A User Guide to the RAISE Tools as used in the DTU 02263 course

Anne Haxthausen, Morten Lindegaard and Torben Gjaldbæk
IMM, DTU

February 6, 2008

Contents

1 Introduction

This document contains a user guide to how RSL specifications can be type checked, translated into SML and printed with LaTeX using the RAISE tools developed at UNU/IIST.
First, in section 2 the required installations are described. Then, in sections 3–5 it is described how to use the tools in three steps:

- create a text file containing the considered specification
- type check the specification
- print the specification with LaTeX

Finally, in section 6 it is described how to translate executable specifications into SML and execute them.

For further details, read the full user and installation guide from UNU/IIST available at: http://www.iist.unu.edu/newrh/III/3/1/page.html.

2 Required installations

This user guide assumes that you have installed the RAISE tools developed at UNU/IIST, emacs and one of the newer versions (e.g. 110.54) of New Jersey ML. For how to do that for Linux and Windows, see the DTU installation guides available on http://www.imm.dtu.dk/courses/02263/.

2.1 Note to Windows users

Note that Windows users that have followed Andreas installation guide should execute

```
c:\emacs\bin\runemacs.exe
```

in order to start emacs.

3 How to create RSL specifications in text files

RSL specifications are written in ordinary text files. Use `emacs` to edit a specification of an RSL module with name *X* in a file with name *X.rsl*. You must use ascii equivalents for the non-ascii RSL symbols. A table of ascii equivalents can be found on page 385 of the RSL book.

Here is an example of a specification to be stored in a file with the name `COUNTER.rsl`:

```
scheme COUNTER =
  class
    variable counter : Nat := 0
```

```
    value increase : Unit -> write counter Nat
    axiom increase() is counter := counter + 1 ; counter
  end
```

If a module depends on other modules, their names (without ".rsl") should appear as a comma separated list at the start of the file. This list is the module's context. The tool calculates the transitive closure of contexts, so if A depends on B and B depends on C then you must include `C` as the context for B, and you may for the context of A use either `B` or `B, C`. The files `A.rsl`, `B.rsl`, and `C.rsl` must all be in the same directory. When a module is checked, the context modules are checked first, so you only need to run the tool on top level modules.

4 How to type check RSL specifications using `rsltc`

To syntax and type check a specification X, stored in a file *X.rsl*, open the file in `emacs` and choose `RSL` → `Type check`. [1]

If there are errors, `rsltc` outputs messages in the form

```
X.rsl:m:n: text
```

which indicates that there is an error described by *text* in the file *X.rsl* at line m and column n.

As an example, consider the following erroneous specification:

```
scheme COUNTER =
  class
    variable counter : Nat := 0
    value increase : Unit -> write counter Nat
    axiom increase() is counter := counter + true ; counter
  end
```

Type checking gives the following output:

```
Checking COUNTER ...
COUNTER.rsl:5:46: Argument types Nat and Bool
incompatible with '+' type Int >< Int -> Int or
Real >< Real -> Real or
Int -> Int or
Real -> Real
Finished COUNTER
rsltc completed: 1 error(s) 0 warning(s)
```

After changing the specification to the correct version (section 3), type checking gives the following output:

[1] Alternatively, issue the command `rsltc X` from the command prompt.

```
Checking COUNTER ...
Finished COUNTER
rsltc completed: 0 error(s) 0 warning(s)
```

5 How to print RSL specifications with LaTeX

This is done in two steps:

1. create a LaTeX document as described in section 5.1
2. make a postscript file from the LaTeX document as described in section 5.2

5.1 Creating a LaTeX document

LaTeX files are plain text files having a name of the form *Y.tex*, e.g. `main.tex`. A basic LaTeX file for use with RSL specifications looks like this:

```
\documentclass{article}
\usepackage{rslenv}
\begin{document}
%here you can write text and
%include RSL specifications
%   in one of two ways as described in the following subsections
\end{document}
```

5.1.1 RSL specifications in separate text files

If you have an RSL specification in a separate text file (say, `X.rsl`) that you want to print neatly, you create a LaTeX file (say, `main.tex`) which includes the specification by using the command `\RAISEIN{X}`

A basic LaTeX file including the specification in `X.rsl` looks like this:

```
\documentclass{article}
\usepackage{rslenv}
\begin{document}
\RAISEIN{X}
\end{document}
```

To print the specification, do the following:

1. Open `main.tex` in `emacs`.
2. From `emacs`, issue the command `M-x mkdoc` (that is, press Alt-x, write 'mkdoc' and press Enter). `mkdoc` generates a file `X.tex` from `X.rsl`. If there had been several `RAISEIN` commands, then `mkdoc` would have generated a tex file for each included rsl file.

The file can now be converted to PostScript format as described in section 5.2.

5.1.2 RSL specifications within LaTeX files

RSL specifications can also be written directly within a LaTeX document. This is convenient if you e.g. want to include fractions of a specification within the ordinary text of an article or a report.

RSL formulae within a LaTeX file should be written between `\RSLatex` and `\endRSLatex`, e.g.:

```
\documentclass{article}
\usepackage{rslenv}
\begin{document}
%\RSLatex
%  scheme COUNTER =
%    class
%      variable counter : Nat := 0
%      value increase : Unit -> write counter Nat
%      axiom increase() is counter := counter + 1 ; counter
%    end
%\endRSLatex
\bp
\>\kw{scheme} COUNTER {\EQ}\\
\>\>\kw{class}\\
\>\>\>\kw{variable} counter : \kw{Nat} :{\EQ} 0\\
\>\>\>\kw{value} increase : \kw{Unit} {\RIGHTARROW} \kw{write} counter \kw{Nat}\\
\>\>\>\kw{axiom} increase() {\IS} counter :{\EQ} counter {\PLUS} 1 ; counter\\
\>\>\kw{end}
\ep
\end{document}
```

By pressing key F2 from within emacs, the RSL formulae in ascii-format are converted into LaTeX-format:

```
\documentclass{article}
\usepackage{rslenv}
\begin{document}
%\RSLatex
%  scheme COUNTER =
%    class
%      variable counter : Nat := 0
%      value increase : Unit -> write counter Nat
%      axiom increase() is counter := counter + 1 ; counter
%    end
%\endRSLatex
\bp
\>\kw{scheme} COUNTER {\EQ}\\
\>\>\kw{class}\\
```

```
\>\>\>\kw{variable} counter : \kw{Nat} :{\EQ} 0\\
\>\>\>\kw{value} increase : \kw{Unit} {\RIGHTARROW} \kw{write} counter \kw{Nat}\\
\>\>\>\kw{axiom} increase() {\IS} counter :{\EQ} counter {\PLUS} 1 ; counter\\
\>\>\kw{end}
\ep
\end{document}
```

The file can now be converted to PostScript format as described in section 5.2.

By pressing the key F3, the LaTeX-formatted formulae are converted back to the original ascii-format.

Notice that you may need to change some of the spacing in your ascii formulae to make it appear nicely in the final PostScript file — indentation has to be done by inserting blank spaces.

5.2 Creating a PostScript file from a LaTeX document

5.2.1 Under Windows

To make a PostScript file from a LaTeX file called `main.tex`, do the following:

1. Open the file `main.tex` in `emacs`.
2. Choose `Command` → `LaTeX`. [2]
3. Choose `Command` → `dvips`. [3]

The result is a file called `main.ps`, which may be viewed with *GSview*.

5.2.2 In the E-databar

To make a PostScript file from a LaTeX file called `main.tex`, issue the following commands at the command prompt:

```
latex main
dvips main.dvi -o main.ps
```

The result is a file called `main.ps`, which may be viewed with */usr/bin/gv*.

6 How to translate RSL specifications into SML and run test cases

This section is a short extract of the UNU/IIST RAISE tools user guide.

[2] Alternatively: From the command prompt, run `latex main`

[3] Alternatively: Choose Command → File and press Enter or from the command prompt, run `dvips main.dvi -o main.ps`

6.1 Which specifications can be translated

The tool `rsltc` can translate RSL specifications into SML programs provided that they are in a certain form as explained in the UNU/IIST user guide.

6.2 Translation

To translate a scheme named `X` (in file `X.rsl`) use the following command in a command prompt[4]:
`rsltc -m X.rsl`

The result of the translation are two files: `X.sml` and `X_.sml` containing the SML code that is a translation of `X`. Now you can start an SML session and load `X.sml` by the command `use "X.sml";`.

If `X` contains test cases (see below) you can translate and run the test cases directly from emacs by choosing the item `SML` → `Translate to SML and run` from the RSL menu. Then you can save the output from the run of SML in a file `X.sml.results` by returning to the buffer displaying the RSL file and selecting the item `SML` → `End SML run and save results` in the RSL menu.

6.3 Test cases

There is a new extension to RSL to support interpretation and translation. In addition to type, value, variable, etc. declarations you can now have a test case declaration. The keyword **test_case** is followed by one or more test case definitions. Each test case definition is an optional identifier in square brackets (just like an axiom name) followed by an RSL expression. The expression can be of any type, and it can be imperative. To an interpreter the test cases mean expressions that are to be evaluated.

So if you wrote

test_case
[t1] 1 + 2

you would expect to see the interpreter output

[t1] 3

6.4 Example and exercise

1. Write the following scheme in a file `X.rsl` using emacs:

scheme X =
class
test_case
[t1] 1 + 2,

[4] Alternatively, open the file with emacs and chose `RSL` → `SML` → `Translate to SML`.

[t2] **true** ∨ **false**
end

2. Type check the scheme by choosing the item `Type check` from the RSL menu.

3. Translate the scheme by choosing the item `SML` → `Translate to SML` from the RSL menu and check that two files, `X.sml` and `X_.sml`, have been created.

4. Run the test cases by choosing the item `SML` → `Run SML file` from the RSL menu.

```
Standard ML of New Jersey v110.54 [built: Sat Sep 24 16:19:34 2005]
- [opening /home/ah/.../X.sml]
[autoloading]
[library $smlnj/cm/cm.cm is stable]
[library $smlnj/internal/cm-sig-lib.cm is stable]
[library $/pgraph.cm is stable]
[library $smlnj/internal/srcpath-lib.cm is stable]
[autoloading done]
val it = () : unit
val it = true : bool
[autoloading]
[autoloading done]
val it = () : unit
val it = () : unit
[autoloading]
[autoloading done]
val it = () : unit
val it = true : bool
[autoloading]
[autoloading done]
[opening X_.sml]
structure RT_Int : <sig>
structure RT_Bool : <sig>
structure X : <sig>
open X
val it = () : unit
val it = () : unit
val it = () : unit
val it = () : unit
[t1] 3
val it = () : unit
[t2] true
val it = () : unit
```

```
val it = () : unit
val it = () : unit
val it = () : unit
val it = () : unit
-
```

Lot of the output is not useful for you. It shows the RSL library being loaded. Don't worry that there is not a "stable" version: the .bin file is the compiled version. From the RSL library the structures RT_Int and RT_Bool are loaded, then the structure X generated from the RSL input.

Finally we get the results of the two test cases, followed in each case by the line

```
val it = () : unit
```

This is just SML reporting completion of the function that generated the test case: the current value (it) is the unit value "()" (which is just like its counterpart in RSL). The last identical lines are the results of other functions used to load and run X.sml and X_.sml.

5. Save the output from the run of SML in a file `X.sml.results` by returning to the buffer displaying the RSL file and selecting the item `SML` → `End SML run and save results` in the RSL menu. Check that `X.sml.results` is generated and contains

```
[t1] 3
[t2] true
```