows the output



```
ERROR (Math.theory:5:16):
  there is no value named factorials
theory Math was processed with 1 error
```

In the editing area, the theory is then displayed as

The position of the error in the file is indicated by an icon  on the left bar, by a corresponding red marker on the right bar and by underlining the syntactic phrase in red; moving the mouse pointer over the icon on the left or over the marker on the right displays the corresponding error message. The same icon at the top of the editing tab and in the tab *Symbols* indicates that the theory has an error. Moving the mouse pointer over the red square on the top-right corner of the editing area displays the number of errors in the theory. After fixing the error and pressing the button *Save File* , the correct state is restored.

The tab *All Tasks* on the right now looks as follows:



It displays a folder *theory Base* with subfolder *type checking conditions*; the folder icons  indicate that these folders contain (subfolders with) open tasks to be performed. Indeed, the subfolder *type checking conditions* contains two tasks labeled *Interval [MIN_INT..MAX_INT] is not empty* and *Interval [0..MAX_INT] is not empty* (the task labels start with automatically generated tags of the form *[ccc]* for unique referencing). The icon  indicates that this task is “new”, the red font of the task description indicates that the task is not yet performed. Moving the mouse pointer over the tasks (respectively right-clicking the tasks and selecting the option *Print Task*) shows the task descriptions:

```
Task: [2f4] Interval [MIN_INT..MAX_INT] is not empty
Status: new
Type: verify type checking condition
Goal formula: MIN_INT <= MAX_INT
```

```
Task: [xlq] Interval [0..MAX_INT] is not empty
Status: new
Type: verify type checking condition
Goal formula: MIN_INT <= MAX_INT => 0 <= MAX_INT
```

Right-clicking the tasks and selecting the option *Print Classical Proving Problem* shows the detailed proofs to be performed for performing the tasks: in the first case, this proof is

```
Declarations:
STRING: TYPE;
MIN_INT: INT = -2147483648;
MAX_INT: INT = 2147483647;
Goal: MIN_INT <= MAX_INT
```

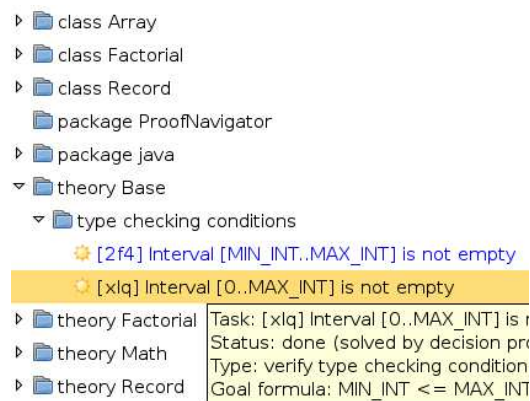
and in the second case, this proof is


```
Declarations:
STRING: TYPE;
MIN_INT: INT = -2147483648;
MAX_INT: INT = 2147483647;
Goal: MIN_INT <= MAX_INT => 0 <= MAX_INT
```

As indicated by the folder names *theory Base* and *type checking conditions*, these tasks have been generated by type-checking the automatically constructed theory *Base* displayed in the *Symbol* tab to the left; Right-clicking the theory symbol and selecting the option *Print Declaration* displays the definition of the theory

```
theory Base
{
  MIN_INT: INT = -2147483648;
  MAX_INT: INT = 2147483647;
  int: TYPE = [MIN_INT..MAX_INT];
  nat: TYPE = [0..MAX_INT];
  ...
}
```

The theory introduces two types *int* and *nat* as integer intervals with bounds *MIN_INT* and *MAX_INT* respectively 0 and *MAX_INT*. Since types must not be empty, the type checker generated the tasks to prove $MIN_INT \leq MAX_INT$ and $0 \leq MAX_INT$ shown above. These simple tasks can be automatically proved. Right-clicking the tasks and selecting the option *Execute Task* immediately closes the tasks (alternatively we may also right-click the parent folder and select *Execute Task* or right-click the *All Tasks* button and select *Execute All Tasks*). The tab *All Tasks* now looks as follows:



The task icon  and the blue font color indicate that the tasks are now “closed”, so the theory could be type-checked well. Right-clicking the tasks and selecting “Reset Task” resets the tasks to their original “new” state.

Program We can now create (in analogy to the creation of file *Math.theory*) a program file with the name *Factorial.java* which holds the program class *Factorial* described above. Saving the file type-checks it and creates in tab *Symbols* a class symbol *Factorial* with method symbol *fac* and parameter symbol *n*. As for theories, also the identifiers in the source file are active, double-clicking e.g. on *fac* in the editing area prints the method declaration in the *Console area* and highlights the symbol *fac* in the *Symbols* tab. Correspondingly, double-clicking the symbol *fac* moves the editor to the position of the declaration of the method and highlights its header.




To formally specify the behavior of the method *fac* with the help of the theory *Math*, we annotate the class *Factorial* by special program comments `/*@ . . . @*/` as follows:

```
/*@
theory uses Math {
```

```

    // the mathematical factorial function
    factorial: NAT -> NAT = Math.factorial;
}
/*@/
public class Factorial
{
    public static int fac(int n) /*@
        requires VAR n >= 0;
        ensures VALUE@NEXT = factorial(VAR n);
    @*/
    {
        int i=1;
        int p=1;
        while (i <= n) /*@
            invariant 1 <= VAR i AND VAR i <= VAR n+1
                AND VAR p = factorial(VAR i);
            decreases VAR n - VAR i + 1;
        @*/
        {
            p = p*i;
            i = i+1;
        }
        return p;
    }
}

```

The RISC ProgramExplorer window has then the state shown in Figure 3.2. The actual program code is displayed as shown in Figure 3.3. Annotations can become “folded” away from the program source code; clicking on the icon  folds the annotation, clicking on the icon  unfolds it again. Moving the mouse pointer over the icon  displays the content of the folded annotation in a yellow “tip” window.

The annotation `theory ...` before the class declaration introduces the “local” theory for the class i.e. those entities that may be further on referenced by short names; the `uses Math` clause indicates that the local theory refers to entities of the previously defined theory *Math*. The local theory is simple: it just defines a function *factorial* by the corresponding function in theory *Math* which can be referenced by the long name *Math.factorial*. Alternatively, we might have referred in the following directly to *Math.factorial* (however, even then an empty declaration `theory uses Math { }` is required because we refer to theory *Math*) or we might have just axiomatized the function *factorial* directly in the local theory (without referring to theory *Math* at all).

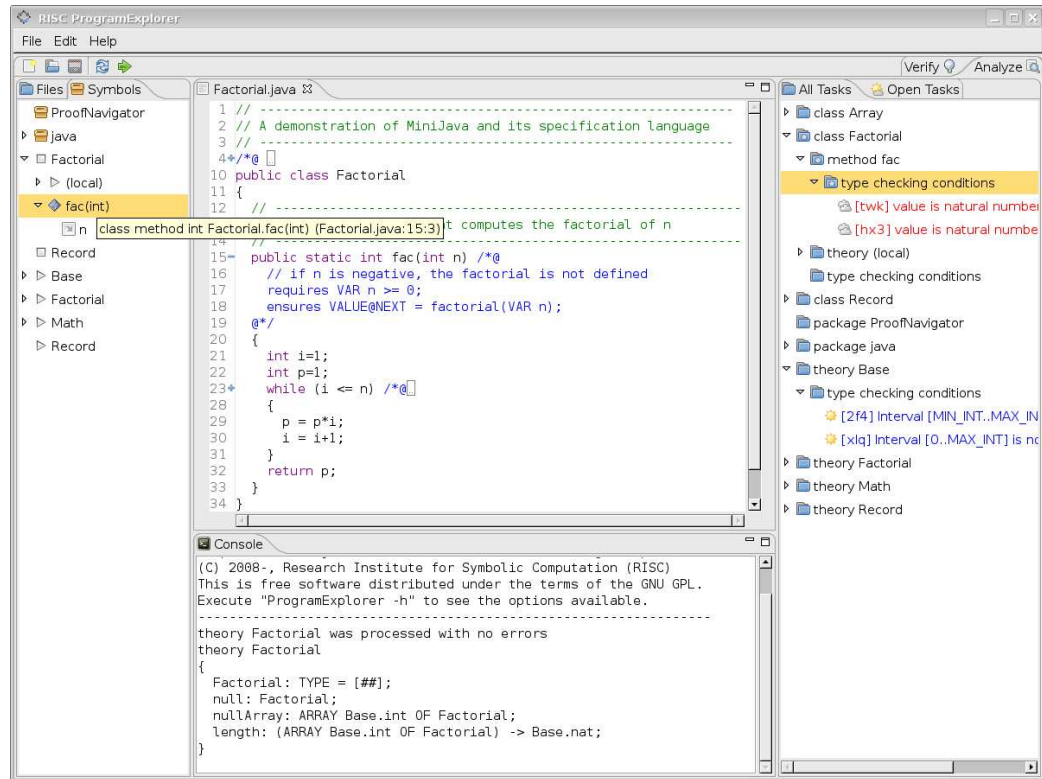


Figure 3.2: A Program Class

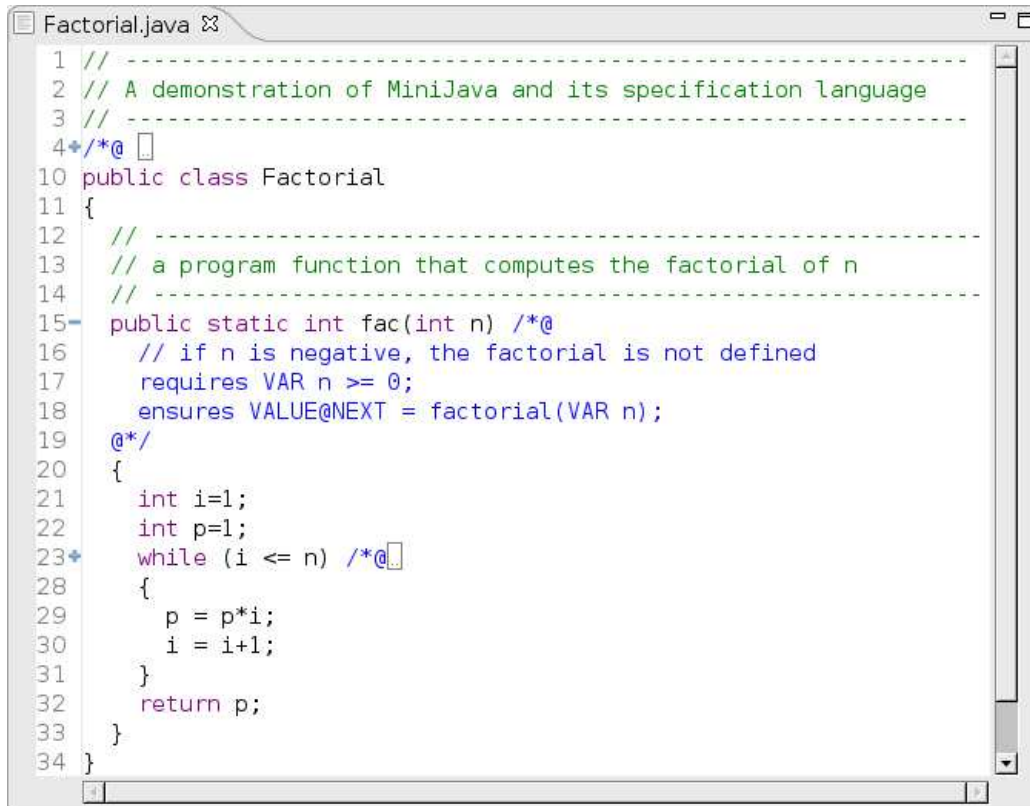
The annotation `requires ...ensures ...` after the header of method `fac` introduces a method specification by a precondition (`requires ...`) that describes the assumptions on the prestate of the method call (in particular constraints of the method arguments) and a postcondition (`ensures ...`) that describes the obligation on the poststate of the method call (in particular obligations on the method result). In our case, the precondition

```
requires VAR n >= 0;
```

states that the value of the program variable n (the method parameter) indicated by the term `VAR n` must not be negative when the method is called; the postcondition

```
ensures VALUE@NEXT = factorial(VAR n);
```

states the method result indicated by the term `VALUE@NEXT` must be identical to the value of the logical function *factorial* when applied to the value of n .



```

1 // -----
2 // A demonstration of MiniJava and its specification language
3 // -----
4+/*@
10 public class Factorial
11 {
12 // -----
13 // a program function that computes the factorial of n
14 // -----
15 public static int fac(int n) /*@
16 // if n is negative, the factorial is not defined
17   requires VAR n >= 0;
18   ensures VALUE@NEXT = factorial(VAR n);
19 @*/
20 {
21   int i=1;
22   int p=1;
23+  while (i <= n) /*@
28   {
29     p = p*i;
30     i = i+1;
31   }
32   return p;
33 }
34 }

```

Figure 3.3: The Program Class in Detail

Finally, the body of the `while` loop is annotated by a loop invariant and a termination term: the loop invariant essentially states the relationship of the prestate of the loop to the poststate of every iteration of the loop body; the termination term denotes a non-negative integer number that is decreased by every iteration of the loop. In our case, the invariant

$$\text{invariant } 1 \leq \text{VAR } i \text{ AND } \text{VAR } i \leq \text{VAR } n+1 \\ \text{AND } \text{VAR } p = \text{factorial}(\text{VAR } i);$$

limits the range of the iteration counter i and states that the value of the program variable p is identical to the factorial of the value of i . The termination term

$$\text{decreases } \text{VAR } n - \text{VAR } i + 1;$$

states that the value of i is decremented by every loop iteration but does not become bigger than $n+1$.

Type-checking the annotation gives rise to two new tasks inserted in task folder *class Factorial*, subfolder *method fac*, subfolder *type checking conditions*. As usual double-clicking on the tasks highlights the corresponding source code positions; moving the mouse over the tasks shows their description:

```
Task(Factorial.java:15:32:4:36):
  [twk] value is natural number
Annotation line:
  ensures VALUE@NEXT = factorial(VAR n);
                                     ^

Status: new
Type: verify type checking condition
Goal formula: old n >= 0 => MIN_INT >= 0 OR var n >= 0

Task(Factorial.java:23:20:3:35):
  [hx3] value is natural number
Annotation line:
  AND VAR p = factorial(VAR i);
                                     ^

Status: new
Type: verify type checking condition
Goal formula:
  old n >= 0 AND 1 <= var i AND var i <= (var n+1) =>
  MIN_INT >= 0 OR var i >= 0
```

As we can see, both conditions were derived from applying the function *factorial* to the value of a program variable. Since *factorial* is only defined on natural numbers, it has to be proved that the value of the respective program variable is not negative. Right-clicking on the folder *type checking conditions* and selecting the option *Execute Task* discharges these conditions automatically such that no task remains open.

3.2 Searching for Records

This example deals with the specification of a program that searches in an array of records for a record with a specific key. The example is based on the following program class:

```
class Record
{
  String key;
```

```
int value;

Record(String k, int v)
{
    key = k;
    value = v;
}

boolean equals(String k)
{
    boolean e = key.equals(k);
    return e;
}

public static int search(Record[] a, String key)
{
    int n = a.length;
    for (int i=0; i<n; i++)
    {
        Record r = new Record(a[i].key, a[i].value);
        boolean e = r.equals(key);
        if (e) return i;
    }
    return -1;
}

public static void main()
{
    int N = 10;
    Record[] a = new Record[N];
    for (int i=0; i<N; i++)
        a[i] = new Record("abc", i);
    a[5] = new Record("xyz", 5);
    int i = search(a, "xyz");
    System.out.println(i);
}
}
```

This program introduces an object type *Record* with a string field *key* and an integer field *value*. The type has a constructor to build a record from an given string and integer and a method *equals* that allows to check whether the record has the denoted key. This function calls the method *equals* on object type *String*; while this class is part of the Java standard library, it has to be explicitly defined in the

RISC ProgramExplorer. We therefore introduce a dummy class

```
package java.lang;
public class String
{
    public boolean equals(String s) return false;
}
```

solely for declaring the method *equals* (without caring for the actual representation of strings or the actual implementation of the method). Unlike real Java, our programming language only allows to call program methods with return values to initialize/assign to variables, not as parts of program expressions¹. The two statements

```
boolean e = key.equals(k);
return e;
```

in the body of *equals* can therefore *not* be merged into one.

The core of the program is the method *search* which takes an array *a* of records and a key and returns the index of the first record in *a* that contains that key (or -1 , if there is no such record). The core of the method body is represented by the two statements

```
Record r = new Record(a[i].key, a[i].value);
boolean e = r.equals(key);
```

The first statement builds a record *r* from the key and the value of record *a*[*i*]. The second statement calls the method *equals* on *r* to compare its key with *key*. This apparently clumsy way of using the function *equals* is necessary because the specification formalism considers object variables (variables of object types) to hold *object values* rather than *object references* (which considerably simplifies reasoning because then the modification of an object via one variable cannot affect an object referenced by another variable).

However, since the programming language Java (like most programming languages) lets object variables hold references, the semantics of our programming language would deviate from classical program semantics. Therefore the type checker ensures that two different program variables cannot refer to the same object; consequently it does not make any difference whether an object variable

¹The reason is that methods may cause side effects and we do not want the computation of program expressions to cause side effects

holds an object value or an object reference. Consequently, if *equals* would modify its record, above solution would not update array *a* (independent of whether object variables hold object values or object references) while the solution

```
Record r = a[i];
boolean e = r.equals(key);
```

would update *a* in a language with reference semantics for objects but not in a language with value semantics. For similar reasons, the even shorter solution

```
boolean e = a[i].equals(key);
```

is also (even syntactically) prohibited. See Appendix A for a more thorough description of the constraints of our programming language compared to Java.

The method *main* of the program creates an array, fills it with values, updates it, and calls the method *search* in the usual way. It also calls the method `System.out.println` of the Java Standard API. This method has to be declared with the help of the dummy classes

```
package java.lang;
import java.io.*;
public class System
{
    public static PrintStream out;
}

package java.io;
public class PrintStream
{
    public void println(boolean b)
    public void println(int i)
    public void println(char c)
    public void println(String s)
    public void println()
}
```

Type-checking the class *Record* creates a new theory *Record* in the same package as the class. Right-clicking this theory from the tab *Symbols* and selecting *Print Declaration* displays

```
theory Record uses java.lang.String, Base
```

```

{
  Record: TYPE =
    [#key: java.lang.String.String, value: Base.int#];
  null: Record;
  nullArray: ARRAY Base.int OF Record;
  length: (ARRAY Base.int OF Record) -> Base.nat;
}

```

which introduces the following entities:

- a logical record type *Record* which contains one field for each object variable in class *Record*; the specification language considers program variables of object type *Record* to hold values of the logical type *Record*;
- a constant *null* representing the logical counterpart of the `null` pointer of type *Record*.
- a type *nullArray* representing the logical counterpart of the `null` pointer of type `Record[]`.
- a function *length* representing the logical counterpart of the program selector `.length` when applied to arrays of type `Record[]`.

As can be seen from the declaration, the logical counterparts of program arrays have logical type *ARRAY Base.int OF ...* where *Base.int* (type *int* in theory *Base*) denotes the logical counterpart of the program type *int*.

The program is now specified with the help of the following local theory

```

/*@
theory uses Base, Record, java.lang.String
  String: TYPE = java.lang.String.String;
  notFound:
    PREDICATE (ARRAY Base.int OF Record.Record,
              Base.int, String) =
    PRED (a:ARRAY Base.int OF Record.Record,
         n: Base.int, key: String):
    FORALL (i:INT):
      0 <= i AND i < n => a[i].key /= key;

@*/
class Record { ... }

```

which introduces a predicate *notFound* to describe that in an array *a* of records, all positions less than *n* hold records whose keys are different from *key*.

The program method *search* can now be specified as

```
public static int search(Record[] a, String key) /*@
  requires VAR a /= Record.nullArray;
  ensures
    (LET result=VALUE@NEXT,
     n = Record.length(VAR a)
    IN
     IF result = -1 THEN
       notFound(VAR a, n, VAR key)
     ELSE
       0 <= result AND result < n AND
       notFound(VAR a, result, VAR key) AND
       VAR a[result].key = VAR key
     ENDIF);
  @*/
{ ... }
```

The method's precondition states that *search* must not be called with the array `null` as argument and that its result is either -1 (indicating that the given key has not been found in the array) or that its result is the smallest index of the array such that the corresponding record has the denoted key. The specification makes use of local logical variables *result* and *n* representing the return value of the method and the length of the method parameter *a*.

The core loop of the method's body can be annotated as

```
for (int i=0; i<n; i++)
  /*@
    invariant VAR a /= Record.nullArray
      AND VAR n = Record.length(VAR a)
      AND 0 <= VAR i AND VAR i <= VAR n
      AND notFound(VAR a, VAR i, VAR key);
    decreases VAR n - VAR i;
  @*/
{ ... }
```

to give a suitable invariant and termination term.

The state after type-checking the annotated program is shown in Figure 3.4.

Type checking the annotated program class generates two type checking conditions. The first one is generated for the local theory of *Record* and printed as

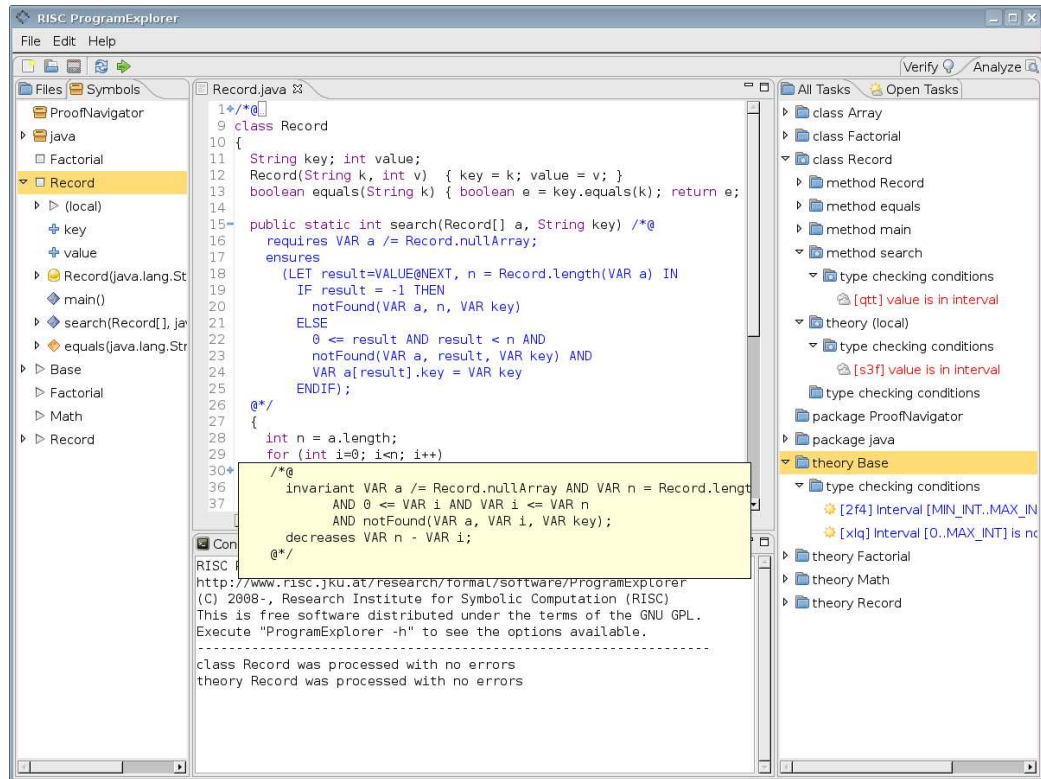


Figure 3.4: Searching for Records

```

Task (Record.java:1:1:6:44) :
  [s3f] value is in interval
Annotation line:
  FORALL(i:INT): 0 <= i AND i < n =>
    ... a[i].key /= key;
    ^
Status: new
Type: verify type checking condition
Goal formula:
  0 <= i AND i < n => MIN_INT <= i AND i <= MAX_INT

```

It has to make sure that variable i is an element of type *Base.int* (the base type of the array type of a). The second is generated for the method *search* and printed as

```

Task (Record.java:15:52:6:27) :
  [qtt] value is in interval

```

```

Annotation line:
    notFound(VAR a, n, VAR key)
                ^

Status: new
Type: verify type checking condition
Goal formula:
    old a /= Record.nullArray AND result = value@next
    AND n = Record.length(var a) =>
    (MIN_INT <= 0 OR MIN_INT <= n) AND
    (MAX_INT <= MAX_INT OR n <= MAX_INT)

```

It has again to make sure that variable i is an element of type *Base.int* (the type of the parameter n of *notFound*). Right-clicking on task folder *class Record* and selecting the menu option *Execute Task* automatically discharges these tasks.

3.3 Failed Tasks and Interactive Proofs

The previous sections dealt only with tasks (type-checking conditions) that could be automatically solved by the integrated decision procedure. In this section, we will discuss what happens if the automatic decision procedure does not succeed.

The first kind of failure can be demonstrated by the theory

```

theory Proving1
{
  // type-checking task can be proved unsatisfiable
  a: INT = 1;
  b: INT = 0;
  T: TYPE = [a..b];
}

```

which attempts to erroneously define an empty type T (by an interval of the integers whose lower bound is bigger than the upper bound). Type-checking this theory generates the task

```

Task(Proving1.theory:6:13):
  [iai] Interval [a..b] is not empty
Status: new
Type: verify type checking condition
Goal formula: a <= b


```

If we select *Execute Task*, the task is now printed as


```

Task(Proving1.theory:6:13):
  [ia] Interval [a..b] is not empty
Status: failed
      (deemed unsolvable by decision procedure)
Type: verify type checking condition
Goal formula: a <= b

```

and tagged in the task tree with the icon  (which indicates task status “failed”). The reason is that the RISC ProgramExplorer, after unsuccessfully trying to perform the task with goal $a \leq b$, tries to perform the *opposite* of the task, i.e. the task with the negated goal formula $a > b$. Since this succeeded, the original task can be considered as impossible, i.e. it receives status “failed”. A task with such a status indicates an error in the corresponding theory/program written by the user.

The second kind of failure can be demonstrated by the theory

```

theory Proving2
{
  // type-checking task cannot be solved
  a: INT;
  b: INT;
  T: TYPE = [a..b];
}

```

which defines a type as a subrange of the integers with unspecified bounds a and b . Type checking this theory yields the task

```

Task(Proving2.theory:6:13):
  [ia] Interval [a..b] is not empty
Status: new
Type: verify type checking condition
Goal formula: a <= b

```

If we select *Execute Task*, the task is now printed as

```

Task(Proving2.theory:6:13):
  [ia] Interval [a..b] is not empty
Status: open (interactive proof required)
Type: verify type checking condition
Goal formula: a <= b

```

and tagged in the task tree with the icon  (which indicates task status “open”). In this case, neither the task nor its opposite could be automatically solved, i.e.

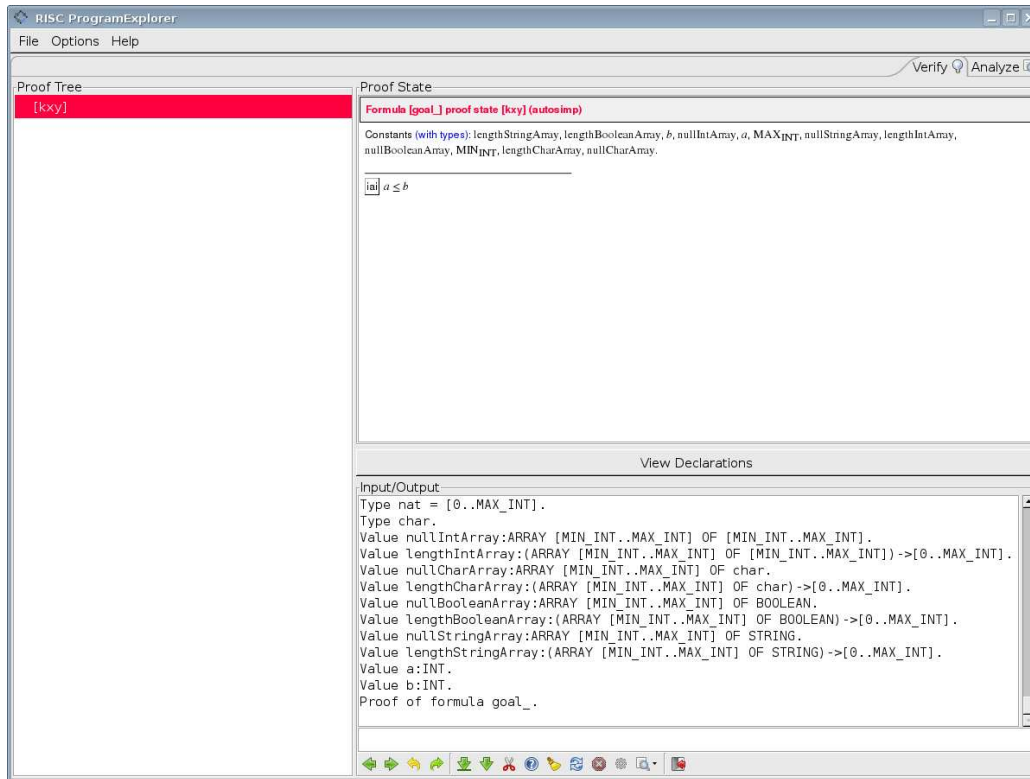




Figure 3.5: An Interactive Proof

the automatic decision procedure has failed. The only chance to solve this task (if any) is now by a computer-assisted *interactive* proof. If we select *Execute Task* for a second time, the RISC Program Explorer switches to the view “Verify” depicted in Figure 3.5.

This view is essentially a view on the RISC ProofNavigator [8, 5], a computer-assisted interactive proving assistant integrated into the RISC Program Explorer. The task generated by the RISC Program Explorer has been translated into a proving problem of the RISC ProofNavigator which can now be attempted with the aid of human intelligence. However, in our example, this attempt is in vain: the proof situation does not contain any assumptions, pressing the button *View Declarations* displays the corresponding declarations shown in Figure 3.6. Since a and b are undefined, there is no chance of completing the proof successfully. We therefore press the button *Quit Proof*  and return (after a confirmation) to the “Analyze” view with task status still unchanged as indicated by the icon .

```

Declarations
 STRING ∈ type
 MIN_INT ∈ Z = -2147483648
 MAX_INT ∈ Z = 2147483647
 int ∈ type = [MIN_INT, MAX_INT]
 nat ∈ type = [0, MAX_INT]
 char ∈ type
 nullIntArray ∈ array [MIN_INT, MAX_INT] of [MIN_INT, MAX_INT]
 lengthIntArray ∈ (array [MIN_INT, MAX_INT] of [MIN_INT, MAX_INT]) → [0, MAX_INT]
 nullCharArray ∈ array [MIN_INT, MAX_INT] of char
 lengthCharArray ∈ (array [MIN_INT, MAX_INT] of char) → [0, MAX_INT]
 nullBooleanArray ∈ array [MIN_INT, MAX_INT] of B
 lengthBooleanArray ∈ (array [MIN_INT, MAX_INT] of B) → [0, MAX_INT]
 nullStringArray ∈ array [MIN_INT, MAX_INT] of STRING
 lengthStringArray ∈ (array [MIN_INT, MAX_INT] of STRING) → [0, MAX_INT]
 a ∈ Z
 b ∈ Z
 goal ≡ a ≤ b

```

Figure 3.6: The Declarations

It is unlikely (but nevertheless possible) that the type-checker generates tasks that cannot be solved by the integrated automatic decision procedure but can be solved by an interactive proof. If this should be the case, the proof remains *persistent* across multiple invocations of the RISC ProgramExplorer; it can be later displayed (menu option *Print Status Evidence*) and also replayed again (menu option *Execute Task*). Menu option *Reset Task* erases the proof and returns the task to state “new”.

If there is no ongoing interactive proof, the user may manually switch to the view “Verify” and enter declarations and commands of the RISC ProofNavigator. The use of the software is then essentially the same as in the standalone version of the RISC ProofNavigator.

Thus the RISC ProofNavigator is already fully integrated into the task solution framework of the RISC ProgramExplorer; while this is not of major importance for the purpose of verifying type checking conditions, it will become essential in the future for verifying tasks originating from the envisioned more general reasoning and verification problems.

References

- [1] ANTLR v3 Parser Generator, 2010. <http://www.antlr.org>.
- [2] Clark Barrett. CVC Lite Homepage, April 2006. New York University, NY, <http://www.cs.nyu.edu/acsys/cvcl>.
- [3] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13–17, 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer, 2004.
- [4] The Java Modeling Language (JML), 2010. <http://www.jmlspecs.org>.
- [5] The RISC ProofNavigator, 2010. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, <http://www.risc.jku.at/research/formal/software/ProofNavigator>.
- [6] Wolfgang Schreiner. A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2008. <http://www.risc.jku.at/people/schreine/papers/ProgramCalculus2008.pdf>.
- [7] Wolfgang Schreiner. Understanding Programs. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, July 2008. <http://www.risc.uni-linz.ac.at/people/schreine/papers/Understanding2008.pdf>.
- [8] Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, 21(3):277–291, 2009.

Appendix A

Programming Language

In this appendix, we sketch the language that is used in the RISC ProgramExplorer for describing programs (its formal syntax is described in Appendix F.1). This programming language can in the following sense be considered as a “MiniJava”, i.e. as (a variant of) a subset of Java: Assume that a program can be parsed and type-checked by the RISC ProgramExplorer without error. If this program can be also compiled by the Java compiler without error, the execution of the generated target code behaves as specified by Java¹.

Deviations In detail, MiniJava has the following *deviations* compared to Java (such that a program that can be parsed and type-checked by the RISC ProgramExplorer cannot be compiled in Java):

Visibility Modifiers The modifiers `public`, `protected`, and `private` are recognized but ignored; in fact MiniJava treats all entities as if declared with modifier `public`. Consequently, if a MiniJava program violates the specified access constraints, it cannot be compiled by a Java compiler.

Constraints The following items describe *constraints* of MiniJava (such that a program that can be compiled with Java cannot be parsed or type-checked by the RISC ProgramExplorer)².

¹It should be noted that “MiniJava” was designed as a simple imperative programming language whose concrete syntax and semantics is immediately familiar to many programmers and can thus represent the basis for understanding formal specifications of imperative languages. It is not designed as the starting point of the specification of full Java.

²Actually, only the major constraints are listed (more constraints can be detected by investigating the syntax specified in Appendix F.1).

Inheritance MiniJava does not support inheritance; every class denotes an object type that is incompatible with the object type of any other class.

Interfaces MiniJava does not support interfaces.

Method Calls A method call with a return value may only appear on the right side of a variable initialization or of a variable assignment, not as an expression within another expression.

Throwing Exceptions An exception can be only thrown by a statement of form `throw new Exception(string)` where *string* denotes a string literal, respectively a value of type `java.lang.String`.

References The type system of MiniJava restricts a program such that every object variable can be considered to hold an object *value* itself (rather than a *reference* to the actual object value) which considerably simplifies reasoning about objects. More concretely, this restriction ensures that two different references cannot denote the same object (and so an update of the object value via one reference cannot affect the object value denoted by any other reference). In particular,

- a variable of an object type may only receive the result of a constructor call or of a method call;
- a `return` statement may only return (the result of) a constructor call, a method call, or an object path `v . . .`, where `v` denotes a local variable of the current method;
- a method/constructor call may receive as an argument of an object type only (the result of) a constructor call, a method call, or an object path `v . . .` where `v` denotes a local variable or a method parameter that is not the base of an object path which appears as another argument in the same method/constructor call.

Here an object path `v . . .` denotes the variable `v`, possibly trailed by a sequence of selectors of the form `.var` (an object variable selector) or `[exp]` (an array index selector).

Java Classes The RISC ProgramExplorer does not itself provide/implement the classes of the Java API (also not the classes `java.lang.String` used for character strings or `java.lang.System` used for standard input/output); if such classes are used in programs, the programmer must provide corresponding class stubs in (a subpackage of) a package `java` within the package hierarchy seen by the RISC ProgramExplorer (see Section C).

Specification Comments The contents of program comments of the form `/*@`
`...@*/` and `//@` `...` are interpreted as formal program specifications; the lan-
guage of these specifications are explained in the following section.

Appendix B

Specification Language

In this appendix, we describe the language that is used in the RISC ProgramExplorer for specifying programs. This language is based upon the logic language of the RISC ProofNavigator as explained in Section B.1. With this language whose formal syntax is described in Appendix F.2, theories can be constructed as described in Sections B.2 and B.3. With the help of theories, we may specify programs as described in Sections B.4, B.5, and B.6.

B.1 Logic Language

The logic language of the RISC ProgramExplorer is based on the language of the RISC ProofNavigator [5, 8]. In the following, we only describe the differences respectively extensions.

B.1.1 Declarations

The logic language allows to introduce by declarations

- type constants,
- object/function/predicate constants,
- constants denoting formulas (to be proved) and axioms (assumed true).

While the language of RISC ProofNavigator considers both terms and formulas as elements of the syntactic domain (*value*) *expression* (formulas are just expressions

denoting a Boolean value, mismatches between terms and formulas are detected by the type checker), the RISC ProgramExplorer decomposes *expression* into two syntactic domains *term* and *formula* (which already enables the parser to detect mismatches). Nevertheless, on the semantic level predicates are just considered as functions whose result is a Boolean value.

Object/function/predicate constants can now be defined as follows:

ident* : *type* = *term This definition introduces an object constant *ident* defined as *term*. If *type* denotes the type *BOOLEAN*, *ident* can be used as a 0-ary predicate constant.

ident* : *type* <=> *formula This definition introduces a 0-ary predicate constant *ident*; here *type* must denote the type *BOOLEAN*.

ident* : *type* = LAMBDA (*params*) : *term This definition introduces a new function constant *ident* which is defined as a function that binds its concrete arguments to the parameters *params* and returns as a result the value of *term* in the environment set up by the binding. Here *type* must denote a corresponding function type. If the domain of *type* denotes the type *BOOLEAN*, *ident* can be considered as a predicate constant.

ident* : *type* = PRED (*params*) : *formula This definition introduces a predicate constant *ident* which is defined as a predicate that binds its concrete arguments to the parameters *params* and returns as a result the truth value of *formula* in the environment set up by the binding. Here *type* must denote a corresponding function type whose domain denotes the type *BOOLEAN*; here the type *PREDICATE* (see the following subsection) is recommended.

B.1.2 Types

The RISC ProgramExplorer introduces the additional types

```
STRING
PREDICATE (types)
```

STRING is an unspecified type which plays a role in the mapping of program types to logical types, see the next subsection.

PREDICATE (*types*) is a synonym of

```
(types) -> BOOLEAN
```

The use of this type syntactically simplifies the definitions of predicate constants (see the previous subsection).

B.1.3 Mapping Program Types to Logical Types

The subsequent subsections describe how a logical formula may refer to the values of program variables. This requires the mapping of program values to logical values and of program types to logical types.

This mapping is based on the automatically generated theory `Base` in the unnamed top-level package:

```
theory Base
{
  MIN_INT: INT = -2147483648;
  MAX_INT: INT = 2147483647;
  int: TYPE = [MIN_INT..MAX_INT];
  nat: TYPE = [0..MAX_INT];
  char: TYPE;
  nullIntArray: ARRAY int OF int;
  lengthIntArray: (ARRAY int OF int) -> nat;
  nullCharArray: ARRAY int OF char;
  lengthCharArray: (ARRAY int OF char) -> nat;
  nullBooleanArray: ARRAY int OF BOOLEAN;
  lengthBooleanArray: (ARRAY int OF BOOLEAN) -> nat;
  nullStringArray: ARRAY int OF STRING;
  lengthStringArray: (ARRAY int OF STRING) -> nat;
}
```

In detail, program types are mapped to logical types as follows:

boolean The program type `boolean` is mapped to the logical type `BOOLEAN`.

int The program type `int` is mapped to the logical type `Base.int` which is a subrange of `INT`.

char The program type `char` is mapped to the logical type `Base.char` (which is currently unspecified).

class C Every class `C` is automatically translated to a theory `C` that resides in the same package as the class. This theory contains a record type `C` that contains one field for every object variable of the class. The program type `C` is mapped to this record type `C`.

Additionally the automatically generated theory contains the constant

```
    null: C;
```

that represent the program value `null` of type `C`.

Character strings If the user provides a program class `java.lang.String`, string literals in programs are considered as values of this class which is mapped to the logical type `java.lang.String.String`.

However, if the user does not provide such a class, string literals in programs are considered as values of a pseudo-type that is mapped to the logical type `STRING`.

$T[]$ The program type $T[]$ is mapped to the type `ARRAY Base.int OF T'` where T' is the logical type to which the program type T is mapped.

For every class C , the automatically generated theory contains the constants

```

nullArray: ARRAY Base.int OF C;
length: (ARRAY Base.int OF C) -> Base.nat;

```

that represent the program value `null` of the array type $C[]$ respectively the field access operator `.length` for arrays of type $C[]$.

For the program types `boolean[]`, `int[]`, `char[]`, and `String[]`, the theory `Base` contains constants that represent the `null` values of these array types respectively the field access operator `.length` for arrays of these types.

B.1.4 Program Variables

Synopsis

```

OLD var
VAR var

```

Description Within the context of a state predicate of a specification (e.g. a method precondition), both `OLD var` and `VAR var` refer to the “current” state of the program variable *var*.

Within the context of a state relation of a specification (e.g. a method postcondition or loop invariant), `OLD var` refers to the value of the program variable *var* in the prestate of the specified execution; `VAR var` refers to the value of *var* in the corresponding poststate.

More specifically, `OLD var` respectively `VAR var` denotes the logical value to which the value of the program variable is mapped. Therefore the type of `OLD var` respectively `VAR var` is the logical type to which the type of the program variable is mapped.

Pragmatics A reference to a program variable *var* in a formula is tagged with keyword `OLD` or `VAR` to explicitly distinguish it from a reference to a logical variable; we thus emphasize that its value actually results from mapping a program value to a logical value.

We choose the keywords and their interpretations in both state conditions and state relations in order to minimize the confusion of programmers:

- If there is a corresponding state relation (e.g. method postcondition), we may prefer in the precondition the use of `OLD var` since we thus refer in both the precondition and the postcondition to the same value in the same way.

However, if there is no corresponding state relation, the syntax `OLD var` in a state condition looks awkward since the condition only refers to a single state: here we may prefer `VAR var`.

- In a loop invariant (which also denotes a state relation), `VAR var` refers to the value of the variable after the execution of the loop body, while `OLD var` refers to the state of the variable in the prestate of the loop. If the invariant does not refer to the prestate (as it is often the case), the invariant can be thus expressed in terms of `VAR var` only.

B.1.5 Program States

The logic language introduces a new kind of values called *states* with corresponding types, constants, functions, and predicates.

B.1.5.1 Type `STATE`

Synopsis

```
STATE  
STATE (type)
```

Description A type of this family denotes the set of states that may result from the execution of a command. The type `STATE` indicates that the execution of the command must not return a value (i.e. that the command is executed within a function of result type `void`); the type `STATE (type)` indicates that the command may return a value of the denoted *type*.

B.1.5.2 State Constants

Synopsis

```
NOW
NEXT
```

Description Within the context of a state predicate of a specification (e.g. a method precondition), both constants `NOW` and `NEXT` denote the “current” state.

Within the context of a state relation of a specification (e.g. a method postcondition or loop invariant), the constant `NOW` denotes the prestate of the specified execution while the constant `NEXT` denotes the corresponding poststate.

Pragmatics To simplify the semantics, `NEXT` is also defined in the context of a state predicate.

In a loop invariant, `NOW` refers to the prestate of the loop, while `NEXT` refers to the poststate of the loop body.

B.1.6 State Functions

Synopsis

```
VALUE@state
MESSAGE@state
```

Description These functions are evaluated over *state* whose type is of form `STATE` or `STATE(result)`.

If *state* results from the execution of `return value`, the term `VALUE@next` refers to (the logical mapping of) *value*. The type of *state* must be of form `STATE(result)`; the type of `VALUE@next` is *result* (which is the logical mapping of the type of *value*).

If *state* results from the execution of `throw new exception(message)`, the term `MESSAGE@next` refers to (the logical mapping of) *message*. Its type is the logical mapping of the program type `java.lang.String` (which must be the type of *message*).

B.1.6.1 State Predicates

Synopsis

```
EXECUTES@state
CONTINUES@state
BREAKS@state
RETURNS@state
THROWS@state
THROWS(exception)@state
```

Description These predicates are evaluated over *state* whose type is of form STATE or STATE(*result*):

- EXECUTES@*state* is true if and only if none of the following four predicates is true.
- CONTINUES@*state* is true if and only if *state* results from the execution of `continue`.
- BREAKS@*state* is true if and only if *state* results from the execution of `break`.
- RETURNS@*state* is true if and only if *state* results from the execution of `return of return value`.
- THROWS@*state* is true if and only if *state* results from the execution of `throw new exception(message)` (for any *exception* type and string *message*).
- THROWS(*exception*)@*state* is true if and only if *state* results from the execution of `throw new exception(message)` (for any character string *message*).

B.1.6.2 State Pair Predicates

Synopsis

```
READONLY
WRITESONLY var, ...
```

Description These formulas are evaluated in the context of a pair of execution states (e.g. a method postcondition or loop invariant) called the “prestate” and the “poststate” of the execution.

READONLY is true if and only if the value of every program variable is in the poststate of the execution the same as in the prestate.

WRITESONLY *name, ...* is true if and only if the value of every program variable that is not listed in “*var, ...*” is in the poststate of the execution the same as in the prestate.

B.2 Theory Definitions

Synopsis

```
package package ;
import package.* ;
import package.theory ;
...
theory theory uses theories
{ declarations }
```

Description A theory definition introduces by a list of *declarations* a “theory” i.e. a collection of logic entities that may be used in other theories or for the specification of programs.

The clause `theory theory` states the name of the theory as *theory*. The optional clause `package package` states that the new theory resides in *package* and may be referenced elsewhere by the long name *package.theory*; likewise any *entity* introduced by *declarations* may be referenced elsewhere by the long name *package.theory.entity*. If the `package` clause is omitted, the theory resides in the unnamed top-level package.

An `import` clause imports theories from other packages such that they may be referenced from the current theory not only by their long names of form *package.theory* by also by their short names of form *theory*. A clause

```
import package.*;
```

imports all theories from *package*; a clause

```
import package.theory;
```

imports from *package* only *theory*. If multiple *package.** import theories with the same name, these theories can be only referenced by their long name unless one of the packages is also imported as *package.theory*; then this theory can also be referenced by the short name. Multiple *package.theory* imports of different theories with the same short name *theory* are prohibited.

Every theory referenced by declarations in the current theory must be listed in the clause *uses theory, ...*, either by the long name of the theory or, if the theory was imported, by its short name.

Pragmatics A theory with long name *package.theory* must reside in a file *theory.theory* in a subdirectory *package* of a directory that is considered as a root of the package hierarchy. The name *package* may have form *p1.p2...pn*; the corresponding directory path is then *p1/p2/.../pn*.

The clause *import ...* is modeled after the semantics of the corresponding Java clause but imports theories rather than classes.

The clause *uses theory, ...* was introduced to simplify the computation of dependencies between classes and theories; in a subsequent version of the language, this clause may be well dropped.

B.3 Class Specifications

Synopsis

```

/*@
  import package.*;
  import package.theory;
  ...
  theory uses theory, ...
  { declarations }
@*/
classheader { ... }

```

Descriptions A class specification introduces by a list of *declarations* the “local theory” of a class i.e. a theory of those entities that may be referenced by their short names in the specification of methods, loops, and commands of the class (the entities introduced in other theories may be always referenced by the long name *package.theory.entity*). If a class has no such specification, the

local theory is empty; the specifications in this class may therefore only refer to entities introduced in other theories.

An `import` clause imports theories from other packages, see Section B.2.

Every theory referenced by declarations in the local theory (respectively by the specifications of methods, loops, statements in the current class) must be listed in the clause `uses theory, ...`, either by the long name of the theory or, if the theory was imported, by its short name.

Pragmatics The clause `import ...` is modeled after the semantics of the corresponding Java clause but imports theories rather than classes.

The clause `uses theory, ...` was introduced to simplify the computation of dependencies between classes and theories; in a subsequent version of the language, this clause may be well dropped.

B.4 Method Specifications

Synopsis

```

methodheader
/*@
    assignable vars ;
    signals exceptions ;
    requires formula ;
    diverges formula ;
    ensures formula ;
    decreases term ;
@*/
{ statements }

```

Description This specification describes the observable behavior of a method (class method, object method, or constructor) by the following clauses:

assignable vars This optional clause lists the variables *vars* that are visible in the scope of the declaration of the method (object and class variables of the current class, class variables of other classes, respectively variables that represent components of such variables, but *not* parameters and local variables of the method) and whose values may be changed by the execution of the method.

If the clause is omitted, the method must not modify any variable that is visible in the scope of the method declaration.

signals exceptions This optional clause lists the types of the *exceptions* that may be thrown by the execution of the method (excluding “runtime exceptions” such as “division by zero” that may be thrown by the execution of primitive operations).

If the clause is omitted, the method must not throw any exception.

requires formula This optional clause states that it is only legal to call the method in a state (the method’s “prestate”) that satisfies the given *formula*.

If the clause is omitted, the *formula* is considered as “true”, i.e. it is legal to call the method in any state.

diverges formula This optional clause states that the method will terminate (by returning normally or by throwing an exception) when called in any legal state that satisfies also the negation of *formula* (i.e. the method is allowed to run forever when called in any legal state that satisfies *formula*).

If the clause is omitted, the *formula* is considered as “false”, i.e. the method must terminate when called in any legal prestate.

ensures formula This optional clause states that, for every legal prestate of the method, every state in which the method terminates is only legal if it is related to the method’s prestate by *formula*.

If the clause is omitted, the *formula* is considered as “true”, i.e. the method may terminate with any poststate.

decreases term This optional clause states that, for every call of the method in a legal state, the value of *term* denotes a non-negative integer number which is decreased in every (directly or indirectly) recursive call of the method (such that chain of recursive method calls must eventually end).

If the clause is omitted, no default is assumed.

Pragmatics This specification is in essence modeled after the “light-weight” specification format of JML, the Java Modeling Language [4]; however, a fixed order is required and specific default values for missing clauses are given. Furthermore, the specification follows (not precedes) the method’s declaration header to emphasize that the specification appears in the scope of the parameters of the method.

If the clause `decreases term` is missing in a (directly or indirectly recursive) method, the termination of the method can probably not be proved.

B.5 Loop Specifications

Synopsis

```

while (exp)                for (forheader)
/*@                        /*@
  invariant formula ;      invariant formula ;
  decreases term ;         decreases term ;
@*/                          @*/
body                          body

```

Description The optional clause `invariant formula` states that the state in which the loop checks the value of `exp` for the first time (the loop’s “prestate”) is related by `formula`

1. to the loop’s prestate itself and
2. to every state that arises immediately after the execution of the loop’s `body` (the body’s “poststate”).

If the clause is omitted, the formula is assumed to be “true”.

The optional clause `decreases term` states that

1. the value of `term` in the loop’s prestate and in every poststate of the loop’s `body` denotes a non-negative integer number, and that
2. the value of `term` immediately before the execution of the loop’s `body` is greater than the value of `term` after the execution of the loop’s `body`.

Consequently the loop cannot perform an infinite number of iterations.

Pragmatics It should be noted that the formulation of the invariant above relates the loop’s prestate to the body’s poststate which, due to the existence of state functions and state predicates in the formula language, may be considered as different from the prestate of the subsequent loop iteration, respectively, if the loop terminates, from the loop’s poststate. For instance, if the body executes a `break` statement, the loop’s prestate is related to the body’s poststate by the formula `BREAKS@NEXT` but to the loop’s poststate by `EXECUTES@NEXT`. The first formula is more precise since it describes that the loop terminates from the execution of the loop body which the second formula does not. Our formulation therefore allows to express stronger invariants.

B.6 Statement Specifications

Synopsis

```
//@ assert formula ';'
statement
```

Description The specification states that immediately before the execution of *statement* (i.e. in the statement's "prestate") *formula* holds.

Pragmatics The specification creates an additional proof obligation but then also more information for the verification of *statement* and its successors.

Appendix C

Program Invocation

The shell script `ProgramExplorer` is the main interface to the program i.e. the program is typically started by executing

```
ProgramExplorer &
```

However, if the script is copied/renamed/linked to `ProofNavigator` and executed as

```
ProofNavigator &
```

the program starts with a standalone interface to the RISC ProofNavigator [5] (which is part of the RISC ProgramExplorer).

Invoking the script as

```
ProgramExplorer -h
```

gives the following output which lists the available startup options and the environment variables used:

```
RISC ProgramExplorer Version 0.3 (April 8, 2010)
http://www.risc.jku.at/research/formal/software/ProgramExplorer
(C) 2008-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "ProgramExplorer -h" to see the options available.
```

```
-----
Usage: ProgramExplorer [OPTION]...
OPTION: one of the following options:
```

`-h, --help`: print this message.
`-cp, --classpath [PATH]`: directories representing
the top package.

Environment Variables:

`PE_CLASSPATH`:
the directories (separated by ":") representing
the top package
(default the current working directory)

`PE_CVCL`
the command for executing the cvcl checker
(default "cvcl")

`PE_JAVAC`
the command for compiling java programs
(default "javac")

`PE_JAVA`
the command for executing java programs
(default "java")

`PE_CWD`
the directory used for compiling/executing
(default the current working directory)

`PE_MAIN`
the name of the main class of the program
(default "Main")

The command accepts the following startup options:

- h, --help** With this option, the description shown above is printed and the program terminates.
- cp, --classpath *Path*** This option expects as *Path* a sequence of directories separated by the colon character ":". The program considers these directories to jointly represent the root of the package hierarchy; by default, the current working directory (path ".") alone represents the root. The various directories in *Path* must not have different class files (extension `.java`), theory files (extension `.theory`), or subdirectories of the same name.

The program uses the values of the following environment variables.

PE_CLASSPATH If the program is started without the command line option `-cp/--classpath Path`, the value of this variable is considered as the *Path*, see the description of the option given above.

PE_CVCL The value of this environment variable is considered as the path to the executable of the Cooperating Validity Checker (CVC) Lite version 2.0; by default, the path `cvcl` is assumed.

PE_JAVAC The value of this environment variable is considered as the path to the executable of the Java compiler; by default, the path `javac` is assumed.

PE_JAVA The value of this environment variable is considered as the path to the executable of the Java runtime environment; by default, the path `java` is assumed.

PE_CWD The value of this environment variable is considered as the path of the directory used for compiling/executing respectively creating subdirectories; by the default the current working directory “.” is used.

PE_MAIN The value of this environment variable is considered as the name of the main class of the program to be compiled and executed; by default the value `Main` is used.

Appendix D

Program Installation

The installation of the program is thoroughly described in the files `README` and `INSTALL` of the distribution; we include these files verbatim below.

D.1 `README`

README

Information on the RISC ProgramExplorer.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
Copyright (C) 2008-, Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria, <http://www.risc.jku.at>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

RISC ProgramExplorer

<http://www.risc.jku.at/research/formal/software/ProgramExplorer>

This is the RISC ProgramExplorer, an interactive program reasoning environment that is under development at the Research Institute for Symbolic Computation (RISC). This software is freely available under the terms of the GNU General Public License, see file `COPYING`.

The current version is a first demonstrator skeleton that incorporates the overall technological and semantic framework (programming language and formal specification language) and integrates the RISC ProofNavigator as an interactive proving assistant. Work is ongoing to provide this skeleton with the envisioned program reasoning capabilities.

The RISC ProgramExplorer runs on computers with x86-compatible processors under the GNU/Linux operating system. For learning how to use the software, see the file "main.pdf" in directory "manual"; examples can be found in directory "examples".

Please send bug reports to the author of this software:

Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
<http://www.risc.jku.at/home/schreine>
 Research Institute for Symbolic Computation (RISC)
 Johannes Kepler University
 A-4040 Linz, Austria

Third Party Software

 The RISC ProgramExplorer uses the following open source programs and libraries. Most of this is already included in the RISC ProgramExplorer distribution, but if you want or need, you can download the source code from the denoted locations (local copies are available on the RISC ProgramExplorer web site) and compile it on your own. Many thanks to the respective developers for making this great software freely available!

CVC Lite 2.0
<http://www.cs.nyu.edu/acsys/cvcl>

This is a C++ library/program for validity checking in various theories.

The RISC ProgramExplorer currently only works with CVCL 2.0, not the newer CVC3 available from <http://www.cs.nyu.edu/acsys/cvc3>. To download the CVCL 2.0 source, go to the RISC ProofNavigator web site (URL see above), Section "Third Party Software", and click on the link "CVCL 2.0 local copy".

RIACA OpenMath Library 2.0
<http://www.riaca.win.tue.nl/products/openmath/lib>

This is a library for converting mathematical objects to/from the OpenMath representation.

Go to the link "OMLib 2.0" and then "Downloads".
 Download one of the "om-lib-src-2.0-rc2.*" files.

General Purpose Hash Function Algorithms Library
<http://www.partow.net/programming/hashfunctions>

A library of hash functions implemented in various languages.

Go to the link "General Hash Function Source Code (Java)" to download the corresponding zip file.

ANTLR 3.2
<http://www.antlr.org>

This is a framework for constructing parsers and lexical analyzers used for processing the programming/specification language of the RISC ProgramExplorer.

On a Debian 5.0 GNU/Linux distribution, just install the package "antlr3"

by executing (as superuser) the command

```
apt-get install antlr3
```

ANTLR 2.7.6b2

<http://www.antlr.org>

This is a framework for constructing parsers and lexical analyzers used for processing the logic language of the RISC ProofNavigator.

On a Debian 5.0 GNU/Linux distribution, just install the package "antlr" by executing (as superuser) the command

```
apt-get install antlr
```

The Eclipse Standard Widget Toolkit 3.5

<http://www.eclipse.org/swt>

This is a widget set for developing GUIs in Java.

Go to section "Stable" and download the version "Linux (x86/GTK2)" (if you use a 32bit x86 processor) or "Linux (x86_64/GTK 2)" (if you use a 64bit x86 processor).

Mozilla Firefox 3.* or SeaMonkey 2.* (or higher)

<http://www.mozilla.org>

See the question "What do I need to run the SWT browser in a standalone application on Linux GTK or Linux Motif?" in the FAQ at <http://www.eclipse.org/swt/faq.php>.

Chances are that the SWT browser will work with the Firefox included in your Linux distribution (but it will *not* work with the Firefox downloaded from the Mozilla site). For instance, on a Debian 5.0 GNU/Linux distribution, just install Firefox by executing (as superuser) the command

```
apt-get install iceweasel
```

If the SWT browser does not work with the Firefox included in your GNU/Linux distribution, go to the page <http://www.mozilla.org/projects/seamoney> to download and install the SeaMonkey 2.* browser instead. You might have to set the environment variable MOZILLA_FIVE_HOME in the "ProgramExplorer" script to "/usr/lib/mozilla".

The GIMP Toolkit GTK+ 2.X (or higher)

<http://www.gtk.org>

This library is required by "Eclipse Linux (x86/GTK2)" and by "Mozilla 1.7.8 GTK2".

On a Debian 3.1 GNU/Linux distribution, the package is automatically installed, if you install the "mozilla-browser" package (see above).

On another GNU/Linux distribution, go to the GTK web package, section "Download", to download GTK+.

Java Development Kit 6 (or higher)

<http://java.sun.com/j2se>

Go to the "Downloads" section to download the Sun JDK 6.

Tango Icon Library 0.8.9

<http://tango-project.org/>

 Go to the section "Base Icon Library", subsection "Download", to download the icons used in the ProgramExplorer.

 End of README.

D.2 INSTALL

 INSTALL

Installation notes for the RISC ProgramExplorer.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
 Copyright (C) 2008-, Research Institute for Symbolic Computation (RISC)
 Johannes Kepler University, Linz, Austria, <http://www.risc.jku.at>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

 Installation

 The RISC ProgramExplorer is available for computers with x86-compatible processors (32 bit as well as 64 bit) running under the GNU/Linux operating system. The core of the RISC ProgramExplorer is written in Java but it depends on various third-party open source libraries and programs that are acknowledged in the README file.

To use the RISC ProgramExplorer, you have three options:

- A) You can just use the distribution, or
- B) you can compile the source code contained in the distribution, or
- C) you can download the source from a Subversion repository and compile it.

The procedures for the three options are described below.

A Note for Users of Other Operating Systems

 The RISC ProgramExplorer currently runs on computers with x86-compatible processors under GNU/Linux only.

If your computer has an x86-compatible processor but runs another operating system (e.g. MS Windows or MacOS), you may setup a virtual machine that runs GNU/Linux as the guest operating system by using some virtualization software

such as the free VirtualBox software (<http://www.virtualbox.org>). Then you can install the RISC ProgramExplorer in the guest operating system and thus also use the software on your computer.

Currently the RISC ProgramExplorer is only a demonstrator; once it becomes generally usable, we will provide a corresponding virtual machine for download from the RISC ProgramExplorer web site.

A) Using the Distribution

We provide a distribution for computers with ix86-compatible processors running under the GNU/Linux operating system (the software has been developed on the Debian 5.0 "lenny" distribution, but any other distribution will work as well). If you have such a computer, you need to make sure that you also have

- 1) A Java 6 or higher runtime environment.

You can download the Sun JRE 6 from
<http://java.sun.com/j2se>

- 2) The Mozilla Firefox or SeaMonkey browser.

On a Debian 5.0 GNU/Linux system, just install the package "iceweasel" by executing (as superuser) the command

```
apt-get install iceweasel
```

On other Linux distributions, first look up the FAQ on

```
http://www.eclipse.org/swt/faq.php
```

for the question "What do I need to run the SWT browser in a standalone application on Linux GTK or Linux Motif?" The RISC ProgramExplorer uses the SWT browser, thus you have to install the software described in the FAQ.

See the README file for further information.

- 3) The GIMP Toolkit GTK+ 2.6.X or higher.

On a Debian 5.0 GNU/Linux system, GTK+ is automatically installed, if you install the Mozilla browser as described in the previous paragraph.

On other Linux distributions, download GTK+ from <http://www.gtk.org>

For installing the RISC ProgramExplorer, first create a directory INSTALLDIR (where INSTALLDIR can be any directory path). Download from the website the file

```
ProgramExplorer-VERSION.tgz
```

(where VERSION is the number of the latest version of the ProofNavigator) into INSTALLDIR, go to INSTALLDIR and unpack by executing the following command:

```
tar xzf ProgramExplorer-VERSION.tgz
```

This will create the following files

```
README          ... the readme file
INSTALL         ... the installation notes (this file)
CHANGES       ... the change history
COPYING        ... the GNU Public License
bin/
```

```

ProgramExplorer ... the main script to start the program
cvcl             ... CVC Lite, a validity checker used by the software.
doc/
  index.html    ... API documentation
examples/
  README       ... short explanation of examples
  *.theory     ... some example theories
  *.java       ... some example program specifications
lib/
  *.jar        ... Java archives with the program classes
  swt32/      ... SWT for GNU/Linux computers with 32 bit processors
    swt.jar
  swt64/      ... SWT for GNU/Linux computers with 64 bit processors
    swt.jar
manual/
  main.pdf     ... the PDF file for the manual
  index.html   ... the root of the HTML version of the manual
src/
  fmriscl/    ... the root directory of the Java package "fmriscl"
    ProgramExplorer/
      Main.java ... the main class for the RISC ProgramExplorer
    ProofNavigator/
      *.java    ... the sources for the RISC ProofNavigator
    External/
      *.java    ... third-party sources

```

Open in a text editor the script "ProgramExplorer" in directory "bin" and customize the variables defined for several locations of your environment. In particular, the distribution is configured to run on a 32-bit processor. If you use a 64-bit processor, uncomment the line "SWTDIR=\$LIBDIR/swt64" (and remove the line "SWTDIR=\$LIBDIR/swt32").

Put the "bin" directory into your PATH

```
export PATH=$PATH:INSTALLDIR/bin
```

You should now be able to execute

```
ProgramExplorer
```

to run the RISC ProgramExplorer. If you rename/copy/link the script to "ProofNavigator" and execute

```
ProofNavigator
```

the program starts with a standalone interface to the RISC ProofNavigator.

B) Compiling the Source Code

To compile the Java source, first make sure that you have the Java 6 SE development environment installed. You can download the Sun Java 6 SE from

```
http://java.sun.com/javase/downloads/index.jsp
```

Furthermore, on a GNU/Linux system you need also the Mozilla Firefox or SeaMonkey browser, GTK2 and the GIMP toolkit GTK+ installed (see Section A).

Now download the distribution and unpack it as described in Section A.

The RISC ProgramExplorer distribution contains an executable of the validity checker CVC Lite for GNU/Linux computers with x86-compatible processors. To compile the validity checker for other systems, you need to download the CVC

Lite source code (see the README file) and compile it with a C++ compiler. See the CVC Lite documentation for more details.

To compile the Java source code, go to the "src" directory and execute from there

```
javac -cp ".../lib/*:~/lib/swt32/*" fmrisc/ProgramExplorer/Main.java
```

(replace "swt32" by "swt64" on a 64bit system).

You may ignore the warning about "unchecked" or "unsafe" operations, this refers to Java files generated automatically from ANTLR grammars.

Then execute

```
jar cf ../lib/fmrisc.jar \  
  fmrisc/**/*.class fmrisc/**/*.class fmrisc/**/*.class
```

Finally, you have to customize the "ProgramExplorer" script in directory "bin" as described in Section A. You should then be able to start the program by executing the script.

C) Downloading the Source Code from the Subversion Repository

You can now download the source code of any version of the ProofNavigator directly from the ProofNavigator Subversion repository.

To prepare the download, first create a directory SOURCEDIR (where SOURCEDIR can be any directory path).

To download the source code, you need a Subversion client, see http://en.wikipedia.org/wiki/Comparison_of_Subversion_clients for a list of available clients. On a computer with the Debian 5.0 distribution of GNU/Linux, it suffices to install the "svn" package by executing (as superuser) the command

```
apt-get install svn
```

which will provide the "svn" command line client.

Every ProofNavigator distribution has a version number VERSION (e.g. "0.1"), the corresponding Subversion URL is

```
svn://svn.risc.jku.at/schreine/FM-RISC/tags/VERSION
```

If you have the "svn" command-line client installed, execute the command

```
svn export \  
  svn://svn.risc.jku.at/schreine/FM-RISC/tags/VERSION SOURCEDIR
```

to download the source code into SOURCEDIR. With other Subversion clients, you have to check the corresponding documentation on how to download a directory tree using the URL svn://... shown above.

After the download, SOURCEDIR will contain the files of the distribution as shown in Section A; you can compile the source code as explained in Section B.

End of INSTALL.

Appendix E

Task Directories

The system generates in the current working directory (respectively the directory specified by the environment variable `PE_CWD`, see the previous section) two subdirectories named `ProofNavigator` and `.PETASKS.Tag.0`. The directory `ProofNavigator` represents a context directory of the form that is described in the manual of the RISC ProofNavigator [5]; it is used, if the user enters in the *Analyze* view directly (not in the context of any task as described below) commands for the RISC ProofNavigator.

The directory `.PETASKS.Tag.0` represents the persistent store for the task tree of the program; *Tag* is a number that denotes the time when the program was started that created this directory. The content of the directory is a hierarchy of subdirectories that corresponds to the hierarchy of task folders and tasks of the program. Each directory is named `Name.Tag.Cntr` where *Name* is derived from the name of the task folder respectively task, *Tag* denotes the time when the program was started that created this directory, and *Cntr* represents an automatically generated sequence counter.

The content of each task directory depends on the particular kind of the task. Currently the directory may contain the following items:

File `goal` This file contains the log of an attempt to perform the task fully automatically by translating it to a CVCL query and invoking CVCL.

Directory `ProofNavigator` This directory represents a context directory of the RISC ProofNavigator [5] that contains all information related to an attempt to perform the task by a computer-assisted manual proof.

Every directory generated by the RISC ProgramExplorer contains a file `.PEDIR`; if the directory contains also a file `FREED` this indicates that the directory was freed

and may be reused. If a new directory is to be created, it is first attempted to reuse a directory with the same basic *Name* from a previous invocation of the program (as indicated by *Tag*) or a freed directory of the same invocation (as indicated by *Tag* and *FREED*); in both cases, thus previously created RISC ProofNavigator proofs of tasks with the same names will be retained. Otherwise, a new directory is created; if a directory of the desired name already exists, the value of *Cntr* is incremented to yield a new directory name.

Appendix F

Grammars

In this appendix, we describe the concrete syntax of the programming language and of the specification language. The grammars are given in the notation of the parser generator ANTLR v3 [1] used for the implementation of the parser and of the lexical analyzer. Non-determinism in grammatical rules is resolved by extra means provided by ANTLR (in particular semantic predicates) which are omitted from this presentation. On the level of the programming language described in Section F.1, every specification annotation is lexically parsed as a comment yielding the token *ANNOTATION*; the actual grammar of the various kinds of annotations is described in Section F.2 under the header “specifications” by the syntactic domains *unitspec*, *methodspec*, *loopspec*, and *statementspec*. The grammar of theory declarations is specified there by the syntactic domain *theorydecl*.

F.1 Programming Language

```
// -----  
// classes and methods  
// -----  
  
// a compilation unit  
unit : clasdecl ;  
  
// a class declaration  
clasdecl :  
    ( 'package' ';' )?  
    ( 'import' name ( '.' '*' )? ';' )*  
    ( ANNOTATION )?  
    ( 'abstract' | 'final' | 'public' )* 'class' IDENT  
    ( 'extends' name 'implements' names )?
```

```
'{' ( topdecl )* '}'
EOF ;

// a top-level declaration
topdecl :
  objectvar | classvar |
  constructor | objectmethod | classmethod ;

// an object variable, possibly with initialization
objectvar : modifiers typeexp IDENT ( '=' valexp )? ';' ;

// a class variable, possibly with initialization
classvar : modifiers 'static' modifiers typeexp IDENT
  ( '=' valexp )? ';' ;

// a constructor declaration
constructor :
  visibility IDENT '(' ( params )? ')' throwdecls
  ( ANNOTATION )?
  b=block ;

// declaration of an object method
objectmethod :
  modifiers
  ( typeexp | 'void' ) IDENT '(' ( params )? ')' throwdecls
  ( ANNOTATION )?
  b=block ;

// declaration of a class method
classmethod :
  modifiers 'static' modifiers
  ( typeexp | 'void' ) IDENT '(' ( params )? ')' throwdecls
  ( ANNOTATION )?
  block ;

// -----
// statements
// -----

// an execution statement
statement :
  ( an=ANNOTATION )?
  ( emptystat | block | assignment | methodcall | localvar
  | conditional | whileloop | forloop
  | continuestat | breakstat | returnstat | throwstat | trycatch
  | assertion ) ;

// an empty statement
emptystat : ';' ;
```

```

// a statement block
block : '{' ( statement )* '}' ;

// an assignment or method call with return value
assignment : assigncore ';' ;

// the core of an assignment statement
assigncore :
    lval ( '='
        ( valexp | n=name '(' vs=valexps ')'
          | 'new' t=name '(' vt=valexps ')' )
        | '++'
        | '+=' valexp
        | '--'
        | '-=' valexp
        ) ;

// a method call without return value
methodcall : name '(' valexps ')' ';' ;

// a local variable declaration, possibly with initialization
localvar : localvarcore ';' ;

// the core of a local variable declaration
localvarcore :
    ( 'final' )? typeexp IDENT
    ( '=' ( valexp | name '(' valexps ')'
          | 'new' name '(' valexps ')' ) )? ;

// a conditional statement with one or two branches
conditional :
    'if' '(' valexp ')' statement ( 'else' statement )? ;

// a while loop
whileloop : 'while' '(' valexp ')' ( ANNOTATION )? statement ;

// a for loop
forloop :
    'for'
    '(' ( assigncore | localvarcore )? ';'
      ( valexp )? ';' ( assigncore )? ')'
    ( ANNOTATION )?
    statement ;

// a continue statement
continuestat : 'continue' ';' ;

// a break statement

```

```
breakstat : 'break' ';' ;

// a return statement, possibly with return value
returnstat : 'return' ( valexp )? ';' ;

// a throw statement
throwstat : 'throw' 'new' name '(' valexp ')' ';' ;

// a try catch block
trycatch : 'try' block ( 'catch' '(' param ')' =block )+ ;

// an assertion
assertion : 'assert' valexp ';' ;

// -----
// value expressions
// (binding powers taken from "Java in a Nutshell", 5th ed, p.29)
// -----

// value expressions
valexp : valexp3 ;

// disjunctions
valexp3 : valexp4 ( '||' v1=valexp4 )* ;

// conjunctions
valexp4 : valexp8 ( '&&' valexp8 )* ;

// equalities/inequalities
valexp8 : valexp9 ( '==' valexp9 | '!=' valexp9 )* ;

// relations
valexp9 : valexp11
  ( '<' valexp11 | '<=' valexp11 |
    '>' valexp11 | '>=' valexp11 )* ;

// sums and differences
valexp11 : valexp12 ( '+' valexp12 | '-' valexp12 )*
;

// products and quotients
valexp12 : valexp13
  ( '*' valexp13 | '/' valexp13 | '%' valexp13 )* ;

// array creation
valexp13 : 'new' typeexp '[' valexp ']' | valexp14 ;

// unary operators
valexp14 : '+' valexp14 | '-' valexp14
```

```

        | '!' valexp14 | valexp15 ;

// selector operations
valexp15 : valexp16 ( rselector ( r1=rselector )* )? ;

// atoms
valexp16 :
    IDENT | INT | 'true' | 'false' | 'null'
| STRING | CHAR | '(' valexp ')' ;

// -----
// auxiliaries
// -----

// class-level modifiers
modifiers: visibility ( 'final' visibility )? ;

// visibility modifiers
visibility: ( 'private' | 'protected' | 'public' )? ;

// throw declarations
throwdecls: ( 'throws' names )? ;

// a method's parameter list
params : param ( ',' param )* ;

// a method parameter
param : typeexp IDENT ;

// a type expression
typeexp : typeexpbase ( '[' ']' )* ;

// a type expression
typeexpbase : 'int' | 'boolean' | 'char' | name | IDENT ;

// a value expression list
valexps : ( valexp ( ',' valexp )* )? ;

// a name
name : ( IDENT '.' )* IDENT ;

// a sequence of names
names: name ( ',' name )* ;

// a location of a variable
lval : IDENT ( lselector )* ;

// an lvalue selector
lselector : '[' valexp ']' | '.' IDENT ;

```

```

// an rvalue selector
rselector :
  '[' valexp ']' | '.' 'getMessage' '(' ')' | '.' IDENT ;

// -----
// lexical syntax
// -----

IDENT : REALLETTER ( LETTER | DIGIT ) * ;

INT : DIGIT ( DIGIT ) * ;

STRING : '"' ( ~('"' | '\\\ ' | EOL) | ESCAPED ) * '"' ;

CHAR : '\\\ ' ( ~('\\\ ' | '\\\ ' | EOL) | ESCAPED ) '\\\ ' ;

WS : ( ' ' | '\t' | EOL ) ;

ANNOTATION
  ( '///' ( '@' .* EOL | (~'@') => .* EOL ) WS? ) +
  | '/*' ( '@' .* '@*/' | .* '*/' ) ;

fragment
REALLETTER : ( 'a'..'z' | 'A'..'Z' ) ;

fragment
LETTER : ( 'a'..'z' | 'A'..'Z' | '_' ) ;

fragment
DIGIT : ( '0'..'9' ) ;

fragment
EOL : ( '\n' | '\r' | '\f' | '\uffff' ) ;

fragment
ESCAPED : '\\\ '
  ( '\\\ ' | '"' | '\\\ ' | 'n' | 't' | 'b' | 'f' | 'r'
  | ('u' HEX HEX HEX HEX) ) ;

fragment
HEX : '0'..'9' | 'a'..'f' | 'A'..'F' ;

```

F.2 Specification Language

```

// -----
// specifications

```

```

// -----
// a unit specification
unitspec :
  imports 'theory' ( 'uses' names )?
  '{' declarations '}' EOF ;

// a method specification
methodspec :
  ( 'assignable' names ';' )?
  ( 'signals' names ';' )?
  ( 'requires' formula ';' )?
  ( 'diverges' formula ';' )?
  ( 'ensures' formula ';' )?
  ( 'decreases' term ';' )?
  EOF ;

// a loop annotation
loopspec :
  ( 'invariant' formula ';' )?
  ( 'decreases' term ';' )? EOF ;

// a command pre-state annotation
statementspec : ( 'assert' formula ';' )? EOF ;

// a theory declaration
theorydecl :
  ( 'package' name ';' )? imports
  ( 'public' )* 'theory' IDENT ( 'uses' names )?
  '{' ( ( declaration )? ';' )* '}' EOF ;

// -----
// declarations and types
// -----

imports : ( 'import' name ( '.' '*' )? ';' )* ;

declarations : ( ( declaration )? ';' )* ;

declaration :
  IDENT ':'
  ( 'TYPE'
  | 'TYPE' '=' typeExp
  | 'FORMULA' formula
  | 'AXIOM' formula
  | typeExp ( '=' term
              | '=' 'PRED' paramList ':' formula
              | '<=>' formula )?
  ) ;

```

```

typeExp :
  typeExpBase '->' typeExp
| '(' typeExp ( ',' typeExp )+ ')' '->' typeExp
| 'ARRAY' typeExpBase 'OF' typeExp
| typeExpBase ;

typeExpBase :
  name
| 'BOOLEAN'
| 'INT'
| 'NAT'
| 'REAL'
| 'STRING'
| 'STATE' ( '(' typeExp ')' )?
| '[' typeExp ( ',' typeExp )+ ']'
| '[' t0=typeExp ']'
| '# IDENT ':' typeExp ( ',' IDENT ':' typeExp )* '#]'
| 'SUBTYPE' '(' term ')'
| '[' term '..' term ']'
| 'PREDICATE' ( '(' typeExp ( ',' typeExp )* ')' )?
| '(' t0=typeExp ')'
;

// -----
// formulas
// -----

// quantifiers bind weakest
formula :
  'FORALL' paramList ':' formula
| 'EXISTS' paramList ':' formula
| formula10
;

// lets
formula10 returns [ Formula value = null; ] :
  'LET' vdefinition ( ',' vdefinition )* 'IN' formula10
| f=formula20
;

// implications, equivalences, exclusive ors
formula20 :
  formula30 '=>' formula20
| formula30 ( '<=>' formula30 | 'XOR' formula30 )?
;

// disjunctions
formula30 : formula40 ( 'OR' formula40 )*

```



```

;

// conjunctions
formula40 : formula50 ( 'AND' formula50 )*
;

// logical negations
formula50 : 'NOT' formula50 | formula60 ;

// equality and inequality and relations
formula60 :
    term ( '=' term | '/=' term |
           '<' term | '<=' term | '>' term | '>=' t1=term )
| formula70 ;

// atomic predicates
formula70 :
    name '(' term ( ',' term )* ')'
| 'EXECUTES' '@' statearg
| 'CONTINUES' '@' statearg
| 'BREAKS' '@' statearg
| 'RETURNS' '@' statearg
| 'THROWS' '@' statearg
| 'THROWS' '(' name ')' '@' statearg
| formula100
;

// argument to state predicate
statearg : 'NOW' | 'NEXT' | name ;

// atoms
formula100 :
    name | 'OLD' name | 'VAR' name
| 'TRUE' | 'FALSE'
| 'IF' formula 'THEN' formula
  ( 'ELSIF' formula 'THEN' formula )* 'ELSE' formula 'ENDIF'
| 'WRITESONLY' names | 'READONLY'
| '(' formula ')' ;

// -----
// terms
// -----

// quantifiers bind weakest
term :
(
    'LAMBDA' paramList ':' term
| 'ARRAY' paramList ':' term
| term10

```

```

)
;

// lets
term10 :
  'LET' vdefinition ( ',' vdefinition ) * 'IN' term10
| term20 ;

// sums and differences (differences are left-associative)
term20 : term30 ( '+' term30 | '-' term30 ) * ;

// products and quotients (quotients must be left-associative)
term30 : term40 ( '*' t1=term40 | '/' term40 ) * ;

// power terms (left-associative to be compatible with CVCL)
term40 : term50 ( '^' t1=term50 ) * ;

// unary arithmetic operators
term50 : '+' term50 | '-' term50 | term60 ;

// updates
term60 :
  term70 ( 'WITH'
    ( '.' (NUMBER | IDENT) | '[' term ']' ) + ':= ' term70 ) * ;

// selections
term70 : term80 ( '.' (NUMBER | IDENT) | '[' term ']' ) * ;

// applications
term80 :
  'VALUE' '@' term100
| 'MESSAGE' '@' term100
| term100 ( '(' term ( ',' term ) * ')' ) * ;

// atoms
term100 :
  name
| NUMBER
| STRING
| 'TRUE'
| 'FALSE'
| 'OLD' name
| 'VAR' name
| 'NOW'
| 'NEXT'
| '(' term ( ',' term ) * ')'
| '(# IDENT := term ( ',' IDENT := term ) * '#)'
| 'IF' formula 'THEN' term
  ( 'ELSIF' formula 'THEN' term ) * 'ELSE' term 'ENDIF' ;

```

```

paramList : '(' param ( ',' param )* ')' ;

param : IDENT ( ',' IDENT )* ':' typeExp ;

// value definition
vdefinition :
  IDENT ':' typeExp '=' term
| IDENT '=' term ;

// a name
name : IDENT ( '.' IDENT )* ;

// a sequence of names
names : name ( ',' name )* ;

// -----
// lexical syntax
// -----

IDENT: REALLETTER (LETTER | DIGIT)* ;
NUMBER: DIGIT (DIGIT)* ;

// strings
STRING : '"' ( ~('"' | '\\\ ' | EOL) | ESCAPED )* '"' ;

fragment
REALLETTER: ('a'..'z' | 'A'..'Z' );

fragment
LETTER: ( REALLETTER | '_' );

fragment
DIGIT: ('0'..'9');

WS: ( ' ' | '\t' | EOL | COMMENT) ;

fragment
EOL: ('\n' | '\r' | '\f');

fragment
COMMENT : '//' .* EOL | '/*' .* '*/' ;

fragment
ESCAPED : '\\\ '
  ( '\\\ ' | '"' | '\\ ' | 'n' | 't' | 'b' | 'f' | 'r'
  | ('u' HEX HEX HEX HEX) ) ;

fragment HEX : '0'..'9' | 'a'..'f' | 'A'..'F' ;

```