The United Nations University

UNU-IIST

International Institute for Software Technology

RAISE Tool User Guide

Chris George

April 17, 2008

UNU-IIST Report No. 227

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director

The United Nations University

UNU-IIST

International Institute for Software Technology

P.O. Box 3058
Macao

RAISE Tool User Guide

Chris George

Abstract

This is a user guide for the "rsltc" RAISE tool. This provides type checking; pretty-printing; generation of confidence conditions; showing module dependencies; translation to Standard ML, to C++, and to PVS; and translation to RSL from UML class diagrams. The user guide provides full instructions on the use and installation of this tool on Unix, Linux, and Windows platforms.

Chris George is a Senior Research Fellow at UNU/IIST, 1 September 1994 - 31 August 2003. He is one of the main contributors to RAISE, particularly the RAISE method, and that remains his main research interest. Before coming to UNU/IIST he worked for companies in the UK and Denmark.

Contents

1 Introduction

This is a user guide for the "rsltc" RAISE tool. This provides type checking; pretty-printing; generation of confidence conditions; showing module dependencies; translation to Standard ML, to C++, and to PVS; and translation to RSL from UML class diagrams. The user guide provides full instructions on the use and installation of this tool on Unix, Linux, and Windows platforms.

2 Changes to RSL

A small number of changes have been made to RSL since *The RAISE Specification Language* [1] was published.

2.1 With expressions

There is a new kind of class expression:

with *element*-object_expr-list **in** class_expr

which allows the qualifications in names to be omitted. This is particularly useful when you redefine operators. For example, if you have in a scheme S

...
value + : T × T → T
...

and in another you have S as a parameter or make an object from it, then without the with expression you have to write

class
  object O : S
...
  O.(+)(x,y)
...
end

The operator + outside the object O has to be called O.(+) and used prefix.

Now you can write instead

with O **in class**
  object O : S
...

```
  x + y
...
end
```

The same can be done if "O : S" is a scheme parameter.

The meaning of "**with** X **in**" is, essentially, that an applied occurrence of a name N in its scope can mean either N or X.N. It is necessary, of course, that of the two possibilities either only one exists or they are distinguishable.

With expressions may be nested. "**with** Y **in with** X **in**" means that a name N in its scope can mean N or X.N or Y.N or Y.X.N. (The last arises because X is in the scope of the outer **with**, and so can mean X or Y.X.)

It is generally more efficient to use a single **with** rather than nesting them. "**with** Y, X **in**" means that a name N in its scope can mean N or X.N or Y.N. Order within a single **with** is not significant.

2.2 Comments

Because the tool is not a structure editor it can be much more flexible about comments than was the original definition of RSL. Two kinds of comments are allowed wherever white space is possible:

/∗ ... ∗/ comments can extend over as many lines as you like and are nestable.

−− in a line makes the rest of the line a comment

2.3 Prefix + and −

These were omitted from the original RSL, so that you had, for example, to write "0 − 1" instead of just "−1". They are now included.

2.4 == symbol

The infix operator == is included, with the same precedence as =. Semantically this symbol is undefined, and it has no predefined type. It is intended to be used to define abstract equalities, but you can define it in any way you want to.

2.5 Finite maps

There are both finite maps $\xrightarrow[m]{}$ (ASCII symbol `-m->`) and infinite maps $\xrightarrow[\sim m]{}$ (`-~m->`). Finite maps have finite domains and are deterministic when applied to values in their domain. Finite maps were introduced in *The RAISE Development Method* [2].

2.6 Extra meanings for $\in$, $\notin$, and hd

The infix operators $\in$ and $\notin$ can now be applied to lists and maps as well as sets. The meanings correspond to the following definitions for arbitrary types T and U:

value
$\in$: T × T$^\omega$ → **Bool**
t ∈ l ≡ t ∈ **elems** l,

$\notin$: T × T$^\omega$ → **Bool**
t ∉ l ≡ t ∉ **elems** l,

$\in$: T × (T $\overset{\sim}{\underset{m}{\to}}$ U) → **Bool**
t ∈ m ≡ t ∈ **dom** m,

$\notin$: T × (T $\overset{\sim}{\underset{m}{\to}}$ U) → **Bool**
t ∉ m ≡ t ∉ **dom** m

The point of adding these meanings is that the RSL is shorter, and that it is easier to translate the RSL into more efficient code, as there is no need to generate a set from the list or map before applying $\in$ or $\notin$.

The prefix operator **hd** can now be applied to (non-empty) sets and maps. The meaning of **hd** in these two cases is as if **hd** were defined as follows for arbitrary types T and U:

value
hd : T-**infset** $\overset{\sim}{\to}$ T
hd : (T $\overset{\sim}{\underset{m}{\to}}$ U) $\overset{\sim}{\to}$ T
axiom
∀ s : T-**infset** • s ≠ {} ⇒ **hd** s ∈ s
∀ m : T $\overset{\sim}{\underset{m}{\to}}$ U • m ≠ [ ] ⇒ **hd** m ∈ m

The operator **hd** can therefore be used to select an arbitrary member of a non-empty set or of the domain of a non-empty map. This allows many examples of quantified and comprehended expressions to be written in ways that allows them to be translated. The choice of **hd** for this operator may seem a little strange, but using an existing operator avoids adding another keyword.

2.7 Test cases

There is also 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.

Test cases are not an official part of RSL. You can think of them as no more than comments (although the type checker will report errors in them). But to an interpreter they 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

Test cases are interpreted in order, and the result of one can affect the results of others if they are imperative. Suppose, for example, we have have an imperative (integer) stack with the usual functions. Then, if the stack variable is initialised to empty, the following test cases

test_case
[t1] push(1) ; push(3) ; push(4) ; top(),
[t2] pop() ; pop() ; top(),
[t3] pop() ; is_empty()

should produce the following interpreter output

[t1] 4
[t2] 1
[t3] **true**

Test case declarations are only interpreted or translated if they occur at the top level. This means that you can conveniently test modules "bottom-up", since test cases in lower level modules will be ignored when higher ones are tested later.

We need to be precise about what we mean here by the "top level". Suppose we define a scheme `X`. We might add some test cases to it, or we might define a separate scheme to test `X`, defined by

scheme TEST_X = **extend** X **with class test_case** ... **end**

But it might be the case that both `X` and `TEST_X` contain test cases. It is then assumed that the intention is to translate and execute `X` and `TEST_X` separately, so that in `TEST_X` the test cases should *not* include those from `X`. To be more precise, for the purpose of deciding what test cases are included, the "top level" means the class of the module given as input to the translator or interpreter, and:

- test cases from global objects (apart from the top level module, which may be an object) are not included

- test cases from embedded objects (objects defined within classes) are not included
- in "**extend** class1 **with** class2", the test cases from class1 are not included if class1 is a scheme instantiation
- otherwise all test cases are included

2.8 Features of RSL not supported

The tool follows *The RAISE Development Method* [2] in not allowing embedded schemes (schemes defined within classes). Schemes should be defined at the top level and put in their own files (see section 3).

The axiom quantification **forall** has been removed. Axioms should be quantified individually. This also follows *The RAISE Development Method*.

3 Putting RSL into files

A global scheme or object named X, say, must be placed in a file X.rsl. (Case of the file name is significant in Unix and Linux, not in Windows.) So only one global scheme or object is allowed per file.

3.1 Mathematical characters

The display syntax of RSL uses a number of mathematical characters like $\in$ and $\forall$ that are not available in the ASCII character set. In order for RSL to be put into text files (which are what the RAISE tool uses, and are easily portable) there are ASCII equivalents for all the special characters. These are shown in figure 1.

Sym	ASCII	Sym	ASCII	Sym	ASCII
$\times$	`><`	*	`-list`	$^\omega$	`-inflist`
$\rightarrow$	`->`	$\xrightarrow{\sim}$	`-~->`		
$\xrightarrow[m]{}$	`-m->`	$\xrightarrow[m]{\sim}$	`-~m->`	$\leftrightarrow$	`<->`
$\wedge$	`/\`	$\vee$	`\/`	$\Rightarrow$	`=>`
$\forall$	`all`	$\exists$	`exists`	$\bullet$	`:-`
$\Box$	`always`	$\equiv$	`is`	$\neq$	`~=`
$\leq$	`<=`	$\geq$	`>=`	$\uparrow$	`**`
$\in$	`isin`	$\notin$	`~isin`	$\subset$	`<<`
$\subseteq$	`<<=`	$\supset$	`>>`	$\supseteq$	`>>=`
$\cup$	`union`	$\cap$	`inter`	$\dagger$	`!!`
$\langle$	`<.`	$\rangle$	`.>`	$\mapsto$	`+>`
$\parallel$	`||`	$\#$	`++`	$\square$	`|=|`
$\prod$	`|^|`	λ	`-\`	$^\circ$	`#`

Figure 1: ASCII forms of RSL symbols

3.2 Contexts

A module (scheme or object) S that uses other modules A and B needs to tell the type checker that A and B are its context. The file S.rsl for S will start

```
A, B

scheme S = ...
```

The type checker will check A and B (recursively including any modules in their contexts) and then S. The order of A and B does not matter. If B is also in the context of A then it does not matter if B is included in the context of S or not.

The context illustrated above means that the type checker will look for A.rsl and B.rsl in the same directory as S.rsl. Context files may also be in other directories (on the same drive in Windows). References to them may use absolute or relative paths, and Unix-style paths are used (so that RSL files may be passed between Windows and Unix systems).

For example, suppose S.rsl is in /home/user/raise/rsl, and A.rsl is in /home/user/raise/rsl/lower. Then the context reference to A in S.rsl may be

```
lower/A                          or
/home/user/raise/rsl/lower/A     or even
../rsl/lower/A
```

When a module is checked, the context modules are checked first, so you only need run the tool on top level modules.

4 Tool components available

There is a collection of related component tools, nearly all based on the type checker `rsltc`. They are invoked from the command line as shown in table 1.

If the RSL file being used is `X.rsl`, then `<file>` may be given as `X` or `X.rsl`.

These tool components are described in the following sections.

A more convenient interface to the rsltc tool components can be obtained using emacs — see section 15.

See section 13.5 for more information on invoking the UML to RSL translator.

Command	Component
`rsltc <file>`	Type check
`rsltc -pp <file>`	Parsing (no type check) plus pretty printing of current module
`rsltc -c <file>`	Type check plus confidence condition generation on current module
`rsltc -cc <file>`	Type check plus confidence condition generation on all modules
`rsltc -d <file>`	Parsing (no type check) plus display of module dependencies
`rsltc -g <file>`	Generation of VCG file to show module dependencies
`rsltc -m <file>`	Translation to SML
`rsltc -c++ <file>`	Translation to C++
`rsltc -cpp <file>`	Translation to Microsoft's Visual C++
`rsltc -pvs <file>`	Translation to PVS
`UML2RSL <xml-file>`	Translation to RSL from UML

Table 1: rsltc commands

5 Type checker

Type checking is performed on any context files first, followed by the input module mentioned in the command.

The tool outputs the names of modules it is checking. If it finds errors it also outputs (on standard output) messages in the form

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

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

Apart from syntax errors the tool runs to completion. So remember that some errors may be the consequences of earlier ones.

In the case of syntax errors the tool is usually one token ahead of what caused the error, so you may need to go back past the preceding space or new line to find the cause.

6 Pretty printer

The pretty printer for RSL was written by Ms He Hua, as reported in [3].

Provided there are no syntax errors, a pretty-printed version of the input module is output on standard output.

The default output line length is 60 characters. If you wish to vary this then you can use the command

```
rsltc -pl n <file>
```

for a line length `n`, which must be at least 30.

```
rsltc -p S > S.pp
```

will pretty-print S.rsl into a file S.pp. If S.pp is acceptable you will need to copy it into S.rsl to use rsltc on it again.

Warning: the command "`rsltc -p S > S.rsl`" merely empties the file S.rsl.

The emacs interface (section 15) is much more convenient for pretty-printing.

7 Confidence condition generation

The confidence condition generator was written by Tan Xinming.

Confidence conditions are conditions that should generally be true if the module is not to be inconsistent, but that cannot in general be determined as true by a tool. The following conditions are generated (provided the type checker first runs successfully):

1. Arguments of invocations of functions and operators are in subtypes, and, for partial functions and operators, preconditions are satisfied.
2. Values supposed to be in subtypes are in the subtypes. These are generated for
   - values in explicit value definitions;
   - values of explicit function definitions (for parameters in appropriate subtypes and satisfying any given preconditions);
   - initial values of variables;
   - values assigned to variables;
   - values output on channels.
3. Subtypes are not empty.
4. Values satisfying the restrictions exist for implicit value and function definitions.
5. The classes of actual scheme parameters implement the classes of the formal parameters.
6. For an implementation relation or an implementation expression, the implementing class implements the implemented class. This gives a means of expanding such a relation or expression, by asserting the relation in a theory and then generating the confidence conditions for the theory.
7. A definition of a partial function without a precondition (which generates the confidence condition **false**).
8. A definition of a total function with a precondition (which generates the confidence condition **false**).

Confidence conditions are output on standard output. They take the form

```
X.rsl:m:n CC:
-- comment on source of condition
condition
```

Examples of all the first 4 kinds of confidence conditions listed above are generated from the following intentionally peculiar scheme (in which line numbers have been inserted so that readers can relate the following confidence conditions to their source):

```
 1 scheme CC =
 2 class
 3   value
 4     x1 : Int = hd <..>,
 5     x2 : Int = f1(-1),
 6     x3 : Nat = -1,
 7     f1 : Nat -~-> Nat
 8     f1(x) is -x
 9     pre x > 0
10   variable
11     v : Nat := -1
12   channel
13     c : Nat
14   value
15     g : Unit -> write v out c Unit
16     g() is v := -1 ; c!-1
17   type
18     None = {| i : Nat :- i < 0 |}
19   value
20     x4 : Nat :- x4 < 0,
21     f2 : Nat -> Nat
22     f2(n) as r post n + r = 0
23 end
```

This produces the following confidence conditions (which are all provably false):

```
Checking CC ...
Finished CC
CC.rsl:4:19: CC:
-- application arguments and/or precondition
let x = <..> in x ~= <..> end

CC.rsl:5:18: CC:
-- application arguments and/or precondition
-1 >= 0 /\ let x = -1 in x > 0 end

CC.rsl:6:14: CC:
-- value in subtype
-1 >= 0

CC.rsl:8:5: CC:
-- function result in subtype
all x : Nat :- (x > 0 is true) => -x >= 0

CC.rsl:11:13: CC:
```

```
-- initial value in subtype
-1 >= 0

CC.rsl:16:17: CC:
-- assigned value in subtype
-1 >= 0

CC.rsl:16:24: CC:
-- output value in subtype
-1 >= 0

CC.rsl:18:26: CC:
-- subtype not empty
exists i : Nat :- i < 0

CC.rsl:20:8: CC:
-- possible value in subtype
exists x4 : Nat :- x4 < 0

CC.rsl:22:5: CC:
-- possible function result in subtype
all n : Nat :- exists r : Nat :- n + r = 0

rsltc completed: 10 confidence condition(s) generated
rsltc completed: 0 error(s) 0 warning(s)
```

In the case of implementation relations and conditions, the confidence condition is typically the conjunction of a number of conditions, each of which also has a `file:line:column` reference, followed by `IC:` (to indicate an implementation condition), plus some text, as a comment in the condition. Usually these references are to the appropriate place in the implementing class, but in the case of an axiom in the implemented class they are to the axiom, since this will typically have no equivalent in the implementing class.

In general, confidence conditions are not generated for definitions that are syntactically identical in the implementing and implemented classes.

For example, consider the schemes A0 and A1 and the theory A_TH below that asserts that A1 implements A0:

scheme A0 =
  class
    value
      x : **Int**,
      y : **Int** • y > 1,
      z : **Int** = 2
    axiom
      [x_pos]
        x > 0
  end

scheme A1 =
  class
    value
      x : **Int** = 1,
      y : **Int** = 3,
      z : **Int** = 2
  end

A0, A1
theory A_TH :
axiom
  ⊢ A1 ≼ A0
end

Generating confidence conditions for A_TH produces the following output:

```
Loading A0 ...
Loaded A0
Loading A1 ...
Loaded A1
Checking A_TH ...
Finished A_TH
A_TH.rsl:4:12: CC:
-- implementation conditions:
in A1 |-
  -- A1.rsl:5:7: IC: value definition changed
  y > 1 /\
  -- A0.rsl:10:9: IC: [x_pos]
  x > 0

rsltc completed: 1 confidence condition(s) generated
rsltc completed: 0 error(s) 0 warning(s)
```

This confidence condition can be proved.

8 Showing module dependencies

These are shown in a simple ASCII representation. For example, generating them for the scheme QSI from section 11.14 produces the output

```
QSI
        QSPEC
                QSP
        QUICKSORT
                QSP
```

```
I
```

which shows that QSI depends on QSPEC, QUICKSORT and I, and that the first two of these depend in turn on QSP.

9 Drawing a module dependency graph

If run on a file `X.rsl` this generates input for the Visualisation of Computer Graphs (VCG) tool in a file `X.vcg`. For example, applying it to the scheme QSI from section 11.14 generates a file which VCG displays as in figure 2.

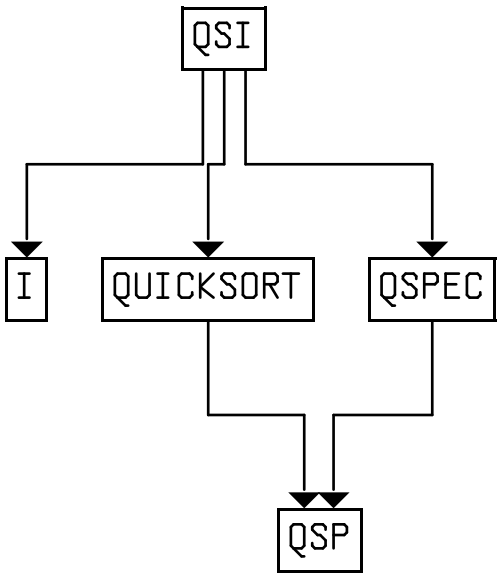

Figure 2: Module dependencies for the scheme QSI

Schemes are drawn as red rectangles, objects as light-blue rectangles, theories as yellow diamonds, and development relations as cyan triangles.

VCG does automatic layout, but there are a number of parameters that may be set interactively or in the file `X.vcg` to change the result. The graph can be exported as a graphic file in various formats for printing or use in documents.

For printing diagrams in black and white involving schemes and objects, it is useful to change "red" to "white" near the start of `X.vcg`, and to export from VCG to postscript format using the "grey" colour mode rather than "b&w". This gives a black and white diagram in which schemes are white rectangles and objects are lightly shaded rectangles.

9.1 Long file names in Windows

VCG only uses the old MS-DOS "8+3" file names. If, for example, your top-level file is called `LONG_MODULE_NAME.rsl` then rsltc will save the vcg input file as `LONG_MODULE_NAME.vcg`. But when the VCG tool is started it will display the error message "Cannot load file LONG_MODULE_NAME.vcg". This is easy to circumvent: use the `Load file` command in the `File` menu and you will see a file name like `LONG_M~1.vcg`, which you can select. This should be the one you need. If you have created several module dependency graphs in the same folder with module names with the same first 5 characters, the results will only differ in the final digit in the VCG menu.

10 SML translator

10.1 Introduction

The SML translator was written by Ke Wei, as reported in [4].

We use the term RSL_{SML} for the subset of RSL that is accepted by the the translator. RSL_{SML} excludes object arrays, channels, axioms, abstract types, union types, implicit value and function definitions. It only includes quantified expressions, comprehended expressions, and implicit let expressions if they conform to the rules given below in sections 10.7.11, 10.7.6, and 10.7.23.

The translator converts all RSL identifiers to unique SML identifiers, which start with the original identifier. This ensures both uniqueness and no clashes with SML reserved words. It is not intended that the SML code be readable or changeable by hand, nor that users need to know SML.

10.1.1 Compilers and platforms

The translator produced code is intended for use with SML/NJ (SML of New Jersey), which is based on SML'97 [5]. The run-time system for SML/NJ is freely available for a variety of platforms from `http://cm.bell-labs.com/cm/cs/what/smlnj/`. The translator has been tested on Solaris, Linux and Windows 9X and NT.

10.1.2 Known errors and problems

The following lists mention known errors and problems.

- The arithmetic types **Int** (including **Nat**) and **Real** are naively translated into `Int` and `Real` without regard for actual limits and precision.
- The translation relies heavily on the syntactic form of the input, which means that often a semantically equivalent piece of RSL text cannot be translated or is translated differently. For example, a record or variant type is accepted, but the equivalent expansion into a sort, some value signatures, and some axioms is not accepted.

10.2 Activating the SML translator

The translator is activated from a shell with the command

```
rsltc -m <file>
```

where `<file>` takes the form `X` or `X.rsl` and contains the RSL scheme or object named `X`. It generates files `X.sml` and `X_.sml`. The latter is loaded by the former.

`X.sml` can be executed by starting the SML run-time system and then giving the command

```
use "<dir>/X.sml";
```

where `<dir>` is the directory where `X.rsl` is located. `dir` should be an absolute path starting with "/" and any intermediate separators must be Unix-style forward slash "/", not DOS-style "\". Note the semicolon ";" at the end of this command. If you forget it you will get a prompt `=` on the next line, and you can type it there.

10.2.1 Output

Executing the file `X.sml` generates some output which is not useful, but which we have not been able to get rid of. For example, the file

scheme X =
  class
    test_case
      [t1]
        1 + 2,
      [t2]
        true ∨ **false**
  end

generates the following output

```
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM; autoload enabled]
- [opening /home/cwg/gentle/ug/ex/X.sml]
val it = () : unit
val it = () : unit
val it = () : unit
val it = true : bool
[starting dependency analysis]
[scanning /home/cwg/sml/rslml/rslml.cm]
[checking /home/cwg/sml/rslml/CM/x86-unix/rslml.cm.stable ... not usable]
[dependency analysis completed]
[Registering for AUTOLOAD...]
val it = () : unit
```

```
[Autoloading...]
[recovering /home/cwg/sml/rslml/CM/x86-unix/rslml.sml.bin... done]
[Autoloading done.]
[opening X_.sml]
structure RT_Int : <sig>
structure RT_Bool : <sig>
structure X : <sig>
open X
val it = () : unit
[t1] 3
val it = () : unit
[t2] true
val it = () : unit
val it = () : unit
val it = () : unit
val it = () : unit
-
```

This shows the RSL library being loaded (see section 10.2.3). 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 three identical lines are the results of other functions used to load and run `X.sml` and `X_.sml`.

Finally there is a prompt "`-`" for the next command. It is possible to return to the file `X.rsl`, make any changes you wish, re-translate, and load `X.sml` again into SML with the `use` command.

10.2.2 Saving the output

In emacs, the output from the run of SML can be saved by returning to the buffer displaying RSL file and selecting the item `Save results from SML run` in the `RSL` menu. This saves the results in a file `X.sml.results`, if the file translated was `X.rsl`. The additional output produced by the SML system is removed, so that the example above for example would give

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

Saving the results will also kill the SML buffer: it will restart next time you run SML.

10.2.3 RSL Library

`X.sml` contains the compilation directive

```
CM.autoload' "<DIR>/rslml.cm";
```

where DIR is the variable RSLML_PATH. This in turn causes the definitions in `rslml.sml` to be loaded. The first time this file is loaded in this way it is compiled; thereafter the compiled version is loaded.

`rslml.sml` contains definitions of SML structures for

- the basic types of RSL and standard functions on them: equality and conversions to strings for output
- parameterised structures for set, list, map and function types
- definitions of the infix and prefix operators on these types
- additional functions for translating quantified and comprehended expressions

We refer to these definitions below as the *RSL library*.

During translation, error messages may be generated with the standard `file:line:column` format showing from where in the RSL specification the message was generated. The cause of the error must be corrected and the translator run again.

During execution, run-time error messages may be produced. There are as follows, where x, y and z indicate values that are part of the generated message, c is a constant and v a variable:

Invalid integer literal x
Division by zero
Modulo zero
Integer exponentiation with negative exponent x
Cannot compute 0 ** 0
Invalid real literal x
Zero raised to non-positive power x
Negative number x raised to non-integer power y
hd applied to empty set
Cannot select from empty set
hd applied to empty list
tl applied to empty list
List applied to index outside index set
Cannot select from empty list hd applied to empty map
Map applied to value outside domain
Nondeterministic enumerated map
Maps do not compose
Cannot select from empty map
List x applied to non-index y
Text x applied to non-index y
Map x applied to non-domain value y

x union y has non-disjoint domains
Cannot compute function equality
Destructor x applied to wrong variant
Reconstructor x applied to wrong variant
Argument of x(y) not in subtype
Precondition of x(y) not satisfied
Result z of x(y) not in subtype
Case incomplete for value x
Value x of c not in subtype
Initial value x of v not in subtype
Value x of v not in subtype
chaos encountered
stop encountered
swap encountered

These messages are generated via exceptions that are caught within each test case, allowing following test cases to be executed.

The SML run-time system may also generate warning messages, for example if let or case expressions are not exhaustive.

10.3 Declarations

A declaration translates to one or more type, constant, function or variable declarations as described below for the various kinds of declarations.

10.3.1 Scheme declarations

Apart from the top level module (which is translated as if it were an object), schemes are only translated when they are instantiated as objects. So a scheme that is instantiated several times will therefore be translated several times. This may appear wasteful, but it saves the need for the restrictions on scheme parameters that would be needed if functors were used for schemes.

10.3.2 Object declarations

An object translates as its translated declarations in a structure of the same name as the object.

An object definition in RSL_{SML} cannot have a formal array parameter.

10.3.3 Type declarations

Type declarations are translated according to their constituent definitions.

Mutually recursive type definitions are not accepted.

Sort definitions Not accepted.

Variant definitions A variant definition translates to a `structure` containing an SML datatype defining the constructors, plus definitions of any destructors and reconstructors. Equality and "toString" functions are also defined.

Wildcard constructors are not accepted.

Short record definitions A short record definition translates to a `structure` containing an SML datatype defining the constructor, plus definitions of any destructors and reconstructors. Equality and "toString" functions are also defined.

Abbreviation definitions An abbreviation definition translates to an SML type definition.

10.3.4 Value declarations

Typings Typings are not accepted.

Explicit value definitions An explicit value definition translates to an SML value definition.

Implicit value definitions Implicit value definitions are not accepted.

Explicit function definitions An explicit function definition translates to an SML function definition.

If the parameter type(s) are subtype(s), run-time checking that arguments satisfy the relevant conditions is included.

If there is a precondition, run-time checking the the precondition is true when the function is invoked is included.

Access descriptors are ignored. The kind of function arrow ($\rightarrow$ or $\xrightarrow{\sim}$) does not matter.

It is not required that the number of parameters matches the number of components in the domain of the function's type expression. For example, the following are all accepted:

type
  U = **Int** × **Bool**
value
  f1: **Int** × **Bool** → **Bool**
  f1(x, y) ≡ ...,
  f2: (**Int** × **Bool**) → **Bool**
  f2(x, y) ≡ ...,

f3: U → **Bool**
f3(x, y) ≡ ...,
f4: U × **Int** → **Bool**
f4(x, y) ≡ ...,
f5: (**Int** × **Bool**) × **Int** → **Bool**
f5(x, y) ≡ ...
f6: (**Int** × **Bool**) × **Int** → **Bool**
f6(x) ≡ ...
f7: **Int** × **Bool** → **Bool**
f7(x) ≡ ...,
f8: (**Int** × **Bool**) → **Bool**
f8(x) ≡ ...,
f9: U → **Bool**
f9(x) ≡ ...,
f10: U × **Int** → **Bool**
f10((x, y), z) ≡ ...,
f11: (**Int** × **Bool**) × **Bool** → **Bool**
f11((x, y), z) ≡ ...

Implicit function definitions Implicit function definitions are not accepted.

10.3.5 Variable declarations

A variable declaration translates to SML variable definitions.

If the variable type is a subtype, run-time checking that the initial value is in the subtype is included.

Multiple variable definitions, and uninitialised single variable definitions are not accepted.

10.3.6 Channel declarations

Not accepted.

10.3.7 Axiom declarations

Not accepted.

10.4 Class expressions

A class expression translates to the definitions which the translation of the contents of the class expression results in.

10.4.1 Basic class expressions

A basic class expression translates as its declarations.

10.4.2 Extending class expression

An extending class expression translates as the two class expressions.

10.4.3 Hiding class expressions

Hiding is ignored: hidden names are visible. Since internal names are used this causes no problems.

10.4.4 Renaming class expression

Renaming is ignored. Since internal names are used this causes no problems.

10.4.5 With expression

With expressions are translated by opening the SML structures for objects.

10.4.6 Scheme instantiations

A scheme instantiation translates as the unfolded scheme with substituted parameters.

10.5 Object expressions

An object expression which is a name is accepted as an actual scheme parameter or as a qualification.

A fitting object expression is accepted as an actual scheme parameter.

Neither element object expressions nor array object expressions are accepted.

10.6 Type expressions

10.6.1 Type literals

The RSL type literals are accepted.

10.6.2 Names

A type name that is not an abbreviation translates as a name. A type name that is an abbreviation translates as the abbreviation.

10.6.3 Product type expressions

A product type expression translates to a structure that defines equality and a "toString" function for output.

10.6.4 Set type expressions

A set type expression translates to an instantiation of the set functor from the RSL library.

10.6.5 List type expressions

A list type expression translates to an instantiation of the list functor from the RSL library.

10.6.6 Map type expressions

A map type expression translates to an instantiation of the map functor from the RSL library.

10.6.7 Function type expressions

A function type expression translates to an instantiation of the function functor from the RSL library.

10.6.8 Subtype expressions

A subtype expression translates as its maximal type.

10.6.9 Bracketed type expressions

A bracketed type expression translates as its constituent type expression.

10.7 Value expressions

10.7.1 Value literals

The RSL value literals are accepted.

10.7.2 Names

A value name translates as a name.

10.7.3 Pre names

Not accepted.

10.7.4 Basic expressions

The only basic expression in RSL_{SML} is **skip**.

10.7.5 Product expressions

A product expression translates to an SML product expression.

10.7.6 Set expressions

Enumerated and ranged set expressions are accepted.

A comprehended set expression can only be translated if it takes one of the forms

{ e | b : T • b ∈ slm }
or
{ e | b : T • b ∈ slm ∧ p }

where b is a binding, T is a type, e is a translatable expression, slm is a translatable expression of set, list, or map type, and p is a translatable expression of type **Bool**.

10.7.7 List expressions

Enumerated and ranged list expressions are accepted.

A comprehended list expression can only be translated if it takes one of the forms

⟨ e | b **in** l ⟩
or
⟨ e | b **in** l • p ⟩

where b is a binding, e is a translatable expression, l is a translatable expression of list type, and p is a translatable expression of type **Bool**.

10.7.8 Map expressions

Enumerated map expressions are accepted.

A comprehended map expression can only be translated if it takes one of the forms

[e1 ↦ e2 | b :T • b ∈ slm]
or
[e1 ↦ e2 | b :T • b ∈ slm ∧ p]

where b is a binding, e1 and e2 are translatable expressions, T is a type, slm is a translatable expression of set, list, or map type, and p is a translatable expression of type **Bool**.

10.7.9 Function expressions

Function expressions are accepted.

10.7.10 Application expressions

An application expression may be translated to a function call, a list application or a map application.

10.7.11 Quantified expressions

Quantified expressions can only be translated if they take one of the following forms:

∀ b : T • b ∈ slm
∀ b : T • b ∈ slm ⇒ p
∀ b : T • b ∈ slm ∧ p ⇒ q
∃ b : T • b ∈ slm
∃ b : T • b ∈ slm ∧ p
∃! b : T • b ∈ slm
∃! b : T • b ∈ slm ∧ p

where b is a binding, T is a type, slm is a translatable expression of set, list or map type, and p and q are translatable expressions of type **Bool**.

10.7.12 Equivalence expressions

Not accepted.

10.7.13 Post expressions

Not accepted.

10.7.14 Disambiguation expressions

A disambiguation expression translates as its constituent value expression.

10.7.15 Bracketed expressions

A bracketed expression translates to a bracketed expression.

10.7.16 Infix expressions

An infix expression translates to the corresponding SML expression.

10.7.17 Prefix expressions

A prefix expression translates to the corresponding SML expression.

A universal prefix expression (□) is not accepted.

10.7.18 Initialise expressions

Not accepted.

10.7.19 Assignment expressions

An assignment expression translates to an assignment expression.

10.7.20 Input expressions

Not accepted.

10.7.21 Output expressions

Not accepted.

10.7.22 Local expressions

Local expressions are accepted.

10.7.23 Let expressions

Explicit let expressions are accepted, subject to the restrictions on patterns listed in section 10.7.25.

An implicit let expression can only be translated if it has one of the forms

let b : T • b ∈ slm **in** ... **end**
or
let b : T • b ∈ slm ∧ p **in** ... **end**

where b is a binding, T a type, slm a translatable expression of set, list, or map type, and p a translatable expression of type **Bool**.

10.7.24 If expressions

An if expression translates to an if expression.

10.7.25 Case expressions

Case expressions are accepted subject to some restrictions on possible patterns when the case is over a variant or record type and involves a function constructor:

- The function must be the constructor of a record or variant.
- **Int**, **Real** and **Text** literal patterns are not accepted.
- Equality patterns are not accepted.

10.7.26 While expressions

While expressions are accepted.

10.7.27 Until expressions

Until expressions are accepted.

10.7.28 For expressions

For expressions are accepted.

11 C++ translator

11.1 Introduction

The C++ translator was developed by Univan Ahn [6].

The C++ translator is based heavily on the translator developed for the original RAISE tools by Jesper Gørtz, Jan Reher, Henrik Snog, and Eld Zierau of Cap Programator. We are grateful for their permission to reuse their work.

We use the term $\text{RSL}_{\text{C++}}$ for the subset of RSL that is accepted by the the translator. $\text{RSL}_{\text{C++}}$ excludes object arrays, channels, axioms, abstract types, union types, implicit value and function definitions. It only includes quantified expressions, comprehended expressions, implicit let expressions, and local expressions if they conform to the rules given below in sections 11.8.12, 11.8.7, 11.8.24, and 11.8.23. It includes overloading of names only if they conform to the rules of overloading of C++: overloaded identifiers must be the names of functions with distinguishable parameter types.

The translator has to generate some names, and these always include somewhere the string "rsl" (where some letters may be upper case). So $\text{RSL}_{\text{C++}}$ does not include identifiers containing this string. Neither does it contain identifiers that are C++ keywords, nor names involving double underscores.

The term *universal types* is used for some generated C++ types. See section 11.12 on universal types.

11.1.1 Compilers and platforms

The translator produced code has been tested with the Free Software Foundation's GNU C++ compiler `g++` version 2.95.2. It is intended to conform to the ANSI C++ standard and so should work with other compilers.

The translator has been run under Solaris, Linux and Windows 9X and NT, and the C++ output compiled and run under these operating systems.

The translator has also been used with Microsoft's Visual C++ compiler, version 6.0. This causes some difficulty, especially with the use of overloaded template functions. It is thought that most problems have been resolved.

11.1.2 Known errors and problems

The following lists mention known errors and problems and a few desirable extensions. Other problems and wishes will undoubtedly emerge from the use of the translator. It is worth noting that many of these problems also arise on manual translation from RSL to C++, and that the problems, with the exception of those concerning **Int** and **Real**, are easily avoided.

- C++ compilers seem not to accept nested namespaces with the same names. So an object `A`, say, should not contain an object also called `A`. This is easily avoided with some manual renaming of objects.
- The arithmetic types **Int** (including **Nat**) and **Real** are naively translated into `int` and `double` without regard for actual limits and precision. Float-integral conversions and the results of integer divide and modulo applied to negative operands are also implementation dependent. And so is the use of arithmetic exceptions on overflow and division by zero.
- In RSL the initial value of a variable declared without initialisation is underspecified within its type. The corresponding variable in C++ will be uninitialised.
- Map enumeration is treated by the translator by means of override instead of union, that way disregarding any possibly resulting non-determinism.
- The translation relies heavily on the syntactic form of the input, which means that often a semantically equivalent piece of RSL text cannot be translated or is translated differently. For example, a record or variant type is accepted, but the equivalent expansion into a sort, some value signatures, and some axioms is not accepted.
- Separate compilation is not currently supported. Running the translator on a number of RSL modules generates one header `.h` file and one body `.cc` or `.cpp` file.
- The use of templates with Microsoft's Visual C++ causes problems. There is an option to translate for this compiler, which greatly reduces the use of templates.

11.2 Activating the C++ translator

The translator is activated from a shell with the command

```
rsltc -c++ <file>
```

where `<file>` takes the form `X` or `X.rsl` and contains the RSL scheme or object named `X`. It generates files `X.h` and `X.cc`.

The translator may also be invoked with the command

```
rsltc -cpp <file>
```

which produces output more likely to satisfy Microsoft's Visual C++ compiler. In this case the second output file is `X.cpp`.

`X.h` contains the include directive

```
#include "RSL_typs.h"
```

and `X.cc` or `X.cpp` contains the include directive

```
#include "RSL_typs.cc"
```

`RSL_typs.h` and `RSL_typs.cc`, together with the other files they include or need, namely

```
RSL_comp.h
RSL_list.cc
RSL_list.h
RSL_map.cc
RSL_map.h
RSL_prod.h
RSL_set.cc
RSL_set.h
RSL_text.h
cpp_RSL.cc
cpp_RSL.h
cpp_io.cc
cpp_io.h
cpp_list.cc
cpp_list.h
cpp_map.h
cpp_set.cc
cpp_set.h
```

must be available. These files are all supplied with rsltc. We refer to them below as the *RSL library files*.

A simple script called "rslcomp" is supplied for compiling. In its Unix/Linux version it takes the form

```
#!/bin/sh
CPP_DIR=...
g++ -o $1 -DRSL_io -DRSL_pre -I$CPP_DIR $1.cc
```

where `...` is the directory where the RSL library files are placed. The command

```
rslcomp X
```

will compile and link the files `X.h` and `X.cc` to make an executable `X`. This should only be used when `X.rsl` includes a **test_case** declaration: otherwise there will be no `main` function. Otherwise `X.cc` may be compiled with hand-written C++ files that `#include "X.cc"`.

You should include `-DRSL_boolalpha` if your compiler accepts the "boolalpha" conversion (e.g. Visual C++ version 6, and perhaps earlier, GNU g++ version 3). See section 11.13.1.

The compilation flag `RSL_io` enables input and output using the streams library of C++: see section 11.13. It is needed when `X.rsl` includes a main function, or there will be compilation errors.

The compilation flag `RSL_pre` is optional. Its inclusion results in run-time checks being included:

- Arguments to functions satisfy any subtype conditions and preconditions. The error message is one of the following (where f is a function identifier and args are the actual parameters):

  Arguments of f(args) not in subtypes
  Precondition of f(args) not satisfied

- Results of functions satisfy any subtype conditions. The error message is of the form (where f is a function identifier, args are the actual parameters, and r the result value):

  Result r of f(args) not in subtype

  Recursion of a function through a subtype condition or precondition is also checked during translation and generates a warning about circularity: execution with `RSL_pre` defined would cause an infinite loop.

- Defining values of constants satisfy any subtype conditions. The error message is (where x is a value and c is a constant identifier):

  Value x of constant c not in subtype

- Initial values of variables satisfy any subtype conditions. The error message is (where x is a value and v is a variable identifier):

  Initial value x of variable v not in subtype

- Assigned values of variables satisfy any subtype conditions. The error message is (where x is a value and v is a variable identifier):

  Assigned value x of variable v not in subtype

- Destructors and reconstructors of variant components are applied to the correct component. The error message is one of the following (where f is a function identifier):

  Destructor f applied to wrong variant
  Reconstructor f applied to wrong variant

The messages are prefixed by the standard `file:line:column` showing where in an RSL file the error occurred.

`RSL_pre` is intended for use in testing. It should be used with `RSL_io`, when any error messages (C++ `strings`) will be generated on standard output, and (except for the destructor/reconstructor errors) the program continues running. If `RSL_io` is not present, an error will simply cause the program to abort. This behaviour is defined by the function `RSL_warn` in the library file `cpp_RSL.h`, so users can easily change this behaviour if they wish by editing this file. In the case of the destructor/reconstructor errors,

the message is also generated on standard output if `RSL_io` is defined, but the program then always aborts. This behaviour is defined by `RSL_fail`, also in the library file `cpp_RSL.h`.

Two kinds of messages may be produced during translation. Both start with the standard `file:line:column` showing from where in the RSL specification the message was generated.

1. Error messages indicate something that could not be translated. The cause of the error must be corrected and the translator run again.
2. Warning messages have a text starting `Warning:`. They indicate RSL that could not be translated completely (and the C++ output will not in general compile) but where it may be possible (and perhaps intended) to correct or complete the C++ code by hand. Examples are value definitions without bodies and implicit function or value definitions.

During execution, run-time error messages may be produced (apart from those listed above when `RSL_pre` is defined). There are as follows, where x and y indicate values that are part of the generated message.

Head of empty set
Choose from empty set
Head of empty list
Tail of empty list
Choose from empty list
List x applied to non-index y
Head of empty map
Choose from empty map
Map x applied to y: outside domain
Head of empty text
Tail of empty text
Choose from empty text
Case exhausted
Let pattern does not match

These messages are produced by the function `RSL_fail`, which outputs the message (a C++ `string`) on standard output (provided `RSL_io` is defined) and aborts. `RSL_fail` is defined in the library file `cpp_RSL.h`, so users can easily change this behaviour if they wish by editing this file.

11.2.1 Example

The following simple scheme SUM:

scheme SUM =
  class
    variable s : **Nat** := 0

    value
      inc : **Nat** $\xrightarrow{\sim}$ **write** s **Unit**
      inc(x) ≡ s := s + x
      pre x > 0

```
    value
      val : Unit → read s Nat
      val() ≡ s

    test_case
      [ t1 ]
        inc(2) ; inc(3) ; val()
end
```

translates to a header file `SUM.h`:

```
// Translation produced by rsltc <date>

#ifndef SUM_RSL
#define SUM_RSL

#include "RSL_typs.h"
extern void inc(const int x_2C7);
extern int val();
extern int s;
#ifdef RSL_io
extern int RSL_test_case_t1();
#endif //RSL_io

#endif //SUM_RSL
```

and a body file `SUM.cc`:

```
// Translation produced by rsltc <date>

#include "SUM.h"
#include "RSL_typs.cc"
void inc(const int x_2C7){
#ifdef RSL_pre

if (!(RSL_is_Nat(x_2C7))) RSL_warn("SUM.rsl:6:7: Arguments of inc not in subtypes");
if (!(x_2C7 > 0)) RSL_warn("SUM.rsl:6:7: Precondition of inc not satisfied");
#endif //RSL_pre

s = s + x_2C7;
}

int val(){
return s;
}

int s = 0;
```

```
#ifdef RSL_io
int RSL_test_case_t1(){
inc(2);
inc(3);
return val();
}

int main(){
cout << "[t1] " << RSL_int_to_string(RSL_test_case_t1()) << "\n";
}

#endif //RSL_io
```

There are several points to note:

- The identifiers exported from the RSL scheme, namely `s`, `inc` and `val` are exported from the C++ code with the same names. This eases the integration of code from the translator with hand-written code.
- Names local to other definitions, like the parameter `x`, may be renamed in the C++ code. Unique names for local names are used in the C++ code to avoid problems with differences between RSL and C++ in scoping and overloading. They always start with the original name.
- If `RSL_pre` is defined, code is included to check the subtype condition and precondition for `inc`. No code is generated to check the initialisation of `s` (since 0 is in `Nat`) or to check the assignment to `s` in `inc` (since the sum of two `Nat`s must be a `Nat`).
- The `test_case` declaration results in the definition of a main function so that the C++ code may be immediately compiled and run for testing. It would be more usual to write the `SUM` module without the test case, and write a second module `SUM_TEST`, say:

  ```
  SUM_TEST = extend SUM with class test_case ... end
  ```

  `SUM_TEST` can then be translated and executed to test `SUM`.
- `SUM` is defined as a scheme. If it had been defined as an object then its definitions would have been placed in a namespace called `SUM`, and its exported names in C++ would then be `SUM::s`, `SUM::inc` and `SUM::val`.

11.3 Declarations

A declaration translates to one or more type, constant, function or variable declarations as described for the various kinds of declarations.

If the declaration occurs in a class expression, the declarations are placed at the outermost level.

Declarations from local expressions are included in-line when they are only variables and constant values. Otherwise they are placed in a separate namespace outside the definition where the local expression occurred: see section 11.8.23.

Type declarations are always placed in the header file. To accommodate C++'s requirement of declaration before use the produced declarations are sorted according to kind in the following order:
type definitions (including those from embedded objects)
embedded objects (in namespaces)
constants and functions
variables
test cases

11.3.1 Scheme declarations

Apart from the top level module, schemes are only translated when they are instantiated as objects. So a scheme that is instantiated several times will therefore be translated several times. This may appear wasteful, but it only affects the size of the C++ source, not the final object code, and saves the need for the restrictions on scheme parameters that would be needed if templates were used for schemes.

11.3.2 Object declarations

An object translates as its translated declarations placed within a namespace of the same name as the object.

An object definition in $\text{RSL}_{\text{C++}}$ cannot have a formal array parameter.

11.3.3 Type declarations

A type declaration translates to one or more type definitions for each type definition in its type definition list. Several type definitions generate C++ class definitions with member functions for test of equality and so on and accompanied by the specified constructor, destructor and reconstructor functions. These type definitions include short record definitions, variant definitions and abbreviation definitions that name product types. Recursive data structures may be specified by means of record variants, which translate to dynamic structures. All classes are declared public as structures, which are prototype declared in front of the definitions proper. All produced type definitions are placed in the header part.

Sort definitions A sort definition translates to an almost empty C++ struct in order to support hand-translation.

type Sort

gives a warning message and translates to (input/output operators omitted):

```
struct Sort/*INCOMPLETE: abstract type*/{
bool operator==(const Sort& RSL_v) const{
return true;
```

```
}

bool operator!=(const Sort& RSL_v) const{
return false;
}

};
```

Variant definitions A variant definition translates to a `struct` containing a tag field identifying the variant-choice and a pointer to the record variant. Allocation and deallocation of record variant structures are handled by the various constructor and member functions by means of reference counts. The following struct is the base class of all record variants

```
struct RSL_class {
  int refcnt;
  RSL_class () {refcnt = 1;}
};
```

The variant

type
V ==
Vconst |
Vint(**Int**) |
Vrec(d1Vrec : **Int** ↔ r1Vrec, d2Vrec : V ↔ r2Vrec)

translates to (input/output operators included)

```
// From the .h file ...
// type and constant declarations and inline functions
static const int RSL_Vconst_tag = 1;
static const int RSL_Vint_tag = 2;
static const int RSL_Vrec_tag = 3;
struct RSL_Vint_type;
struct RSL_Vrec_type;
struct V{
int RSL_tag;
RSL_class* RSL_ptr;
 V(const int RSL_p0 = 0){
RSL_tag = RSL_p0;
RSL_ptr = 0;
}

 V(const RSL_Vint_type* RSL_v){
RSL_tag = RSL_Vint_tag;
RSL_ptr = (RSL_class*)RSL_v;
```

```
}

 V(const RSL_Vrec_type* RSL_v){
RSL_tag = RSL_Vrec_tag;
RSL_ptr = (RSL_class*)RSL_v;
}

void RSL_destructor();
 ~V(){
RSL_destructor();
}

 V(const V& RSL_v);
const V& operator=(const V& RSL_v);
bool operator==(const V& RSL_v)  const;
bool operator!=(const V& RSL_v) const{
return !operator==(RSL_v);
}

};
extern string RSL_to_string(const V& RSL_v);
#ifdef RSL_io
extern ostream& operator<<(ostream& RSL_os, const V& RSL_v);

extern istream& operator>>(istream& RSL_is, V& RSL_v);

#endif //RSL_io
static const V Vconst(RSL_Vconst_tag);
struct RSL_Vint_type : RSL_class, RSLProduct1<int, RSL_constructor_fun>{
 RSL_Vint_type(){}

 RSL_Vint_type(const int RSL_p1) :
  RSL_class(), RSLProduct1<int, RSL_constructor_fun>::RSLProduct1(RSL_p1){}

};
extern V Vint(const int RSL_p1);
struct RSL_Vrec_type : RSL_class, RSLProduct2<int, V, RSL_constructor_fun>{
 RSL_Vrec_type(){}

 RSL_Vrec_type(const int RSL_p1, const V& RSL_p2) :
  RSL_class(),
  RSLProduct2<int, V, RSL_constructor_fun>::RSLProduct2(RSL_p1, RSL_p2){}

};
extern V Vrec(const int RSL_p1, const V& RSL_p2);
inline int d1Vrec(const V& RSL_v){
#ifdef RSL_pre

if (RSL_v.RSL_tag != RSL_Vrec_tag)
{
RSL_fail("V.rsl:6:10: Destructor d1Vrec applied to wrong variant");
```

```
}
#endif //RSL_pre

return ((RSL_Vrec_type*)RSL_v.RSL_ptr)->RSL_f1;
}

inline V d2Vrec(const V& RSL_v){
#ifdef RSL_pre

if (RSL_v.RSL_tag != RSL_Vrec_tag)
{
RSL_fail("V.rsl:6:35: Destructor d2Vrec applied to wrong variant");
}
#endif //RSL_pre

return ((RSL_Vrec_type*)RSL_v.RSL_ptr)->RSL_f2;
}

extern V r1Vrec(const int RSL_p0, const V& RSL_v);
extern V r2Vrec(const V& RSL_p0, const V& RSL_v);
```

The templates `RSLProductn` ($1 \leq n \leq 10$) are defined in `RSL_Prod.h`. Each takes as arguments the n types making the product plus a function generating a string for a constructor. This function is `RSL_constructor_fun` (which generates the empty string) for products and variant components, and a function generating `"mk_T"` for a record type `T`.

```
// From the .cc file ...
// RSL constructor, destructor and reconstructor functions
void V::RSL_destructor(){
switch (RSL_tag) {
case RSL_Vint_tag:
if (--(RSL_ptr->refcnt) == 0)
{
delete (RSL_Vint_type*)RSL_ptr;
}
break;
case RSL_Vrec_tag:
if (--(RSL_ptr->refcnt) == 0)
{
delete (RSL_Vrec_type*)RSL_ptr;
}
break;
}
}


 V::V(const V& RSL_v){
switch (RSL_v.RSL_tag) {
case RSL_Vint_tag:
case RSL_Vrec_tag:
```

```
RSL_v.RSL_ptr->refcnt++;
}
RSL_tag = RSL_v.RSL_tag;
RSL_ptr = RSL_v.RSL_ptr;
}


const V& V::operator=(const V& RSL_v){
if (this == &RSL_v)
{
return RSL_v;
}
switch (RSL_v.RSL_tag) {
case RSL_Vint_tag:
case RSL_Vrec_tag:
RSL_v.RSL_ptr->refcnt++;
}
RSL_destructor();
RSL_tag = RSL_v.RSL_tag;
RSL_ptr = RSL_v.RSL_ptr;
return *this;
}


bool V::operator==(const V& RSL_v) const{
if (RSL_tag != RSL_v.RSL_tag)
{
return false;
}
switch (RSL_tag) {
case RSL_Vint_tag:
return *(RSL_Vint_type*)RSL_ptr == *(RSL_Vint_type*)RSL_v.RSL_ptr;
case RSL_Vrec_tag:
return *(RSL_Vrec_type*)RSL_ptr == *(RSL_Vrec_type*)RSL_v.RSL_ptr;
default:
return true;
}
}


string RSL_to_string(const V& RSL_v){
string RSL_Temp_0;

switch (RSL_v.RSL_tag) {
case RSL_Vconst_tag:
RSL_Temp_0 = "Vconst";
break;
case RSL_Vint_tag:
RSL_Temp_0 = "Vint" + RSL_to_string(*(RSL_Vint_type*)RSL_v.RSL_ptr);
break;
case RSL_Vrec_tag:
```

```
RSL_Temp_0 = "Vrec" + RSL_to_string(*(RSL_Vrec_type*)RSL_v.RSL_ptr);
break;
default:
RSL_Temp_0 = "Unknown variant value";
break;
}
return RSL_Temp_0;
}

#ifdef RSL_io
ostream& operator<<(ostream& RSL_os, const V& RSL_v){
switch (RSL_v.RSL_tag) {
case RSL_Vconst_tag:
RSL_os << "Vconst";
break;
case RSL_Vint_tag:
RSL_os << "Vint" << *(RSL_Vint_type*)RSL_v.RSL_ptr;
break;
case RSL_Vrec_tag:
RSL_os << "Vrec" << *(RSL_Vrec_type*)RSL_v.RSL_ptr;
break;
default:
RSL_os << "Unknown variant value";
break;
}
return RSL_os;
}


const RSL_input_token_type RSL_Vconst_token = Token_StartIndex + 1;

const RSL_input_token_type RSL_Vint_token = Token_StartIndex + 2;

const RSL_input_token_type RSL_Vrec_token = Token_StartIndex + 3;

static void RSL_input_token_V(istream& RSL_is, RSL_input_token_type& RSL_token){
char RSL_buf[128];

RSL_fetch_token(RSL_is, RSL_token, RSL_buf);
if (RSL_token == RSL_constructor_token)
{
if (RSL_streq(RSL_buf, "Vconst"))
{
RSL_token = RSL_Vconst_token;
return;
}
if (RSL_streq(RSL_buf, "Vint"))
{
RSL_token = RSL_Vint_token;
return;
}
```

```
if (RSL_streq(RSL_buf, "Vrec"))
{
RSL_token = RSL_Vrec_token;
return;
}
RSL_token = RSL_error_token;
}
}


istream& operator>>(istream& RSL_is, V& RSL_v){
RSL_input_token_type RSL_token;

V RSL_temp;

RSL_class* RSL_ptr = 0;

RSL_input_token_V(RSL_is, RSL_token);
switch (RSL_token) {
case RSL_Vconst_token:
RSL_temp = V(RSL_Vconst_tag);
break;
case RSL_Vint_token:
RSL_ptr = new RSL_Vint_type;
RSL_is >> *(RSL_Vint_type*)RSL_ptr;
if (RSL_is)
{
RSL_temp = V((RSL_Vint_type*)RSL_ptr);
}
break;
case RSL_Vrec_token:
RSL_ptr = new RSL_Vrec_type;
RSL_is >> *(RSL_Vrec_type*)RSL_ptr;
if (RSL_is)
{
RSL_temp = V((RSL_Vrec_type*)RSL_ptr);
}
break;
default:
RSL_is.clear(ios::badbit);
break;
}
if (RSL_is)
{
RSL_v = RSL_temp;
}
else
{
RSL_is.clear(ios::badbit);
}
return RSL_is;
```

```
}


#endif //RSL_io
V Vint(const int RSL_p1){
RSL_Vint_type* RSL_v = new RSL_Vint_type(RSL_p1);

if (!RSL_v)
{
abort();
}
return V(RSL_v);
}

V Vrec(const int RSL_p1, const V& RSL_p2){
RSL_Vrec_type* RSL_v = new RSL_Vrec_type(RSL_p1, RSL_p2);

if (!RSL_v)
{
abort();
}
return V(RSL_v);
}

V r1Vrec(const int RSL_p0, const V& RSL_v){
#ifdef RSL_pre

if (RSL_v.RSL_tag != RSL_Vrec_tag)
{
RSL_fail("V.rsl:6:27: Reconstructor r1Vrec applied to wrong variant");
}
#endif //RSL_pre

return Vrec(RSL_p0, ((RSL_Vrec_type*)RSL_v.RSL_ptr)->RSL_f2);
}

V r2Vrec(const V& RSL_p0, const V& RSL_v){
#ifdef RSL_pre

if (RSL_v.RSL_tag != RSL_Vrec_tag)
{
RSL_fail("V.rsl:6:50: Reconstructor r2Vrec applied to wrong variant");
}
#endif //RSL_pre

return Vrec(((RSL_Vrec_type*)RSL_v.RSL_ptr)->RSL_f1, RSL_p0);
}
```

Constructors, destructors and reconstructors translate as the identifier or operator does. Wildcard constructors are not accepted.

When the translated code is compiled with the `RSL_io` flag, a handwritten C++ compilation unit can perform input/output of variant values. For example:

```
void main(){
V aV, anotherV;
aV = Vint(42);
cout << "First value: " << aV << "\n";
cout << "Give a value of type V:\n";
cin >> anotherV;
cout << "Second value: " << anotherV << "\n";
}
```

The following is an example of an execution of this program (user lines are marked with `#`):

```
First value: Vint(42)
Give a value of type V:
Vrec(1957, Vint(1969)) #
Second value: Vrec(1957,Vint(1969))
```

Union definitions Not accepted.

Short record definitions A short record definition translates to a C++ class definition including member functions and function definitions that implement the constructor, destructor and reconstructor functions. Note that a short record translates differently from a variant definition — the short record translates without the use of pointers.

type
Complex :: re: **Real** ↔ r im: **Real** ↔ i

translates to

```
// from the .h file ...
char* RSL_mk_Complex_fun(){
return "mk_Complex";
}

typedef RSLProduct2<double, double, RSL_mk_Complex_fun> Complex;
inline Complex mk_Complex(const double RSL_p1, const double RSL_p2){
return Complex(RSL_p1, RSL_p2);
}

inline double re(const Complex& RSL_v){
return RSL_v.RSL_f1;
}
```

```
inline double im(const Complex& RSL_v){
return RSL_v.RSL_f2;
}

extern Complex r(const double RSL_p0, const Complex& RSL_v);
extern Complex i(const double RSL_p0, const Complex& RSL_v);


// from the .cc file ...
Complex r(const double RSL_p0, const Complex& RSL_v){
return Complex(RSL_p0, RSL_v.RSL_f2);
}

Complex i(const double RSL_p0, const Complex& RSL_v){
return Complex(RSL_v.RSL_f1, RSL_p0);
}
```

Abbreviation definitions An abbreviation definition translates to a type definition of a universal type whose name is derived from the structure of the type (see section 11.12 on universal types), plus a definition of the identifier as the universal type. For each translation there is at most one definition of each universal type.

type
B = C × C,
C = {| n : **Int** • n ∈ {0..7} |},
D = **Nat** × **Int**,
E = D → D,
F = C × D → D

translates to

```
typedef int C;

typedef RSLProduct2<int, int, RSL_constructor_fun> RSL_IxI;

typedef RSL_IxI D;

typedef RSL_IxI (* RSL_IxIfIxI)(const RSL_IxI );

typedef RSL_IxIfIxI E;

typedef RSL_IxI (* RSL_Ix6IxI9fIxI)(const int , const RSL_IxI );

typedef RSL_Ix6IxI9fIxI F;

typedef RSL_IxI B;
```

11.3.4 Value declarations

Typings Typings are accepted with a warning that they are incomplete

Explicit value definitions An explicit value definition translates to a constant declared in the header file and defined in the body file. For example

value
  low: **Int** = 0,
  high: **Int** = low + 100,
  max : **Int** = Max(low, high),
  p : **Int** × **Bool** = (7, **true**)

translates to

```
// Header file:
extern const int low;
extern const int high;
extern const int max;
extern const RSL_IxB p;

// Body file:
const int low = 0;
const int high = low + 100;
const int max = Max(low, high);
const RSL_IxB p = RSL_IxB(7, true);
```

Additional code, included if `RSL_pre` is defined, is generated if any constant types are subtypes, to check that the values of the constants are in the subtypes.

An explicit value definition which defines a function translates by means of a C++ pointer to function type. E.g.

value
  f : **Int** × **Int** → **Int** = Max

translates to

```
// Header file:
typedef int (* RSL_IxIfI)(const int , const int );
extern const RSL_IxIfI f;

// Body file:
const RSL_IxIfI f = Max;
```

Implicit value definitions Implicit value definitions are accepted with a warning that they are incomplete.

Explicit function definitions An explicit function definition translates to a C++ function definition.

As an example

value
  sqr : **Real** → **Real**
  sqr(x) ≡ x∗x,

  + : V × **Int** → V
  v + i ≡ Vrec(i, v)

(in the context of the type V defined on page 34) translates to

```
double sqr(const double x_38B){
return x_38B * x_38B;
}

V RSL_PLUS_op(const V& v_4B3, const int i_4B7){
return Vrec(i_4B7, v_4B3);
}
```

Note that a user-defined operator translates into a function with a name derived from the operator name rather than into an operator. This eases the translation of such operators. In particular it makes them all translatable. The names are given in table 2.

Additional code, included if `RSL_pre` is defined, is generated if any parameter types are subtypes or if the function has a precondition.

Access descriptors are ignored. The kind of function arrow (→ or $\xrightarrow{\sim}$) does not matter.

Only one formal function parameter is accepted. It is not possible to translate

f : **Int** → **Int** → **Int**
f(x)(y) ≡ ...

It is not required that the number of parameters matches the number of components in the domain of the function's type expression. For example, the following are all accepted:

type
  U = **Int** × **Bool**
value
  f1: **Int** × **Bool** → **Bool**

Operator	Function name
$=$	RSL_EQ_op
$\neq$	RSL_NOTEQ_op
$==$	RSL_EQEQ_op
$>$	RSL_GT_op
$<$	RSL_LT_op
$\geq$	RSL_GEQ_op
$\leq$	RSL_LEQ_op
$\supset$	RSL_PSUP_op
$\subset$	RSL_PSUB_op
$\supseteq$	RSL_SUP_op
$\subseteq$	RSL_SUB_op
$\in$	RSL_ISIN_op
$\notin$	RSL_NOTISIN_op
$\backslash$	RSL_MOD_op
$\frown$	RSL_CONC_op
$\cup$	RSL_UNION_op
$\dagger$	RSL_OVER_op
$*$	RSL_AST_op
$/$	RSL_DIV_op
$\circ$	RSL_HASH_op
$\cap$	RSL_INTER_op
$\uparrow$	RSL_EXP_op
abs	RSL_ABS_op
int	RSL_INT_op
real	RSL_REAL_op
card	RSL_CARD_op
len	RSL_LEN_op
inds	RSL_INDS_op
elems	RSL_ELEMS_op
hd	RSL_HD_op
tl	RSL_TL_op
dom	RSL_DOM_op
rng	RSL_RNG_op
$+$	RSL_PLUS_op
$-$	RSL_MINUS_op

Table 2: Function names for user-defined operators

f1(x, y) ≡ ...,
f2: (**Int** × **Bool**) → **Bool**
f2(x, y) ≡ ...,
f3: U → **Bool**
f3(x, y) ≡ ...,
f4: U × **Int** → **Bool**
f4(x, y) ≡ ...,
f5: (**Int** × **Bool**) × **Int** → **Bool**
f5(x, y) ≡ ...
f6: (**Int** × **Bool**) × **Int** → **Bool**
f6(x) ≡ ...

f7: **Int** × **Bool** → **Bool**
f7(x) ≡ ...,
f8: (**Int** × **Bool**) → **Bool**
f8(x) ≡ ...,
f9: U → **Bool**
f9(x) ≡ ...,
f10: U × **Int** → **Bool**
f10((x, y), z) ≡ ...,
f11: (**Int** × **Bool**) × **Bool** → **Bool**
f11((x, y), z) ≡ ...

Implicit function definitions Implicit function definitions are accepted with a warning that they are incomplete.

11.4 Variable declarations

A variable declaration translates to a sequence of variable declarations corresponding to its variable definitions.

variable n1, n2 : **Int**, z1, z2 : Complex, seq : **Int***, f : **Int** → **Int**

translates to

```
int n1;
int n2;
Complex z1;
Complex z2;
RSL_lI seq;
RSL_IfI f;
```

Note that in RSL the initial values of these variables are underspecified within their (sub)types. In C++ the variables will be uninitialised.

An initialisation of a variable translates to the corresponding C++ initialisation. Additional code, included if `RSL_pre` is defined, is generated if any types of initialised variables are subtypes, to check that the initial values are in the subtypes.

11.4.1 Channel declarations

Not accepted.

11.4.2 Axiom declarations

Not accepted.

11.5 Class expressions

A class expression translates to the declarations and statements which the translation of the contents of the class expression results in.

11.5.1 Basic class expressions

A basic class expression translates as its declarations.

11.5.2 Extending class expression

An extending class expression translates as the two class expressions.

11.5.3 Hiding class expressions

Hiding is ignored, and a warning given: hidden names are fully visible in the C++ code.

11.5.4 Renaming class expression

Renaming is ignored, and a warning given: all the references to the renamed names use the identifiers in their original definition. This may cause problems with name clashes in the C++ code.

11.5.5 With expression

Objects are translated as namespaces, so with expressions are translated using the C++ declaration `using namespace`. (This allows qualified names to be used without qualification as long as there is no resulting ambiguity.)

11.5.6 Scheme instantiations

A scheme instantiation translates as the unfolded scheme with substituted parameters.

11.6 Object expressions

An object expression which is a name is accepted as an actual scheme parameter or as a qualification.

A fitting object expression is accepted as an actual scheme parameter.

Neither element object expressions nor array object expressions are accepted.

11.7 Type expressions

The translation of a type expression depends on its kind as described below.

11.7.1 Type literals

The type literals are translated as shown in table 3.

RSL	C++
Unit	not generally accepted
Bool	`bool`
Int	`int`
Nat	`int`
Real	`double`
Text	`RSL_string`
Char	`RSL_char`

Table 3: Translation of RSL type literals

The type **Unit** is only accepted as complete parameter or result type in a function type expression. As a result type it translates to `void`.

The types `RSL_string` and `RSL_char` are defined in the RSL library. There are constructors `RSL_string` and `RSL_char` that convert `string` and `char` arguments respectively to `RSL_string` and `RSL_char` values, and an overloaded destructor `RSL_to_cpp` that converts them back to `string` and `char`. So we have the following equivalences, for example:

```
RSL_to_cpp(RSL_string("abc")) == "abc"
RSL_to_cpp(RSL_char('a')) == 'a'
```

These functions make it easy to combine translated RSL with hand-written C++ code.

Note that the function `RSL_to_string` (section 11.13.1) is not quite the same as `RSL_to_cpp`. `RSL_to_string` is intended to generate strings suitable for output (from any type), and produces a string that could be parsed as RSL. For example:

```
RSL_to_string(RSL_string("abc")) == "\"abc\""
```

```
RSL_to_string(RSL_char('a')) == "'a'"
```

11.7.2 Names

A type name that is not an abbreviation translates as a name. A type name that is an abbreviation translates as the abbreviation.

11.7.3 Product type expressions

A product type expression translates as its universal type name (see section 11.12).

11.7.4 Set type expressions

A set type expression translates as its universal type name (see section 11.12).

There is a template `RSLSet` defined in the RSL library files, and if type T translates to the C++ type `Tc`, then T-**set** and T-**infset** both translate to the universal type name `RSL_sTc`, which is `typedef`ed to `RSLSet<Tc>`.

11.7.5 List type expressions

A list type expression translates as its universal type name (see section 11.12).

There is a template `RSLList` defined in the RSL library files, and (except for **Char**) if type T translates to the C++ type `Tc`, then T^* and T^ω both translate to the universal type name `RSL_lTc`, which is `typedef`ed to `RSLList<Tc>`.

Text, **Char***, and **Char**$^\omega$ translate to `RSL_string`. Standard RSL list operators like **hd** are defined for `RSL_string` in an RSL library file. `RSL_string` values are easily converted to and from `string` values using the functions `RSL_to_cpp` and `RSL_string` respectively: see section 11.7.1.

11.7.6 Map type expressions

A map type expression translates as its universal type name (see section 11.12).

There is a template `RSLMap` defined in the RSL library files, and if types T and U translate to the C++ types `Tc` and `Uc`, then T $\xrightarrow[m]{}$ U and T $\xrightarrow{\sim}$ U both translate to the universal type name `RSL_TcmTu`, which is `typedef`ed to `RSLMap<Tc,Uc>`.

11.7.7 Function type expressions

A function type expression translates as its universal type name (see section 11.12).

11.7.8 Subtype expressions

A subtype expression translates as its maximal type.

11.7.9 Bracketed type expressions

A bracketed type expression translates as its constituent type expression.

11.8 Value expressions

11.8.1 Evaluation order

The evaluation order in RSL is left-to-right. In C++ it is often not specified. So we need to be careful when translating expressions that are not readonly. For example, if any of the expressions e1, e2, or e3 is not readonly, then the application f(e1, e2, e3) is translated as if it had been written

let x1 = e1, x2 = e2 **in** f(x1, x2, e3) **end**

This will translate in C++ to something like

```
t1 x1 = e1;
t2 x2 = e2;
f(x1, x2, e3)
```

As well as function applications, a similar approach is taken for enumerated sets, lists and maps.

11.8.2 Value literals

A value literal of type **Bool**, **Char**, **Int**, **Nat**, or **Real** translates to the corresponding constant.

A value literal of type **Text** translates to the corresponding string.

A value literal of type **Unit** is ignored, or translated as the empty statement, or not accepted, depending on the context.

11.8.3 Names

A value name translates as a name (see section 11.10 on names).

11.8.4 Pre names

Not accepted.

11.8.5 Basic expressions

The only basic expression in $\text{RSL}_{\text{C++}}$ is **skip**, which translates to the empty statement.

11.8.6 Product expressions

A product expression translates to an expression using the appropriate constructor. For example, (1,**true**) will translate as `RSL_IxB(1, true)`.

11.8.7 Set expressions

A set expression translates to the appropriate `RSLSet` function call as shown in the examples below. $\text{RSL}_{\text{C++}}$ includes ranged set expressions and enumerated set expressions, and some comprehended set expressions. A ranged set expression, such as {2 .. 7}, translates to `init_ranged_set(2, 7)`, where `init_ranged_set` is defined by

```
static RSL_sI init_ranged_set(const int l_, const int r_){
RSL_sI s_;

s_ = RSL_sI();
for (int i_ = r_;
 i_ >= l_; i_--) {
s_ = RSL_sI(i_, s_);
}
return s_;
}
```

An enumerated set expression, such as {1,4,7}, translates to
`RSL_sI(1, RSL_sI(4, RSL_sI(7, RSL_sI())))`.

Note that the translator can only translate a set expression if its type can be uniquely determined from the context, i.e. it is not possible to translate the expression {} = {} without some disambiguation.

Comprehended set expressions A comprehended set expression can only be translated if it takes one of the forms

{ e | b : T • b ∈ slm }
or
{ e | b : T • b ∈ slm ∧ p }

where b is a binding, T is a type, e is a translatable expression, slm is a translatable expression of set, list, or map type, and p is a translatable expression of type **Bool**.

For example, the expression { 2∗i | i : **Int** • i ∈ s ∧ i > 0 }, where s is of type **Int-set**, translates to

```
namespace RSL_Temp_3 {
// namespace for comprehended set
int RSL_Temp_4(const int i_3F1){
return 2 * i_3F1;
}

bool RSL_Temp_5(const int i_3F1){
return i_3F1 > 0;
}

} // end of namespace RSL_Temp_3

RSL_sI RSL_test_case_RSL_Temp_2(){
return RSL_compss<int, int>(RSL_Temp_3::RSL_Temp_4, RSL_Temp_3::RSL_Temp_5, s);
}
```

`RSL_compss` is a template function defined in a standard RSL library files to generate a set from a set. Its first two parameters are functions, one to generate the expression e in the comprehended set (here `RSL_Temp_4`), and one to evaluate the predicate p (here `RSL_Temp_5`). The third parameter is the expression slm (here the set s).

C++ does not allow functions to be defined locally to other functions or expressions, so `RSL_Temp_4` and `RSL_Temp_5` have to be defined externally, and they are placed in their own namespace. The use of a namespace is not essential, but the approach is close to that used for **local** expressions described in section 11.8.23. This results in a restriction that comprehended set expressions may not occur in recursive functions where the recursion is through the expression e or the predicate p. There are similar restrictions for comprehended lists and maps, for quantified expressions, and for implicit let expressions.

The reason for this restriction is that the extra functions needed for the expression e and the predicate p must be placed out of their original scope. But they may refer, for example, to the parameters of the function they are used in. So function parameters, and other locally defined names like let bindings, have to be copied to the namespace in which the extra functions are defined. But this technique is essentially static, and recursive calls of the same function would result in these copied variables being changed unpredictably.

11.8.8 List expressions

The translation of a list expression depends on the context. As a component in a **for** expression it translates as described there. In other contexts, a list expression translates to the appropriate `RSLList` function call as shown in the examples below.

RSL_{C++} includes ranged list expressions and enumerated list expressions, and some comprehended list expressions. A ranged list expression, such as ⟨2 .. 7⟩, translates to `init_ranged_list(2, 7)`, where `init_ranged_list` is defined by

```
static RSL_sI init_ranged_list(const int l_, const int r_){
RSL_lI lst_;

lst_ = RSL_lI();
for (int i_ = r_;
 i_ >= l_; i_--) {
lst_ = RSL_lI(i_, lst_);
}
return lst_;
}
```

An enumerated list expression, such as ⟨1,4,7⟩, translates to
`RSL_lI(1, RSL_lI(4, RSL_lI(7, RSL_lI())))`.

Note that the translator can only translate a list expression if its type can be uniquely determined from the context, i.e. it is not possible to translate the expression ⟨⟩ = ⟨⟩ without some disambiguation.

A comprehended list expression can only be translated if it takes one of the forms

⟨ e | b **in** l ⟩
or
⟨ e | b **in** l • p ⟩

where b is a binding, e is a translatable expression, l is a translatable expression of list type, and p is a translatable expression of type **Bool**. The translation is very similar to that for comprehended sets: see section 11.8.7. Therefore comprehended list expressions may not occur in recursive functions where the recursion is through the expression e or the predicate p.

11.8.9 Map expressions

A map expression translates to the appropriate `RSLMap` function call as shown in the examples below. RSL_{C++} includes enumerated map expressions and some comprehended map expressions.

[**false** ↦ 0, **true** ↦ 1] translates to
`RSL_BmI(false, 0, RSL_BmI(true, 1, RSL_BmI()))`.

Note that the translator can only translate map expressions if their type can be uniquely determined from the context, i.e. it is not possible to translate the expression {} = **dom** [ ] without some disambiguation.

A comprehended map expression can only be translated if it takes one of the forms

[e1 ↦ e2 | b :T • b ∈ slm]
or
[e1 ↦ e2 | b :T • b ∈ slm ∧ p]

where b is a binding, e1 and e2 are translatable expressions, T is a type, slm is a translatable expression of set, list, or map type, and p is a translatable expression of type **Bool**. The translation is very similar to that for comprehended sets: see section 11.8.7. Therefore comprehended map expressions may not occur in recursive functions where the recursion is through either of the expressions e1 or e2 or the predicate p.

11.8.10 Function expressions

Not accepted.

11.8.11 Application expressions

An application expression may be translated to a function call, a list application or a map application.

A function application translates to a function call. This includes calls of constructors, destructors and reconstructors. Note that as a destructor translates into an inline function an application will translate directly into a C++ class member access.

A list application translates to an element selection: l(i) translates to `l[i]`. A map application translates to an element selection: m(b) translates to `m[b]`

11.8.12 Quantified expressions

Quantified expressions can only be translated if they take one of the following forms:

∀ b : T • b ∈ slm
∀ b : T • b ∈ slm ⇒ p
∀ b : T • b ∈ slm ∧ p ⇒ q
∃ b : T • b ∈ slm
∃ b : T • b ∈ slm ∧ p
∃! b : T • b ∈ slm
∃! b : T • b ∈ slm ∧ p

where b is a binding, T is a type, slm is a translatable expression of set, list or map type, and p and q are translatable expressions of type **Bool**.

The translation involves generating functions which must be defined at the top level. Like the translation of comprehended expressions (section 11.8.7), this means that quantified expressions may not occur in recursive functions where the recursion is through the predicates p or q.

This restriction can require the user to rewrite functions involving quantifiers. For example, consider the common case of a map used to model a relation which may be transitive but should not be cyclic. A particular instance is the "bill of materials" where the map models the "parts" relation. We use natural numbers to model part identifiers.

```
type Bom = Nat m→ Nat-set

value
  parts : Nat × Bom → Nat-set
  parts(i, m) ≡
    {j | j : Nat • j ∈ m ∧ part_of(j, i, m)},

  part_of : Nat × Nat × Bom → Bool
  part_of(j, i, m) ≡
    i ∈ m ∧
    (j ∈ m(i) ∨
     (∃ k : Nat • k ∈ m(i) ∧ part_of(j, k, m))),

  non_circular : Bom → Bool
  non_circular(m) ≡ (∀ i : Nat • i ∈ m ⇒ ∼part_of(i, i, m))
```

The function parts gives the "explosion" of a part to give all its sub-parts.

The function part_of contains a quantified expression, and is also recursive through its predicate. So we must rewrite it for translation.

A standard technique is to replace the quantified expression with a separate function that uses a loop to calculate the same result. We can redefine part_of using a second function:

```
  part_of : Nat × Nat × Bom → Bool
  part_of(j, i, m) ≡
    i ∈ dom m ∧ (j ∈ m(i) ∨ part_of1(j, m(i), m\{i})),

  part_of1 : Nat × Nat-set × Bom → Bool
  part_of1(j, s, m) ≡
      if s = {} then false
      else
        let k = hd s in
          part_of(j,k,m) ∨ part_of1(j,s\{k},m) end
      end
```

The two part_of functions are mutually recursive, but this recursion is bound to terminate (even if the relation is circular). Each recursive call of part_of (through part_of1) reduces the Bom parameter (and it must terminate when this parameter is empty, since the first conjunct in its definition will be false). Each recursive call of part_of1 reduces the set parameter, and it terminates when this set is empty.

11.8.13 Equivalence expressions

Not accepted.

11.8.14 Post expressions

Not accepted.

11.8.15 Disambiguation expressions

A disambiguation expression translates as its constituent value expression.

11.8.16 Bracketed expressions

A bracketed expression translates to a bracketed expression.

11.8.17 Infix expressions

An infix expression generally translates to the corresponding C++ expression. A statement infix expression translates to a statement. The only infix combinator that is accepted is semicolon (;). Note that the semicolon is a combinator in RSL and a kind of terminator in C++.

An axiom infix expression translates as the equivalent if expression, as shown in table 4.

x $\vee$ y	**if** x **then true else** y **end**
x $\wedge$ y	**if** x **then** y **else false end**
x $\Rightarrow$ y	**if** x **then** y **else true end**

Table 4: Translation of logical connectives

A value infix expression translates to either an expression or a function call. User-defined operators are translated into function calls using the function names listed in section 11.3.4 The built-in infix operators are translated as indicated in table 5.

11.8.18 Prefix expressions

A prefix expression generally translates to the corresponding C++ expression.

An axiom prefix expression translates to an expression: $\sim$ translates to `!`.

A universal prefix expression ($\Box$) is not accepted by the translator.

RSL	C++
x = y	`x == y`
x ≠ y	`x != y`
x > y	`x > y`
x < y	`x < y`
x ≥ y	`x >= y`
x ≤ y	`x <= y`
x ⊃ y	`x > y`
x ⊂ y	`x < y`
x ⊇ y	`x >= y`
x ⊆ y	`x <= y`
x ∈ y	`isin(x, y)`
x ∉ y	`!isin(x, y)`
x + y	`x + y`
x − y	`x - y`
x ∗ y	`x * y`
x / y	`x / y`
x ∩ y	`x * y`
x ⌢ y	`x + y`
x ∪ y	`x + y`
x † y	`x + y`
x \ y	`x % y`
x ° y	not accepted

Table 5: Translation of built-in infix operators

A value prefix expression translates to a function call, using the function names in table 6.

RSL	C++
abs	`RSL_abs`
int	`RSL_int`
real	`real`
card	`card`
len	`len`
inds	`inds`
elems	`elems`
hd	`hd`
tl	`tl`
dom	`dom`
rng	`rng`

Table 6: Translation of built-in prefix operators

The operators **abs** and **int** are renamed to avoid collision with the standard C function `abs` and the type `int`. The C++ functions are all defined in the RSL library files.

11.8.19 Initialise expressions

Not accepted.

11.8.20 Assignment expressions

An assignment expression translates to an assignment statement: x := e translates to `x = e;`.

Additional code, included if `RSL_pre` is defined, is generated if the type of x is a subtype, to check that e isin the subtype.

11.8.21 Input expressions

Not accepted.

11.8.22 Output expressions

Not accepted.

11.8.23 Local expressions

Local expressions that only declare variables and constant values are translated with the corresponding C++ definitions in-line.

Local expressions that declare functions need a special treatment, as functions may not be declared inside statements or expressions in C++.

So the C++ definitions arising from the declarations of a local expression that include functions are placed outside the definition in which it occurs. In fact they are placed inside their own namespace, as we shall see below.

Moving the declarations out of their original scope means that local bindings, in particular formal function parameters, will no longer be visible. To counter this, formal function parameters are defined as variables in the namespace and initialised from the original position of the local expression. Local explicit values are defined as variables and assigned their defining values after the parameter variables. Local variables are then assigned their initial values. This allows local values to depend on parameters, and the initial values of local variables to depend on parameters and local values.

As with comprehended expressions (section 11.8.7) this technique is essentially static and cannot deal with recursive functions. So recursive functions that contain local expressions defining functions are not accepted.

11.8.24 Let expressions

An explicit let expression translates to a number of variable declarations (for the identifiers introduced in the bindings) and the translation of the constituent value expression.

A simple let expressions like

let x = 1 **in** k := x + 1 **end**

translates to

```
int x_123 = 1;
k = x_123 + 1;
```

In the scope of

type Complex :: **Real Real**

the let expression (1)

let mk_Complex(i, j) = z **in** k:= **int** (i ∗ i) + **int** (j ∗ j) **end**

translates to

```
double i_B00 = z.RSL_f1;
double j_B02 = z.RSL_f2;
k = RSL_int(i_B00 * i_B00) + RSL_int(j_B02 * j_B02);
```

If the type Complex were a variant type rather than a record, the translation would be

```
if (!(z.RSL_tag == RSL_mk_Complex_tag))
{
RSL_fail("X.rsl:m:n: Let pattern does not match")
}
double i_B00 = (((RSL_mk_Complex_type*)z.RSL_ptr)->RSL_f1);
double j_B02 = (((RSL_mk_Complex_type*)z.RSL_ptr)->RSL_f2);
k = RSL_int(i_B00 * i_B00) + RSL_int(j_B02 * j_B02);
```

RSL_fail is a function that writes a message to standard output (if RSL_io is set) and then aborts.

In a record pattern such as used in (1), the function (here mk_Complex) must be the constructor of a record or variant.

An implicit let expression can only be translated if it has one of the forms

let b : T • b ∈ slm **in** ... **end**
or
let b : T • b ∈ slm ∧ p **in** ... **end**

where b is a binding, T a type, slm a translatable expression of set, list, or map type, and p a translatable expression of type **Bool**.

For example,

let x : **Int** • x ∈ s ∧ x > 0 **in** k := x **end**

(where s is a set) translates to

```
namespace RSL_Temp_16 {
// namespace for implicit let
bool RSL_Temp_15(const int x_B59){
return x_B59 > 0;
}
} // end of namespace RSL_Temp_16

int RSL_test_case_RSL_Temp_14(){
int x_B59 = RSL_chooses<int>(RSL_Temp_16::RSL_Temp_15, s);
k = x_B59;
}
```

If no suitable value can be found the RSL_chooses function will fail with the message "Choose from empty set".

The translation is similar to that for quantified expressions described in section 11.8.12, and for the same reason an implicit let cannot occur in a recursive function where the recursion is through the predicate p.

11.8.25 If expressions

An if expression translates to an if statement or an if expression. An if expression is used if there are no **elsif** branches and the **then** and **else** expressions translate without generating any statements.

Elsif branches translate to nested if statements.

11.8.26 Case expressions

A case pattern translates in general to a condition that the case expression matches the pattern, and one or more variable declarations (for the identifiers introduced in the pattern). The conditions are generally more complicated than can be handled by switch statements, and so if statements are used instead.

A temporary variable is used to ensure the expression being cased on is evaluated first and only once.

For example,

k :=
  case x **of**
    2 → 0,
    1 → 1,
    _ → 2
  end

translates to

```
int RSL_Temp_18 = x;

int RSL_Temp_19;

if (RSL_Temp_18 == 2)
{
RSL_Temp_19 = 0;
}
else
{
if (RSL_Temp_18 == 1)
{
RSL_Temp_19 = 1;
}
else
{
RSL_Temp_19 = 2;
}
}
k = RSL_Temp_19;
```

In the scope of

type
  V ==
    Vconst |
    Vint(**Int**) |
    Vrec(d1Vrec : **Int** ↔ r1Vrec, d2Vrec : V ↔ r2Vrec)
variable v : V, j : **Int**

the case expression

case v **of**

Vconst → j := 5,
Vrec(a,_) → j := a
end

translates to

```
V RSL_Temp_1 = v;

if (RSL_Temp_1 == Vconst)
{
j = 5;
}
else
{
if (RSL_Temp_1.RSL_tag == RSL_Vrec_tag)
{
int a_454 = (((RSL_Vrec_type*)RSL_Temp_1.RSL_ptr)->RSL_f1);
j = a_454;
}
else
{
RSL_fail("X.rsl:m:n: Case exhausted");
}
}
```

RSL_fail is a function that writes a message to standard output (if RSL_io is set) and then aborts.

In a record pattern the function must be the constructor of a record or variant.

11.8.27 While expressions

A while expression translates to a for statement. For example

while j ≥ k **do** j := j − 1 **end**

translates to

```
for ( ; ; ) {
if (!(j >= k))
{
break;
}
j = j - 1;
}
```

11.8.28 Until expressions

An until expression translates to a do statement. For example

do j := j + 1 **until** j > k **end**

translates to

```
do {
j = j + 1;
} while (!(j > k));
```

11.8.29 For expressions

A for expression translates to a block statement that contains the corresponding C++ for statement. The block statement is introduced to limit the scope of the loop variable and possibly extra control variables. If the list expression is a ranged list expression, the translation does not include introduction of list variables, since there is an obvious simple translation. For example

for i **in** ⟨2..5⟩ • i ≥ 4 **do** k := k + i **end**

translates to

```
{
for (int i_25D = 2;
 i_25D <= 5; i_25D++) {
if (i_25D >= 4)
{
k = k + i_25D;
}
}
}
```

Note that the scope of a for expression in RSL is not the same as the scope of a for statement in C++: care needs to be taken if the right limit is not pure, when it could be affected by the body of the loop. For example, if j is a variable,

for i **in** ⟨j..j+5⟩ • i ≥ 4 **do** j := j + i **end**

translates to

```
{
int RSL_Temp_0 = j + 5;

for (int i_2C1 = j;
 i_2C1 <= RSL_Temp_0; i_2C1++) {
if (i_2C1 >= 4)
{
j = j + i_2C1;
}
}
}
```

If the list expression is an enumerated list expression, the translation introduces an array variable to hold the values.

for i **in** ⟨1,3,5⟩ • i ≥ 4 **do** k := k + i **end**

translates to

```
{
int i_325;

int RSL_Temp_0[3];

RSL_Temp_0[0] = 1;
RSL_Temp_0[1] = 3;
RSL_Temp_0[2] = 5;
for (int x_ = 0;
 x_ < 3; x_++) {
i_325 = RSL_Temp_0[x_];
if (i_325 >= 4)
{
k = k + i_325;
}
}
}
```

If the list expression is neither a ranged list expression nor an enumerated list expression, it translates in the obvious way.

for i **in** l • i ≥ 4 **do** k := k + i **end**

translates to

```
{
```

```
int i_389;

RSL_lI list_ = l;

int len_ = len(list_);

for (int x_ = 1;
 x_ <= len_; x_++) {
i_389 = list_[x_];
if (i_389 >= 4)
{
k = k + i_389;
}
}
}
```

An auxiliary variable `list_` is always introduced to contain the list expression. This ensures the list expression is evaluated first and only once.

11.9 Bindings

A binding, which must be an id or op, translates as its constituent id or op. Except at the top level, bindings are translated with a unique extension.

11.10 Names

A name which is a qualified identifier or operator translates as the translated id prefixed with full qualification. For example, A.f will translate as `A::f`, and A.+ as `A::RSL_PLUS_op`.

11.11 Identifiers and operators

An identifier translates to the same identifier, or, if it is from a binding in an inner scope, to the same identifier plus a unique extension.

Identifiers that collide with C++ reserved words are not allowed. Failure to observe this rule is likely to cause compilation errors: it is not detected by the translator.

Users should avoid using identifiers beginning or ending with the 3 characters "rsl" (in any combinations of upper and lower case). Many generated names start `RSL_`. The character ‘ is converted to `Rsl`, and the character ’ to `rsL`. Guards for header files terminate in `_RSL`.

Built-in operators are translated as in the tables in sections 11.8.17 and 11.8.18. User-defined operators are translated as identifiers according to the table on page 45.

11.12 Universal types

The C++ translator generates type names, known as universal types, from the structure of the maximal type. The universal types are introduced in order to cope with RSL's maximal type equivalence.

The names of the universal types are constructed from the structure of the maximal types they represent. Additionally they are given the prefix `RSL_` to ensure that they differ from other names. The different components and type constructors of a type expression are represented in the name of the universal type as described in table 7.

RSL	Representation
Unit	`U`
Int	`I`
Bool	`B`
Real	`R`
Char	`C`
Text	`lC`
Id	`Id_NNN`
Q.Id	`Id_NNN`
$\times$	`-x-`
-set	`s-`
-infset	`s-`
*	`l-`
$^\omega$	`l-`
$\overrightarrow{m}$	`-m-`
$\overset{\sim}{\overrightarrow{m}}$	`-m-`
$\rightarrow$	`-f-`
$\overset{\sim}{\rightarrow}$	`-f-`
( )	`6-9`

Table 7: Construction of universal type names

A name is represented by the original name plus a unique extension (suggested by `NNN` in the table). Qualifiers are ignored. An example is the type expression

Int $\times$ **Char** $\overrightarrow{m}$ (**Bool-set** $\times$ **Real**)*

which becomes `RSL_IxCml6sBxR9`

11.13 Input/output handling

The C++ translator optionally generates code for stream input and output of values of RSL data types.

The i/o routines provide a primitive but easy way of communicating values to and from a program based on translator generated C++ code. The routines are adequate for i/o in a prototype or during debugging. However, due to their lack of error handling at input and formatting at output, they are probably not adequate for handling of interactive i/o in production code.

Non-interactive i/o from and to files etc. can easily be handled.

The i/o facility is based on the C++ concept of streams as described in the standard library header `iostream.h`. When the C++ code is compiled with the the flag `RSL_io` defined, each translated type T is equipped with operators for output and input:

```
struct T {
...
ostream& operator <<(ostream&, const T&);
istream& operator >>(istream&, T&);
...
}
```

Streams can be connected to files. For interactive i/o, the standard streams `cin`, `cout` and `cerr` correspond to standard input, output and error.

Continuing the example, a value tval specified as having type T, can be printed on the standard output with

```
cout << "The value is: " << tval << "\n";
```

Note how RSL data type values can be freely mixed with ordinary C++ values, e.g. strings.

The following code inputs a value of type T to the variable tvar:

```
T tvar;
cout << "Give a value of type V:\n";
cin >> tvar;
```

The user is prompted for a string that can be parsed and interpreted as a literal of type T. If the string obeys the input syntax described below, the literal value is assigned to tvar. Otherwise, tvar is unchanged and the state of the stream is set to ios::badbit.

The i/o facility works for any translated type, no matter how complex.

To ensure smooth integration of handwritten C++ with translator generated code, any user defined type, UD, should come equipped with operators for input and output:

```
struct UD {
...
#ifdef RSL_io

ostream& operator <<(ostream&, const UD&);
istream& operator >>(istream&, UD&);
#endif
...
}
```

11.13.1 Input syntax

The following describes the syntax that input strings must obey in order to be parsed as values of the given type. For all types, only literal values are accepted. Whitespaces, i.e. blanks, tabs and newlines, can be freely added between lexical tokens. Note, for example, that the list delimiters `<.` and `.>` are two tokens, not four, so `<.7,9,13. >` will not be accepted.

Sort types As a sort is translated into an empty class, the generated i/o functions have no effect.

Variant types The input syntax is the same as the RSL syntax for variant literals if the constructors are identifiers. As an example the strings

```
Vconst
Vint(7)
Vrec(8,Vrec(9,Vint(10)))
```

can be interpreted as values of type

type
V ==
Vconst |
Vint(**Int**) |
Vrec(d1Vrec : **Int** ↔ r1Vrec, d2Vrec : V ↔ r2Vrec)

Internal buffer size limits the number of characters in a constructor identifier to 128.

Union types Not accepted by the translator, hence no i/o.

Product types The input syntax is the same as the RSL syntax for product literals. As an example the string `(1, Vint(7), "Margrethe")` can be interpreted as a value of type **Int** × V × **Text** where V is the variant type defined above.

Set types The input syntax is the same as the RSL syntax for set literals. As an example the strings

```
{1,4,9}
{1 .. 100}
```

can be interpreted as values of type **Int-set**.

List types The input syntax is the same as the ASCII version of the RSL syntax for list literals. As an example the strings

```
<.1,1,4,4,9,9.>
<.1 .. 100.>
```

can be interpreted as values of type **Int***.

Text type The input syntax is the same as the RSL syntax for text literals. As an example the string `"Margrethe"` can be interpreted as a value of type **Text**. Note that the quotes must be present. The length of the string must not exceed 256 characters.

Map types The input syntax is the same as the ASCII version of the RSL syntax for map literals. As an example the string

```
[ 1 +> "en", 2 +> "to", 3 +> "tre" ]
```

can be interpreted as a value of type **Int** $\overrightarrow{m}$ **Text**.

Unit type The translator only accepts the unit type in certain contexts, hence no i/o.

Int type As RSL **Int** is translated to C++ `int`, integers are read via the standard int input operator. Therefore, the input syntax differs from the RSL syntax. A unary minus as in -4 can be used whereas e.g. 0-4 is not accepted as it is not a C++ integer literal. Be careful not to enter integer literals that are numerically too large to be represented as `int`s on the target machine. These will be truncated in a machine dependent way.

Nat type As **Nat** is translated to `int`, the rules for **Int** apply to **Nat** as well.

Note that it is possible to input a negative value to a variable that originally was specified as **Nat**. This can be cause for errors.

Bool type RSL **Bool** is translated to C++ `bool`, and the literal values `true` and `false` are interpreted as the corresponding RSL literals.

Real type As RSL **Real** is translated to C++ `double`, floating point numbers are read via the standard double input operator. Therefore, the input syntax differs from the RSL syntax.

Integer literals can be used at input. E.g. 4 is equivalent to 4.0. Exponential notation can be used, e.g. 1.234E-56. A unary minus as in -4.0 can be used whereas e.g. 0.0-4.0 is not accepted as it is not a C++ double literal.

Be careful not to enter literals that are numerically too large to be represented as doubles on the target machine, or contain too many digits to be represented exactly. These will be truncated in a machine dependent way.

Char type The input syntax is the same as the RSL syntax for character literals. As an example the string `'a'` can be interpreted as a value of type **Char**.

Escaped characters are supported. For example, `'\t'` is interpreted as a tab character, `'\''` as a quote, `'\141'` (octal notation) and `'\x61'` (hexadecimal notation) both as `'a'`.

Output syntax All values are converted to output strings with a syntax that is acceptable for input, with the following exceptions:

- The GNU g++ compiler is not fully ANSI compliant, and in particular (before version 3) did not accept the "boolalpha" conversion. **true** and **false** are output as 1 and 0 respectively. If your C++ compiler does accept the "boolalpha" conversion, then you should define the variable `RSL_boolalpha` when invoking the compiler.
- On some machines, very large floating point literals will be represented as infinity. The output form of infinity is `Inf` which is not an acceptable input form.

There is no way to format the output, e.g. to break a long list over several lines.

Conversion to strings A more flexible way of generating outputs has also been added. The translated C++ code includes the definition of an overloaded function `RSL_to_string` that will convert any RSL value to a string value. This makes it easier to introduce some formatting.

When using `RSL_to_string` in hand-written code for a non-built-in type, it is necessary to mention the required type in the RSL code so that `RSL_to_string` is defined for that type. Defining an abbreviation for the type will suffice.

11.14 An example

This section explains how to translate and test the following specification of quicksort.

The parameter for QUICKSORT is the scheme QSP:

```
scheme QSP =
  class
    type Elem

    value
      ≤ : Elem × Elem → Bool
  end
```

The parameterised QUICKSORT scheme is

QSP

```
scheme QUICKSORT(X : QSP) =
  with X in class
    value
      sort : Elem* → Elem*
      sort(l) ≡
        case l of
          ⟨⟩ → ⟨⟩,
          ⟨h⟩ ⌢ t →
            let (t1, t2) = split(h, t, ⟨⟩, ⟨⟩) in
              sort(t1) ⌢ ⟨h⟩ ⌢ sort(t2)
            end
        end,

      split :
        Elem × Elem* × Elem* × Elem* → Elem* × Elem*
      split(x, t, t1, t2) ≡
        case t of
          ⟨⟩ → (t1, t2),
          ⟨h⟩ ⌢ t3 →
            if h ≤ x then split(x, t3, ⟨h⟩ ⌢ t1, t2)
            else split(x, t3, t1, ⟨h⟩ ⌢ t2)
            end
        end
  end
```

To specify quicksort for integers we can create an object I for the actual parameter:

```
object I : class type Elem = Int end
```

Quicksort is then scheme QI, which includes some test cases:

QUICKSORT, I

```
scheme QI =
  class
    object
      Q : QUICKSORT(I)
    test_case
      [t1]
        Q.sort(⟨⟩),
      [t2]
        Q.sort(⟨12, 45, 2, 4, 2, 8, −1, 0⟩)
  end
```

When this is compiled and the result executed, the output is as follows:

```
[t1] <..>
[t2] <.-1,0,2,2,4,8,12,45.>
```

Checking the results for many such tests would be tedious. A better way is to have a more abstract specification of QUICKSORT like QSPEC:

QSP

scheme QSPEC(X : QSP) =
  with X **in class**
    value
      /∗ commented out for translation
      sort : Elem* → Elem*
      sort(l) as l′ post is_permutation(l, l′) ∧ sorted(l′), ∗/

      is_permutation : Elem* × Elem* → **Bool**
      is_permutation(l1, l2) ≡
        (∀ e : Elem •
          e ∈ **elems** l1 ∪ **elems** l2 ⇒
            count(l1, e) = count(l2, e)),

      count : Elem* × Elem → **Nat**
      count(l, e) ≡
        card {i | i : **Int** • i ∈ **inds** l ∧ l(i) = e},

      sorted : Elem* → **Bool**
      sorted(l) ≡
        let s = **inds** l **in**
          (∀ i : **Int** • i ∈ s ⇒
            (∀ j : **Int** • j ∈ s ⇒
              j > i ⇒ l(i) ≤ l(j)))
        end
  end

Then we can instantiate both QUICKSORT and its more abstract specification and use the latter to check the former:

QSPEC, QUICKSORT, I

scheme QSI =
  class
    object
      A : QSPEC(I),
      B : QUICKSORT(I)

```
    value
      check : Int* → Bool
      check(l) ≡
        let l1 = B.sort(l) in
          A.is_permutation(l, l1) ∧ A.sorted(l1)
        end

    test_case
      [t1] check(⟨⟩),
      [t2] check(⟨1⟩),
      [t3] check(⟨1, 4, 3, 2, 7, 4, 6, 3, 8, 100, −2, −5, 8, 200⟩)
end
```

Translating, compiling, and running QSI produces the output

```
[t1] true
[t2] true
[t3] true
```

This allows us to have many test cases (perhaps by random number generation) without the tedium of generating the expected results.

12 PVS translator

12.1 Introduction

The PVS translator was written by Aristides Dasso, as reported in [7].

The subset of RSL that is accepted by the translator is the applicative subset of RSL: it excludes variables, channels, and the sequential and concurrent combinators. It also excludes union types and object arrays.

12.1.1 Use

There are two main reasons for using the PVS translator:

1. To prove RSL confidence conditions.
2. To prove RSL theories and development relations

Confidence conditions Most confidence conditions appear as PVS *type correctness conditions* (TCCs) and so need not be generated by the translator. But a few will not become TCCs, and are generated as

LEMMAs (with names ending _cc$n$, where n is a number). So if you prove all the PVS TCCs and all such lemmas, you will have proved all the confidence conditions.

The confidence conditions that translate to PVS LEMMAs are:

- Enumerated maps have distinct domain elements.
- Explicit recursive functions return values in the result type, if this is a subtype.
- Case expressions are complete.
- Let patterns can be matched.

Theories and development relations Theories and development relations translate into PVS LEMMAs, and so are proved by proving the lemmas in PVS

12.1.2 Compilers and platforms

The translator is intended for use with PVS, available from `http://pvs.csl.sri.com/`. It will work with version 2.4 and above of PVS. PVS is currently only available for Solaris and Linux. It can be used free for non-commercial purposes.

12.1.3 Known errors and problems

There are no known errors.

Correctness of the translation is dependent on some conditions being fulfilled, which are generated either as *type correctness conditions* (TCCs) by PVS when it is run, or as lemmas corresponding to some RSL confidence conditions. See the discussion in section 12.2.3.

During translation, error messages may be generated with the standard `file:line:column` format showing from where in the RSL specification the message was generated. The cause of the error must be corrected and the translator run again.

12.2 Activating the PVS translator

The translator is activated from a shell with the command

```
rsltc -pvs <file>
```

where `<file>` takes the form `X` or `X.rsl` and contains the RSL scheme, object, theory or development relation named `X`. It generates a file `X.pvs`, plus perhaps other PVS files generated from RSL files in the context of `X`. For each such file `Y.rsl`, any corresponding pvs file will be called `Y.pvs`.

`X.pvs` can be loaded into pvs using the shell command

```
pvs X.pvs
```

provided the current directory is that where `X.pvs` is stored.

12.2.1 RSL prelude

PVS sets up a *PVS context* in each directory where it is used on PVS files. Such a PVS context needs also to load a file `rsl_prelude.pvs`. This RSL prelude is a library of definitions in PVS that are used by the translator, plus some theorems that may be used in proofs.

Setting up the RSL prelude Before loading the RSL prelude for the first time you need to run PVS on it to set up some auxiliary files. All you need to do is start PVS in the directory where `rsl_prelude.pvs` is stored, use the command

```
load-prelude-library <pvs_path>/lib/finite_sets
```

where `<pvs_path>` is the directory where PVS is stored, to load the PVS finite sets library, load the file `rsl_prelude.pvs`, run the PVS type checker, and exit.

You only need to do this once, unless you move to a new version of PVS, when you might have to do it again to update the auxiliary files.

Loading the RSL prelude When you create a context for a directory in which you wish to store PVS files generated from RSL ones, you need to load the RSL prelude. You do this with the PVS command

```
load-prelude-library <path>
```

where `<path>` is the path of the directory where `rsl_prelude.pvs` is stored. You only need to do this once for each context. In PVS you issue a command using the Meta key (usually `Esc`) followed by `x`, and then typing the command in the minibuffer.

12.2.2 Extending the RSL prelude

The RSL prelude is a natural place to add theorems about RSL that you find useful. You can do this by adding such theorems to `rsl_prelude.pvs`, but a perhaps better way is to create your own prelude file in the same directory. You could use the translator to generate the file from RSL. Since libraries are loaded by directory rather than file, your file will be loaded automatically in PVS contexts in which you have issued the `load-prelude-library` command. We would be interested to receive such extensions — with their proofs! — and perhaps include them in the RSL prelude in later releases.

12.2.3 Correctness

Correctness is important for any translator, but particularly so when the idea is to enable proofs of properties of the translated specification. Correctness is defined in terms of soundness: an RSL specification

is correctly translated if the translation is *sound*, which means that no theorem can be proved in PVS which is not valid for the RSL specification. We would also like the translation to be *complete*, i.e. we would like all true theorems to be provable. This is not easy to show — it depends on the power of PVS as well as on the translation — but we try to achieve it, and know of no exceptions.

There are two levels at which we consider correctness. Most of the RSL types and type constructors map directly into corresponding PVS types and type constructors. It is then a fairly routine task to map most of the RSL value functions, operators and constructors onto PVS ones, adding to the basic PVS theory when necessary.

But there is a deeper question of the adequacy of PVS to represent all RSL types and values. If, for example, the mapping is not injective then we will effectively equate different RSL values by mapping them to the same PVS ones, and the translation would not be sound: we would effectively create theorems that do not necessarily hold in PVS.

Concurrency and imperative constructs have no counterpart in PVS, so we exclude them from the translation. More problematic is the RSL logic, which permits potentially undefined and nondeterministic expressions. We cannot represent these by PVS expressions, so we have to exclude them. Some like the basic expressions **chaos** and **swap** can be excluded syntactically. The internal choice operator ⌈⌉ is also excluded syntactically. Nondeterminism in maps can be handled by either making sure that the translated RSL will generate suitable TCCs for the PVS, or by generating the required conditions as extra PVS `LEMMA`s. No proof in PVS can be considered sound unless the TCCs and all lemmas are proved. Generation of **swap** in applicative specifications can be checked by confidence conditions for the completeness of case expressions and possible matches in some let expressions.

There remain two issues which we cannot easily handle completely formally, where the RSL theory of the specification is effectively strengthened by the translated PVS theory: recursive functions and nondeterministic let expressions. We can summarize this by noting that in PVS the equality `e = e` holds for any expression `e`. In RSL this is not true if e is undefined or is nondeterministic, and this will occur if e is a non-terminating application of a recursive function, and may occur if e involves an implicit let expression (or, equivalently, the application of **hd** to a set or map). So users should be aware that these must be checked by them. Recursive functions should be terminating, and any use of nondeterminism should involve only *weak* nondeterminism: i.e. the value of any function using nondeterminism should not itself be nondeterministic.

A translation should therefore only be considered correct if:

1. the translation generates no errors or warnings
2. the PVS output type checks
3. all TCCs and confidence condition `LEMMA`s are proved
4. recursive functions terminate
5. functions involving nondeterminism are deterministic

We expect the second condition to hold, but the translator does not check, for example, that no identifier is used which clashes with a reserved word in PVS (see section 12.10), or that the "flattening" of object declarations causes no scope errors (see section 12.3.2).

12.3 Declarations

A declaration translates to one or more theory, type, constant, or function declarations as described below for the various kinds of declarations.

Note that PVS has a *define before use* rule. This means that the order of type and value declarations in PVS may differ from that in RSL.

12.3.1 Scheme declarations

Apart from the top level module (which is translated as if it were an object), schemes are only translated when they are instantiated as objects. So a scheme that is instantiated several times will therefore be translated several times.

There is an exception to this for non-parameterized schemes used in class scope expressions, as we see in section 12.7.29.

12.3.2 Object declarations

An object translates as its translated declarations in a `THEORY` of the same name as the object.

Theories in PVS cannot be nested. So nested object declarations are "flattened". For example, consider the following RSL scheme definition:

```
scheme S =
  class
    object A :
      class
        object B : class b_body end
        a_body
      end
    s_body
  end
```

This translates to:

```
B : THEORY
  BEGIN
      TheoryPart_B
  END B

A : THEORY
  BEGIN
      IMPORTING B
      TheoryPart_A
```

```
    END A

S : THEORY
    BEGIN
         IMPORTING A
         TheoryPart_S
    END S
```

This means that we do not accept object classes that reference entities in their context. For example a_body cannot refer to anything in s_body.

An object definition with a formal array parameter is not accepted: object arrays are normally only used in imperative or concurrent specifications.

12.3.3 Type declarations

Type declarations are translated according to their constituent definitions.

Mutually recursive type definitions are not accepted.

Sort definitions Sort definitions are translated into uninterpreted type definitions in PVS.

However PVS distinguishes between empty (defined with the reserved word `TYPE`) and nonempty types (defined with the reserved word `TYPE+`). This syntactic distinction does not exist in RSL and can be important in the transformation since the PVS type checker will generate an *existence* TCC for every function declared if it can not determine if the result type of the function is a nonempty type. This is so since PVS considers inconsistent the existence of a function with possibly empty result type. (In RSL such a function with parameters from non-empty types would be evidence of the non-emptiness of the result type.)

So we translate every RSL sort into a PVS nonempty type (`TYPE+`). But this extends somewhat the transformation since it is like adding in RSL:

axiom ∃ t: T • **true**

This is unlikely to cause any problems in extending the RSL theory since the declaration of a value of type T or a total function with non-empty domain type and result type T would also imply it. Since it avoids getting extra TCCs we adopt it.

Variant definitions Variant types in RSL translate directly into PVS `DATATYPE`s.

PVS `DATATYPE`s are very similar to RSL variant types, except that:

- PVS includes `recognizers`, boolean functions that return true if their argument is in the corresponding variant. These are added to the translation, given names formed by appending "?" to the constructor name.

- All components in PVS need `accessor`s corresponding to RSL's destructors. In RSL these are optional: the translator generates missing ones with names of the form `acc_`n`_`, where n is an integer used to make unique names, and warns the user that they have been generated.
- PVS does not include RSL's optional reconstructors, so when these are included they are added by defining them explicitly as functions in PVS.
- RSL allows wildcard constructors. These are not accepted by the translator.

For example, the following RSL variant type definition:

type
  V ==
    Vconst |
    Vint(**Int**) |
    Vrec(d1Vrec: **Int** $\xrightarrow{\sim}$ r1Vrec, d2Vrec: V)

generates the following PVS definitions:

```
V: DATATYPE
BEGIN
  Vconst: Vconst?
  Vint(acc_1_: int): Vint?
  Vrec(d1Vrec: int, d2Vrec: V): Vrec?
END V

r1Vrec(z_: int, x_: {x_: V | Vrec?(x_)}): V =
  LET x1_ = d1Vrec(x_), x2_ = d2Vrec(x_)
  IN Vrec(z_, x2_);
```

Short record definitions Records in RSL are translated as PVS `DATATYPE`s with single components.

Abbreviation definitions An abbreviation definition translates to a PVS type definition.

Union Definitions Union definitions are not accepted.

12.3.4 Value declarations

Typings Typings are translated to PVS constant declarations.

Explicit value definitions An explicit value definition translates to a PVS constant declaration.

Implicit value definitions An explicit value definition translates to a PVS constant declaration plus an axiom.

Explicit function definitions A non-recursive explicit function definition translates to a PVS function definition.

If the function has a precondition this is translated into a subtype for the last parameter. For example, the RSL definition

value
  diff : **Nat** × **Nat** $\xrightarrow{\sim}$ **Nat**
  diff(x, y) ≡ x − y
  pre x ≥ y

translates to

```
diff(x: nat, y: {y: nat | x >= y}): nat = x - y;
```

Access descriptors are not accepted. The kind of function arrow (→ or $\xrightarrow{\sim}$) does not matter.

It is not required that the number of parameters matches the number of components in the domain of the function's type expression. For example, the following are all accepted:

type
  U = **Int** × **Bool**
value
  f1: **Int** × **Bool** → **Bool**
  f1(x, y) ≡ ...,
  f2: (**Int** × **Bool**) → **Bool**
  f2(x, y) ≡ ...,
  f3: U → **Bool**
  f3(x, y) ≡ ...,
  f4: U × **Int** → **Bool**
  f4(x, y) ≡ ...,
  f5: (**Int** × **Bool**) × **Int** → **Bool**
  f5(x, y) ≡ ...,
  f6: (**Int** × **Bool**) × **Int** → **Bool**
  f6(x) ≡ ...,
  f7: **Int** × **Bool** → **Bool**
  f7(x) ≡ ...,
  f8: (**Int** × **Bool**) → **Bool**
  f8(x) ≡ ...,
  f9: U → **Bool**
  f9(x) ≡ ...,
  f10: U × **Int** → **Bool**
  f10((x, y), z) ≡ ...,

f11: (**Int** × **Bool**) × **Bool** → **Bool**
f11((x, y), z) ≡ ...

Recursive functions need `MEASURE`s in PVS to show that they terminate. Generating measures automatically is not possible in general, so the translation is to a constant declaration giving the function signature plus an axiom defining the function body. Preconditions are dealt with as for non-recursive functions.

Implicit function definitions An implicit function is translated as a constant declaration giving the function signature plus an axiom defining the postcondition. Preconditions are dealt with as for explicit function definitions.

12.3.5 Variable declarations

Variable declarations are not accepted.

12.3.6 Channel declarations

Channel declarations are not accepted.

12.3.7 Axiom declarations

Axioms are translated to PVS axioms.

12.4 Class expressions

A class expression translates to the definitions which the translation of the contents of the class expression results in.

12.4.1 Basic class expressions

A basic class expression translates as its declarations.

12.4.2 Extending class expression

An extending class expression translates as the two class expressions.

12.4.3 Hiding class expressions

Hiding is ignored (with a warning): hidden names are visible.

12.4.4 Renaming class expression

Renaming is ignored (with a warning).

12.4.5 With expression

With expressions are ignored: names are qualified as if they had been in RSL.

12.4.6 Scheme instantiations

A scheme instantiation translates as the unfolded scheme with substituted parameters.

12.5 Object expressions

An object expression which is a name is accepted as an actual scheme parameter or as a qualification.

A fitting object expression is accepted as an actual scheme parameter.

Neither element object expressions nor array object expressions are accepted.

12.6 Type expressions

12.6.1 Type literals

Except for **Unit** the RSL type literals are translated into the corresponding PVS types as shown in table 8.

RSL	PVS
Bool	`bool`
Int	`int`
Nat	`nat`
Real	`real`
Char	`char`
Text	`string`

Table 8: Type literals

The only possible problem is that in PVS `int` is a subtype of `real`, while the corresponding RSL types are different. This means we must be careful with division and exponentiation, to make sure they give integers with integer arguments.

12.6.2 Names

A type name translates to a type name.

12.6.3 Product type expressions

A product type expression translates to a PVS tuple type.

12.6.4 Set type expressions

Both finite and infinite set type expressions translate to the PVS type `set`. It would be possible to use `finite_set` for finite sets, but in practice this generates a TCC for every function returning such a value, which is tedious to prove. Only the `card` operator actually requires a set to be finite, so it is better to prove finiteness only when it is required.

12.6.5 List type expressions

A finite list type expression translates to the PVS type `list`.

Infinite lists are not accepted.

12.6.6 Map type expressions

Finite and infinite maps are translated into a PVS type `map` defined in the RSL prelude.

The translation of a map is as a function from the domain type to a `DATATYPE`:

```
Maprange[rng: TYPE]: DATATYPE
  BEGIN
    nil: nil?
    mk_rng(rng_part: rng): nonnil?
  END Maprange
```

The result type `nil` when a map is applied indicates the argument is not in the domain. The RSL map application expression m(d) translates to `rng_part(m(d))`, which will generate the appropriate TCC `nonnil?(m(d))` expressing that d is in the range of m.

The `map` type does not include nondeterministic maps: these are not accepted.

12.6.7 Function type expressions

A function type expression translates to PVS function type.

PVS functions are total. We partly deal with partiality in RSL functions: preconditions generate subtypes as described earlier for the translation of functions (section 12.3.4), but nondeterminism is not accepted.

Access descriptors are not accepted.

12.6.8 Subtype expressions

A subtype expression translates to a PVS subtype.

12.6.9 Bracketed type expressions

A bracketed type expression translates as a PVS tuple type with one member.

12.7 Value expressions

12.7.1 Value literals

The RSL value literals are translated into their PVS counterparts.

There are no real literals in PVS, so a real literal $i.d$ (where i is the integer part of the real number and d its decimal part) is translated into $i + d/10^{dn}$ where dn is the number of decimal digits.

12.7.2 Names

A value name translates as a name.

12.7.3 Pre names

Not accepted.

12.7.4 Basic expressions

Basic expressions (**chaos**, **skip**, **stop**, and **swap**) are not accepted.

12.7.5 Product expressions

A product expression translates to a PVS tuple expression.

12.7.6 Set expressions

Sets are modelled in the PVS prelude as functions that return `true` when applied to a member of the set, `false` otherwise.

All set expressions are accepted, as illustrated in table 9.

RSL	PVS
{}	`emptyset`
{x, y}	`add(x, add(y, empyset))`
{x .. y}	`LAMBDA (z : int): x <= z AND z <= y`
{ b | b : T • p }	`{ b : T | p }`
{ e | b : T • p }	`{ u : U | EXISTS (b : T) : p AND u = e}`

Table 9: Set expressions

The third example is a special case of a set comprehension. The more general case is the last, where `u` is a new identifier not free in `p` or `e`, and `U` is the type of `e`.

12.7.7 List expressions

All finite list expressions are accepted, as illustrated in table 10.

RSL	PVS
⟨⟩	`(::)`
⟨x, y⟩	`(:x, y:)`
⟨x .. y⟩	`ranged_list(x, y)`
⟨e | b **in** l • p⟩	`map(LAMBDA (b: {b: T | member(b, l) AND p}): e, filter(l, LAMBDA (b: {b: T | member(b, l)}): p))`

Table 10: List expressions

`ranged_list` is defined in the RSL prelude. `map`, `member`, and `filter` are defined in the PVS prelude.

12.7.8 Map expressions

All map expressions are accepted, but either RSL confidence conditions or PVS TCCs may be generated to check the map is deterministic. Examples are shown in table 11.

`emptymap`, `add_in_map`, `mk_rng`, and `RSL_inverse` are defined in the RSL prelude. The last is the same as the PVS prelude's `inverse` function, but with a subtype to generate a TCC that `LAMBDA (b : T):`

RSL	PVS
[ ]	`emptymap`
[x ↦ p, y ↦ q]	`add_in_map(x,p,add_in_map(y,q,emptymap))`
[b ↦ e | b : T • p]	`LAMBDA (b : T): IF p THEN mk_rng(e) ELSE nil ENDIF`

[e1 ↦ e2 | b : T • p]	`LAMBDA (u : U):`
	`  LET b = RSL_inverse(LAMBDA (b : T): e1)(u) IN`
	`  IF p THEN mk_rng(e2) ELSE nil ENDIF`

Table 11: Map expressions

`e1` is an injective function, a sufficient condition for the map to be deterministic. `u` is a new identifier not free in `p`, `e1`, or `e2`, and `U` is the type of `e1`.

A confidence condition that x≠y is generated for the second example, again a sufficient condition for the map to be deterministic.

12.7.9 Function expressions

Function expressions are accepted.

12.7.10 Application expressions

An application expression may be translated to a function call, a list application or a map application.

12.7.11 Quantified expressions

Quantified expressions are accepted.

12.7.12 Equivalence expressions

Equivalence expressions translate as equalities in PVS.

12.7.13 Post expressions

Post expressions translate as `LET` expressions, as shown in table 12.

RSL	PVS
e **as** b **post** p	`LET b = e IN p`

Table 12: **post** expressions

12.7.14 Disambiguation expressions

Disambiguation expressions are accepted.

12.7.15 Bracketed expressions

A bracketed expression translates to a bracketed expression.

12.7.16 Infix expressions

An infix expression translates to the corresponding PVS expression. Some infix operators in RSL translate to functions in PVS.

12.7.17 Prefix expressions

A prefix expression translates to the corresponding PVS expression.

The universal prefix expression □ e is translated as e, since □ e is equivalent to e for applicative expressions.

12.7.18 Initialise expressions

Not accepted.

12.7.19 Assignment expressions

Not accepted.

12.7.20 Input expressions

Not accepted.

12.7.21 Output expressions

Not accepted.

12.7.22 Local expressions

Not accepted.

12.7.23 Let expressions

Let expressions are accepted.

12.7.24 If expressions

If expressions are accepted.

12.7.25 Case expressions

Case expressions are accepted. They translate to PVS `IF` expressions since the case patterns in RSL are more general than those in PVS.

12.7.26 While expressions

Not accepted.

12.7.27 Until expressions

Not accepted.

12.7.28 For expressions

Not accepted.

12.7.29 Class scope expressions

A class scope expression translates to a PVS `THEORY`. So an RSL **theory** containing several class scope expressions as axioms will in general translate to a PVS file containing a number of theories. The PVS `THEORY` from the class scope expression **in** C $\vdash$ p contains the translation of the class expression C plus the translation of p as a `LEMMA`, i.e. something to be proved.

When, however, a number of class scope expressions have a class expression which is an instantiation of the same non-parameterized scheme, the scheme is translated as a separate PVS `THEORY` and the class scope expressions are translated as a single `THEORY` that imports the `THEORY` for the scheme and contains

all the resulting `LEMMA`s. This makes it possible to use earlier lemmas in proving later ones, since they share the same definitions from the imported `THEORY`.

12.7.30 Implementation relations and expressions

These are expanded into their conditions and these conditions translated.

12.8 Bindings and typings

Bindings and typings are accepted.

RSL allows nested product bindings, but PVS does not. The translation introduces new identifiers of the form `prod`n`_` for inner products, plus let expressions to define the original identifiers in their scopes. For example:

let (w,(x,(y,z))) = e **in** e1 **end**

translates to

```
LET (w,prod1_) = e IN LET (x,prod2_) = prod1_, (y,z) = prod2_ IN e1
```

12.9 Names

RSL names that are identifiers translate to PVS identifiers.

Qualified names translate as qualified names, except that as nested objects become non-nested PVS theories (see section 12.3.2) only the innermost qualifier is needed, e.g. A.B.f translates to `B.f`.

12.10 Identifiers

RSL allows primes on identifiers, but PVS does not. Primes translate to the string `rsL`. It is the user's responsibility to ensure that this translation does not cause name clashes in PVS.

It is also the user's responsibility to ensure that RSL identifiers do not clash with PVS reserved words. These are not case sensitive, unlike the user defined identifiers in PVS. So, for example, Lemma, lemma, and LEMMA would all clash with `LEMMA`.

13 UML to RSL translator

13.1 Introduction

This section presents the UML2RSL tool. This tool can translate UML class diagrams [8] to RSL. The translation is based on the work [9], where the authors explore the use of RSL to formalize UML class diagrams. This user guide provides full instructions on the use and installation of the UML2RSL tool on Unix, Linux and Windows platforms.

The UML2RSL tool was written by Ana Funes, as reported in [9].

13.2 General Description of UML2RSL

The overall pattern of use of UML2RSL is

- Draw a UML class diagram using a graphical tool
- Export the class diagram in XML
- Use UML2RSL to convert the XML file into a collection of RSL files

This is explained in more detail below.

UML2RSL has been developed in Java, which makes it a portable tool. As is shown in figure 3, it takes as input an XML file produced by a UML-based graphical tool, where all the information about a class diagram has been stored; it parses the XML file and, if the input is syntactically correct, translates the class diagram to an RSL specification based on the proposed semantics in [9].

The input XML file must be compliant with the XMI DTD version 1.2. There are several commercially available UML-based graphical tools having among their features the possibility of generating this kind of file. Free tools can be downloaded from http://www.magicdraw.com and from http://www.gentleware.com. The examples presented in this guide have been produced using the version 7.5 community edition of the MagicDraw tool.

To decide if the class diagram is syntactically correct, UML2RSL bases the analysis on a set of rules given in [9].

The resulting RSL specification is modular. It consists of several RSL files. One of them is named `S.rsl` and it corresponds to the top-level module. This module has the specification of the model represented by the whole class diagram. `S.rsl` uses a set of auxiliary modules. Each of them has the specification corresponding to one of the classes in the class diagram. These modules receive as name the corresponding class name in upper case, followed by `S_`. Each RSL module generated for a class use, in turn, a lower level module where the specification for one object of the corresponding class is given. They receive the same name than the class in upper case followed by `_`. Finally, each one of these lower level modules uses, in turn, a module named `TYPES.rsl` where all the abstract types present in the diagram are defined. There is a variation in the module structure of the specification when generalization relationships or templates classes are used as we will see in sections 13.6.2 and 13.6.3.

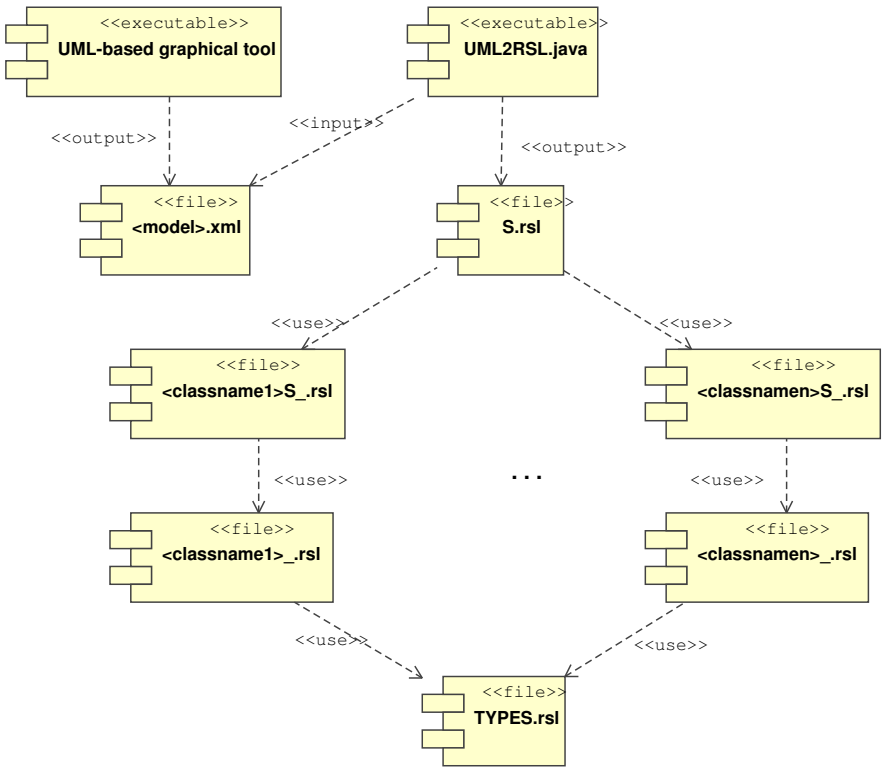

Figure 3: Component diagram

In figure 4 we give a class diagram in UML taken from [10] for a simple system: a Point of Sale System. This example serves to illustrate the resulting specification produced by the translator and to show the corresponding RSL dependency graph.

The produced specification consists of the top-level module `S` which uses `SALES_`, `SALELINEITEMS_`, `MANAGERS_`, `PRODUCTS_`, `PRODUCTCATALOGS_`, `PAYMENTS_`, `POSTS_`, `CASHIERS_`, `ITEMS_`, `CUSTOMERS_`, and `STORES_`. Each of them correspond to a class in the class diagram and they use respectively `SALE_`, `SALELINEITEM_`, `MANAGER_`, `PRODUCT_`, `PRODUCTCATALOG_`, `PAYMENT_`, `POST_`, `CASHIER_`, `ITEM_`, `CUSTOMER_`, and `STORE_` that have the specification for an object of the corresponding class. Finally, the last ones use the module `TYPES`. Figure 5 below shows the dependency module graph produced by the RSL tool for the specification obtained from the class diagram.

13.3 Distribution Files

The distribution files come in a single compressed file (`UML2RSL.tgz`). In this file you can find the following files:

```
Association.class
Attribute.class
```

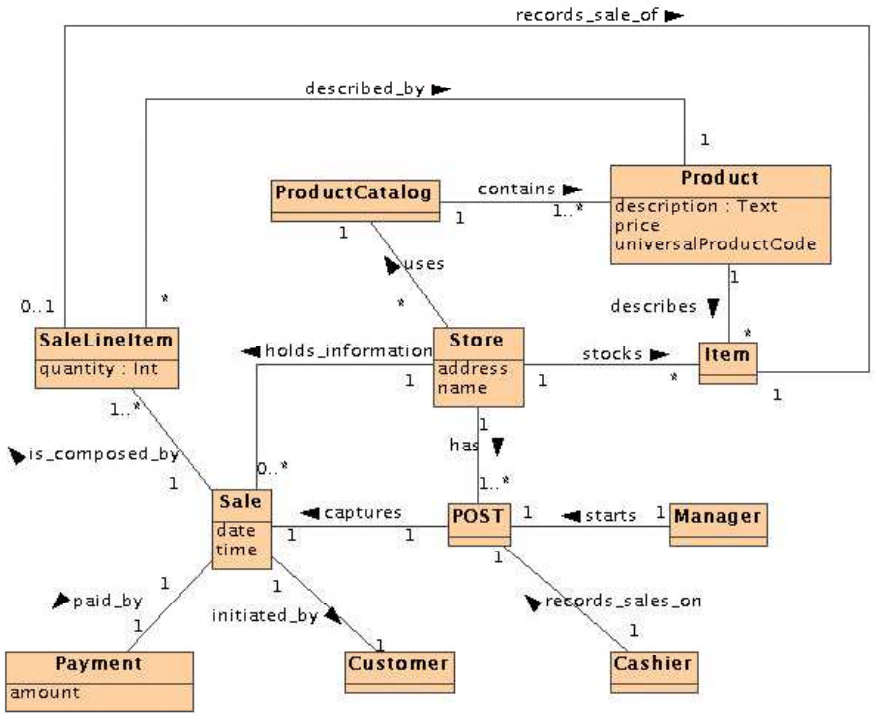

Figure 4: Class diagram for a Point of Sale System

```
Clase.class
ClassDiagram.class
Dependency.class
End.Class
EquivalentTypes.class
EquivalentTypesTable.class
FormalParameter.class
Generalization.class
Instantiation.class
Multiplicity.class
Operation.class
Pair.class
RecAlias.class
RSLKeywordTable.class
UML2RSL.class
UML2RSL.java
```

13.4 Installation

To install the UML2RSL tool you must follow the steps below:

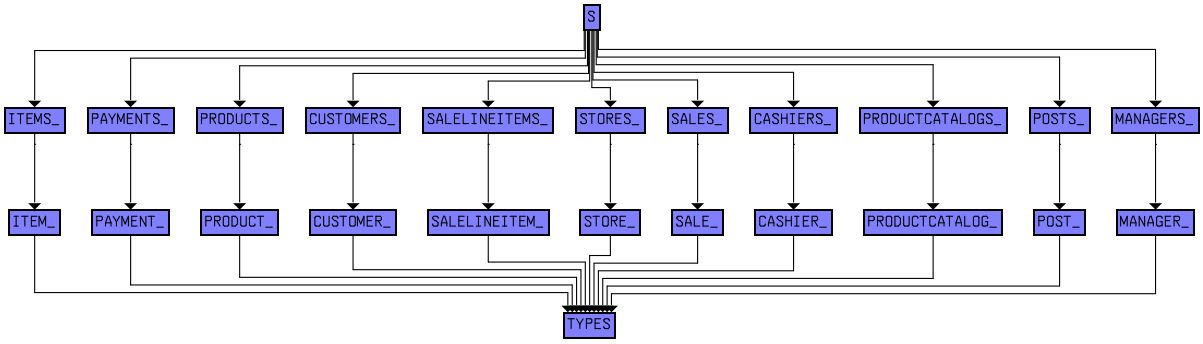

Figure 5: RSL module dependency graph

13.4.1 Installing the DOM Parser

UML2RSL uses a commonly used API (Application Program Interface) for XML processors: the **D**ocument **O**bject **M**odel (DOM) API [11] [12] to parse its input. In DOM, when an XML document is parsed it is represented as a tree. DOM provides a set of APIs to access and manipulate the nodes in the DOM tree.

You need to install the XML Parser for Java 3.2.1 release or a later compatible version.

An XML parser for Java can be downloaded from http://www.alphaworks.ibm.com/tech/xml4j, as a `.tar.gz` file for Unix or Linux, or a `.zip` file for Windows.

13.4.2 Installing the Java Virtual Machine

You must use a Java 1.4 or later compatible Java Virtual Machine to run the UML2RSL application.

Java Virtual Machines developed by Sun (JDK standard) for Unix, Linux and Windows can be downloaded from:

http://java.sun.com/j2se/1.4/

For information on other platforms see: http://java.sun.com/cgi-bin/java-ports.cgi.

13.4.3 Installing the Java byte code files

Decide on an installation directory, and extract from the compressed file `UML2RSL.tgz` all the `.class` files to that directory.

Build it yourself Alternatively, to build the Java byte code files from source, first just copy the Java source file `UML2RSL.java` (extracted from the `.tgz` file) to your installation directory.

In that directory then, for Windows, run the command

```
javac -classpath <xdir>\xml-apis.jar;<xdir>\xercesImpl.jar UML2RSL.java
```

or, for Unix or Linux, run the command

```
javac -classpath <xdir>/xml-apis.jar:<xdir>/xercesImpl.jar UML2RSL.java
```

where `<xdir>` is your installation directory.

In some implementations `xml-apis.jar` and `xercesImpl.jar` are replaced by a single file `xerces.jar`, and the above commands should be adapted accordingly.

If `javac` is not on your PATH, replace `javac` with its absolute name, which might in Windows be something like `C:\j2sdk1.4.1_03\bin\javac`

13.4.4 Creating the UML2RSL launcher

Follow the instructions below based on the operating system you are using.

Windows Create the file `UML2RSL.bat` in your installation directory saving the following command for the Java interpreter in it:

```
java -classpath <idir>;<xdir>\xml-apis.jar;<xdir>\xercesImpl.jar UML2RSL %1 <rdir>
```

where

- `<idir>` is your installation directory
- `<xdir>` is the directory where `xml-apis.jar` and `xercesImpl.jar`, parts of the XML parser for Java, are located. (In some implementations these two are replaced by a single file `xerces.jar`.)
- `<rdir>` is the relative path (from where the xml input file is stored) to a directory where you want the RSL files produced by the tool UML2RSL to be saved.

For example, if the XML parser was stored in `c:\xml4j`, your installation directory is `c:\UML2RSL`, and you want to store your RSL files in a sub-directory `RSL`, then `UML2RSL.bat` would contain

```
java -classpath c:\UML2RSL;c:\xml4j\xml-apis.jar;c:\xml4j\xercesImpl.jar UML2RSL %1 RSL
```

Make sure that your PATH variable includes the directory where the Java interpreter `java.exe` is installed. If you are not sure how to do this, you can replace `java` with its absolute name, which might be something like `c:\j2sdk1.4.1_03\bin\java`

You also need to put UML2RSL somewhere on your path. If you don't or can't do this you can still use it by using its absolute name: see section 13.5 below on using UML2RSL.

Unix and Linux Create the file UML2RSL in your installation directory with the following contents:

```
#!/bin/sh

java -classpath <idir>:<xdir>/xml-apis.jar:<xdir>/xercesImpl.jar UML2RSL $1 <rdir>
```

where

- `<idir>` is your installation directory
- `<xdir>` is the directory where `xml-apis.jar` and `xercesImpl.jar`, parts of the XML parser for Java, are located. (In some implementations these two are replaced by a single file `xerces.jar`.)
- `<rdir>` is the relative path (from where the xml input file is stored) to a directory where you want the RSL files produced by the tool UML2RSL to be saved.

Make UML2RSL executable, using for example the command

```
chmod u+x UML2RSL
```

Move it to somewhere on your path. If you don't or can't do this you can still use it by using its absolute name: see section 13.5 below on using UML2RSL.

13.5 Using UML2RSL

Suppose you have generated a file `model.xml` from your graphic UML tool, and you have set `<rdir>` to `RSL`. You can translate `model.xml` in a shell by

- cd to the directory where `model.xml` is stored
- UML2RSL `model.xml`

and the resulting `.rsl` files will be stored in the sub-directory `RSL`.

If `UML2RSL.bat` (Windows) or the shell script `UML2RSL` (Unix or Linux) is not on your path, you can use its absolute name, e.g. in Windows you can use the command

```
c:\UML2RSL\UML2RSL model.xml
```

assuming your installation directory is `c:\UML2RSL`

See section 13.4.4 for how to create `UML2RSL.bat` (Windows) or `UML2RSL` (Unix and Linux).

13.6 UML Class Diagram Supported Features

In the following sections we present by examples all those UML class diagrams features that can be translated to RSL with the UML2RSL tool.

13.6.1 Basic Class Features

In the simplest case, a class just consists of a name. It also can have a set of attributes, a set of operations and a multiplicity.

Each attribute must have a name, an optional type, a multiplicity, a scope ({classifier} or {instance}), and a changeability ({frozen}, {addOnly} or {changeable}). The default for the scope is {instance}, for the changeability is {changeable}, and for the multiplicity is `1..1`.

An operation must have a name and can have a list of formal parameters, where each parameter must have a name and an optional type. Furthermore, an operation has an optional result-type, a scope and it can be abstract.

The default for the multiplicity of a class is `*..*`.

For example, let us consider the simple class diagram in figure 6 which consists only of the class `Window` whose multiplicity is `*..*` (by default). It has 5 attributes and 2 operations:

`title`, which is a frozen attribute with multiplicity `1..*` and {instance} scope;

`default_size` and `max_size` that are changeable attributes with multiplicity `1..1` and {classifier} scope;

`size` and `visible` are attributes whose changeabilities are {changeable}, their multiplicities are `1..*`, and their scopes are {instance}.

`number_of_windows` is an operation whose scope is {classifier}; it has no parameters and its result type is `Int`.

`show` is an operation whose scope is {instance}; it has no parameters and its result type is not given.

The resulting specification will consist of four RSL modules: `S`, `WINDOWS_`, `WINDOW` and `TYPES`. The module `S` uses the module `WINDOWS_`, which uses `WINDOW_`, and `WINDOW_` uses in turn `TYPES`.

The module `WINDOW_` has the specification for an object of the class `Window`.

```
TYPES
object WINDOW_ :
  with TYPES in
  class
    type Window

    value
      size : Window → Size,
```

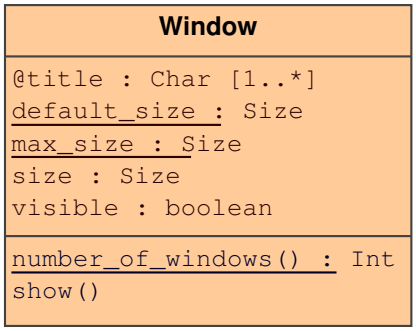

Figure 6: A simple class diagram

default_size : Window → Size,
max_size : Window → Size,
visible : Window → boolean,
title : Window → **Char-set**,

update_size : Size × Window $\xrightarrow{\sim}$ Window
update_size(at, o) **as** o′ **post** size(o′) = at
pre preupdate_size(at, o),

preupdate_size : Size × Window → **Bool**,

update_default_size : Size × Window $\xrightarrow{\sim}$ Window
update_default_size(at, o) **as** o′ **post**
  default_size(o′) = at
pre preupdate_default_size(at, o),

preupdate_default_size : Size × Window → **Bool**,

update_max_size : Size × Window $\xrightarrow{\sim}$ Window
update_max_size(at, o) **as** o′ **post** max_size(o′) = at
pre preupdate_max_size(at, o),

preupdate_max_size : Size × Window → **Bool**,

update_visible : boolean × Window $\xrightarrow{\sim}$ Window
update_visible(at, o) **as** o′ **post** visible(o′) = at
pre preupdate_visible(at, o),

preupdate_visible : boolean × Window → **Bool**,
show : Window → Window,

consistent : Window → **Bool**
consistent(o) ≡ **card** title(o) ≥ 1
end

The RSL abstract type `Window` is generated to specify the class sort. It denotes the set of all possible class instances or objects.

For each attribute, the tool generates an observer on the class sort. Each one of these observers takes an instance of the class and returns a value belonging to the corresponding attribute type.

The operations are specified as RSL functions, which have their domain in the Cartesian product of the class sort and the corresponding operation parameters. Their ranges correspond to the operation result types. When an operation in the class diagram does not return a value, it means that the operation performs some behaviour — based on the class structure and its parameters — which possibly changes the class structure. Therefore, in this situation, the RSL function will return the class sort. For example, the operation `show` in figure 6. However, when the scope of an operation is the class, since the operation acts on the class container, we do not generate the corresponding function on an instance of the class, but on the type defined for the class container. Therefore, the value used to specify the operation is generated in the module that holds the specification of the class. As we can see in the example,`number_of_windows` is placed in the module `WINDOWS_`.

In a class, besides the typical operation intended to return the value of an attribute, it is common to have an operation to modify the attribute, and since, frequently, the update of a given attribute occurs under a given pre-condition, RSL functions are generated for this purpose. Their pre-conditions should be completed by the user. If they are not necessary, then they may be removed. In the example above, the functions `update_size`, `preupdate_size`, `update_default_size`, `preupdate_default_size`, `update_max_size`, `preupdate_max_size`, `update_visible` and `preupdate_visible` were generated for this purpose. The corresponding update and preupdate functions for the attribute `title` were not generated because `title` is a frozen attribute.

The module `WINDOWS_` has the specification for the class `Window`.

```
WINDOW_
object WINDOWS_ :
  with TYPES in
  class
    type
      Window = WINDOW_.Window,
      Windows = Window_Id ~m~> Window

    value
      empty : Windows = [ ],

      add : Window_Id × Window × Windows ~~> Windows
      add(id, o, c) ≡ c † [ id ↦ o ] pre ∼ is_in(id, c),

      del : Window_Id × Windows ~~> Windows
      del(id, c) ≡ c \ {id} pre is_in(id, c),

      is_in : Window_Id × Windows → Bool
      is_in(id, c) ≡ id ∈ dom c,

      get : Window_Id × Windows ~~> Window
      get(id, c) ≡ c(id) pre is_in(id, c),
```

```
    update : Window_Id × Window × Windows ~→ Windows
    update(id, o, c) ≡ c † [ id ↦ o ] pre is_in(id, c),

    number_of_windows : Windows → Int,

    consistent : Windows → Bool
    consistent(c) ≡
      (∀ id : Window_Id •
         id ∈ dom c ⇒ WINDOW_.consistent(c(id))) ∧
      (∀ id1, id2 : Window_Id •
         (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
           WINDOW_.default_size(c(id1)) =
             WINDOW_.default_size(c(id2))) ∧
      (∀ id1, id2 : Window_Id •
         (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
           WINDOW_.max_size(c(id1)) =
             WINDOW_.max_size(c(id2)))
end
```

The type `Windows` describes the set of all the possible observable states in which the class `Window` can be, that is, a set of sets of `Window` objects or, in other words, all the possible sets of objects that can be observed at a given moment. We refer to it as the class container type. A characteristic of the objects is that each object is distinguishable from the other objects of the class, even if they have exactly the same property values. Consequently, the class container type is defined as a map from a set of object identifiers to the `Window` class sort.

For each class in the class diagram, new objects can be created and existing objects can be destroyed or modified. Therefore, some typical functions that operate on the set of instances of each class are generated by the translator. They are `empty`, `add`, `del`, `is_in`, `get`, and `update`.

When the scope of an operation is the class, since the operation acts on the class container, the RSL value used to specify the operation is placed in the module that holds the specification of the class. As we said before, `number_of_windows` is a class scoped operation, therefore its definition is generated in the module `WINDOWS_`.

The resulting RSL code produced for module `S` is given below.

```
WINDOWS_
object S :
  with TYPES in
  class
    type
      Sys :: windows : WINDOWS_.Windows ↔ replace_windows

    value
      update_windows : WINDOWS_.Windows × Sys ~→ Sys
      update_windows(c, s) ≡ replace_windows(c, s)
      pre preupdate_windows(c, s),
```

```
    preupdate_windows : WINDOWS_.Windows × Sys → Bool,
    number_of_windows_in_Windows :
      WINDOWS_.Windows × Sys → Int,
    show_in_Window : WINDOW_.Window × Sys → Sys,

    consistent : Sys → Bool
    consistent(s) ≡ WINDOWS_.consistent(windows(s))

  axiom
    ∀ s : Sys, c : WINDOWS_.Windows •
      update_windows(c, s) as s′ post
        consistent(s′) ∧ frozenAtts_in_Window(s′, s)
      pre consistent(s) ∧ preupdate_windows(c, s)

  value
    frozenAtts_in_Window : Sys × Sys → Bool
    frozenAtts_in_Window(s′, s) ≡
      (∀ id : Window_Id •
        (id ∈ dom windows(s) ∧
          id ∈ dom windows(s′)) ⇒
          WINDOW_.title(windows(s)(id)) =
            WINDOW_.title(windows(s′)(id)))
end
```

Each class in the class diagram has its corresponding field in the RSL record `Sys` to hold its class container. The type `Sys` describes all the possible instances of the system being modeled by the class diagram. In our example there is just one such field, whose name is `windows` that corresponds to the class container for the class `Window`.

The function `update_windows` is an extra operation defined to produce changes on the class container and `preupdate_windows` is the function signature for its pre-condition. The `preupdate_windows` definition will have to be completed later by the developer, according to his needs.

Since we are interested only in consistent systems, it is necessary to express consistency. This is achieved by a collection of axioms expressing the property that all the top-level state-changing functions maintain the system in a consistent state. In our example the only top-level state-changing function we have is `update_windows`, therefore there is an axiom for it. We capture all the model invariants in a boolean function named `consistent` which is defined as the conjunction among all the necessary predicates to express all the model constraints.

Class diagrams as well as their composing elements may have different constraints associated with them. As we want to check consistency on the whole system, that is, to check that all the constraints hold, we define a series of axioms on the top level module in order to check that the system is in a consistent state before and after any state change occurs. For this reason, not only the function `consistent` in the top-level module `S` is generated but a series of boolean functions are generated in the lower level modules as well. Inside each module that has been defined either to specify an object or a class, we define one of these boolean functions — that we also name `consistent`. They allow one to check the consistency of one object and the consistency of all the instances of the class, respectively. The last ones make use of the lower level ones, that is those defined for one instance of the class. The function `consistent` in the

top level module is generated to check the consistency of the whole class diagram, and it uses — in turn — all the lower level functions `consistent` defined for each one of the classes, guaranteeing in this way the consistency of the whole system.

In which of the several `consistent` functions the predicate for expressing a given property is put depends on what kind of restriction we are checking and for which element we want to check it. In our example, we have a predicate in the function `consistent` of the module `WINDOW_` to check the multiplicity of the attribute `title`.

```
consistent : Window → Bool
consistent(o) ≡ card title(o) ≥ 1
```

In the function `consistent` of module `WINDOWS_` we find the conjunction of three predicates.

```
consistent : Windows → Bool
consistent(c) ≡
  (∀ id : Window_Id •
      id ∈ dom c ⇒ WINDOW_.consistent(c(id))) ∧
  (∀ id1, id2 : Window_Id •
      (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
        WINDOW_.default_size(c(id1)) =
          WINDOW_.default_size(c(id2))) ∧
  (∀ id1, id2 : Window_Id •
      (id1 ∈ dom c ∧ id2 ∈ dom c) ⇒
        WINDOW_.max_size(c(id1)) =
          WINDOW_.max_size(c(id2)))
```

The first one checks the consistency of all the objects in the class container, making use of the lower level function `consistent` of module `WINDOW_`. The second and third ones were generated to express the fact that `default_size` and `max_size` are class scoped attributes. Since the multiplicity of the class is `*..*` no predicate must be generated. Other class multiplicities generate constraints (see section `Multiplicities` in [9]).

Finally, in the top-level module `S` of the example, we found a predicate to check the consistency of the whole system. In this example, it reduces to check only the consistency of the `Window` class container making use of the function `consistent` of lower level module `WINDOWS_`.

```
consistent : Sys → Bool
consistent(s) ≡ WINDOWS_.consistent(windows(s))

axiom
  ∀ s : Sys, c : WINDOWS_.Windows •
    update_windows(c, s) as s′ post
      consistent(s′) ∧ frozenAtts_in_Window(s′, s)
    pre consistent(s) ∧ preupdate_windows(c, s)

value
```

```
frozenAtts_in_Window : Sys × Sys → Bool
frozenAtts_in_Window(s′, s) ≡
  (∀ id : Window_Id •
      (id ∈ dom windows(s) ∧
       id ∈ dom windows(s′)) ⇒
         WINDOW_.title(windows(s)(id)) =
           WINDOW_.title(windows(s′)(id)))
```

Note that since it is necessary to check the property {frozen} for the window attribute `title`, a predicate is generated in the post-condition of the axiom corresponding to the class `Window`. It states for each object of class `Window` that the value of the attribute `title` remains unchanged after the system state has changed.

Finally, we have the module `TYPES` that constains the definitions of all the abstract types found in the model. The standard types `boolean`, `char` and `double` found in the class diagram are defined in `TYPES` as `Bool`, `Char` and `Real` RSL types respectively. The type `Window_Id` is used for the object identifiers in the class container of `Window`.

```
object TYPES :
  class

    type
      Size,
      boolean = Bool,
      Window_Id
  end
```

13.6.2 Relationship Features

In a class diagram, the classes can be related through different kinds of relationships. Basically, they are classified into three types: associations, generalizations and dependencies. Instantiations are viewed in UML as stereotyped dependencies, but since each stereotyped element has a particular meaning and we are interested specifically in instantiation, we separate it from general dependencies. UML2RSL accepts all of them. In the next four sub-sections we treat each one separately.

Association UML2RSL can translate only binary associations. However, in [9] we present a decomposition process for associations with arity greater than two. This process can be applied to any n-ary association ($n \geq 2$) before using the translator.

Each association end must have at least a name and can have several adornments: a given multiplicity, a given navigability, it can be composite or aggregate and — like attributes — it can have a changeabilty. The default for association multiplicity is `1..1`, for the navigability is `false`, and for the changeability is {changeable}.

An association is translated as two RSL functions among the involved classes (or one depending on its navigability ends, as we will see next). A function is generated for each of the involved class and it returns

the remaining related object.

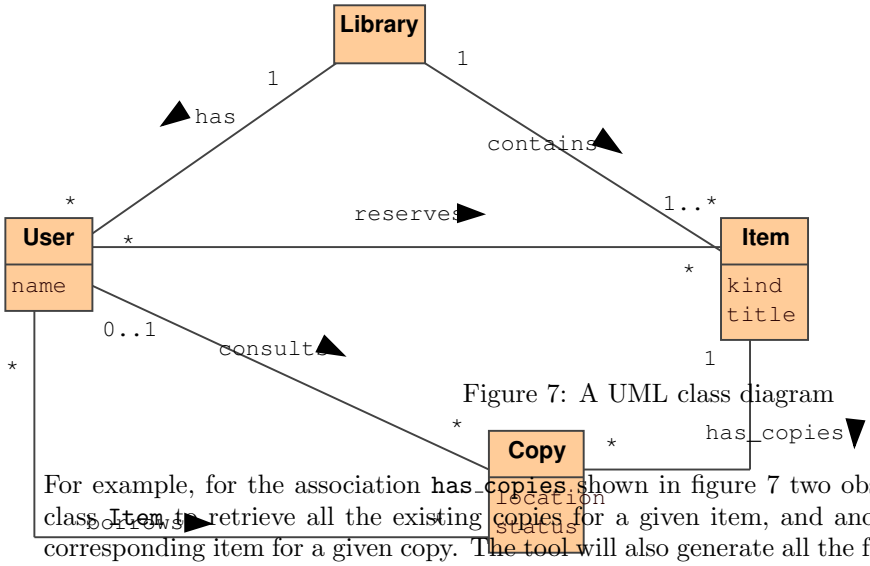

Figure 7: A UML class diagram

For example, for the association `has_copies` shown in figure 7 two observers will be generated: one in class `Item` to retrieve all the existing copies for a given item, and another in class `Copy` to obtain the corresponding item for a given copy. The tool will also generate all the functions for updating the objects retrieved by the association.

TYPES
object ITEM_ :
  with TYPES **in**
  class
    type Item

    value
      title : Item → title,
      . . .
      has_copies : Item → Copy_Id-**set**,

      update_has_copies : Copy_Id-**set** × Item $\xrightarrow{\sim}$ Item
      update_has_copies(a, o) **as** o′ **post** has_copies(o′) = a
      pre preupdate_has_copies(a, o),

      preupdate_has_copies : Copy_Id-**set** × Item → **Bool**,
      . . .

  end

```
TYPES
object COPY_ :
  with TYPES in
  class
    type Copy

    value
      status : Copy → status,
      ...
      has_copies : Copy → Item_Id,

      update_has_copies : Item_Id × Copy ~→ Copy
      update_has_copies(a, o) as o′ post has_copies(o′) = a
      pre preupdate_has_copies(a, o),

      preupdate_has_copies : Item_Id × Copy → Bool,
      ...

  end
```

Note that in the case of the class `Item`, the returned types for `has_copies` is a set of object identifiers of class `Copy`, while the `has_copies` of `Copy` only returns an object identifier of class `Item`. This structural distinction is because the multiplicities at the association ends are different. Association multiplicities are treated in detail in the section `Multiplicities` in [9].

When the association is navigable only in one direction, the tool will generate only one observer. For example, let us suppose that the association `has_copies` is navigable only from `Item` to `Copy`. In this case, only the function corresponding to the module `ITEM_` will be generated. Association Navigation is treated deeply in the section `Association Navigation` in [9].

Since the observers generated for the associations return object identifier(s), a predicate for checking that such object identifiers actually exist in the class containers is generated. When the association is bidirectional, besides checking existence as before, it is also necessary to check bi-navigation. Since this kind of checking involves references to the containers of the navigable classes, they must be placed in a module that has access to all the containers, that is the top level module. Therefore, in the function `consistent` of the top level module the following predicates will be generated for association `has_copies`.

```
(∀ id1 : Item_Id, id2 : Copy_Id •
    (id1 ∈ dom items(s) ∧
      id2 ∈ ITEM_.has_copies(items(s)(id1))) ⇒
       (id2 ∈ copys(s) ∧
         id1 = COPY_.has_copies(copys(s)(id2)))) ∧
(∀ id1 : Copy_Id, id2 : Item_Id •
    (id1 ∈ dom copys(s) ∧
      id2 = COPY_.has_copies(copys(s)(id1))) ⇒
       (id2 ∈ items(s) ∧
         id1 ∈ ITEM_.has_copies(items(s)(id2))))
```

Such predicates for checking existence and bi-navigation take slightly different forms depending on the

association end multiplicities. They are treated in detail in [9], section `Existence and Bi-navigation constraints`.

In this example, the changeability of the association ends is {changeable}, however the tool also translates {addonly} and {frozen} changeabilities. They are tretaed in the section `Attribute and Association End Properties` in [9].

Composition and Aggregation As [8] establishes, aggregation is purely conceptual, and does no more than distinguish the whole from the parts. It does not change the meaning of association. Therefore, aggregations are translated by the tool as general associations.

Composition is a form of aggregation with coincident lifetime of the parts with the whole, i.e. the parts may be created after the whole, and can also be explicitly removed before the whole. However, if the whole is destroyed they die with it. Because of that, the multiplicity of a composite end — unlike an aggregate end — must be always at most one (it cannot be shared by different instances of the owner class). This is checked by the tool.

In figure 8 an example of a composition between the classes `Company` and `Department` and a recursive composition on `Department` are shown.

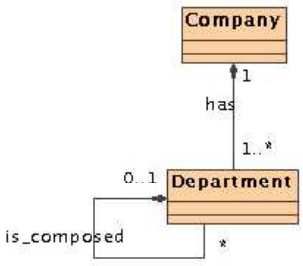

Figure 8: Composition

Structurally, a composition is equivalent to a general association. However, when we formalize the dynamic aspects of a composition, we make a distinction between a general association and a composition based on the property of coincident lifetimes of the whole and the parts. We express this property by means of a post-condition in the remove function of the whole. It assures that always when a whole is deleted all the parts that are currently associated are also deleted.

Given the composition `has` between the part `Department` and the whole `Company` shown in figure 8, the coincident lifetime property is generated by the tool as follows:

del_Company : Company_Id × Sys $\xrightarrow{\sim}$ Sys
del_Company(id, s) **as** s **post**
  (∃
    s′ : Sys, new_whole : COMPANYS_.Companys,
    new_parts : DEPARTMENTS_.Departments,
    parts : Department_Id-**set**
  •

```
        parts = COMPANY_.has(companys(s)(id)) ∧
        new_parts = departments(s) \ parts ∧
        s′ = update_departments(new_parts, s) ∧
        new_whole = COMPANYS_.del(id, companys(s′)) ∧
        s = update_companys(new_whole, s))
pre can_del_Company(id, s),

can_del_Company : Company_Id × Sys → Bool
can_del_Company(id, s) ≡
   COMPANYS_.is_in(id, companys(s)) ∧
   (∃
        s′ : Sys, new_whole : COMPANYS_.Companys,
        new_parts : DEPARTMENTS_.Departments,
        parts : Department_Id-set
    •
        parts = COMPANY_.has(companys(s)(id)) ∧
        new_parts = departments(s) \ parts ∧
        preupdate_departments(new_parts, s) ∧
        s′ = update_departments(new_parts, s) ∧
        new_whole = COMPANYS_.del(id, companys(s′)) ∧
        preupdate_companys(new_whole, s)),
```

Note that in UML, when the type of an attribute corresponds to a class sort, this attribute is, in effect, a composition relationship between the class and the class of the attribute, or, in other words, the attribute is a shorthand for composition. Consequently, this kind of attribute is translated in the same way as a composition relationship which is navigable only from the whole to the parts.

Generalization UML2RSL translates simple inheritance. Multiple inheritance is not supported.

Figure 9 shows an example of generalization between the classes `Person` and `User`.

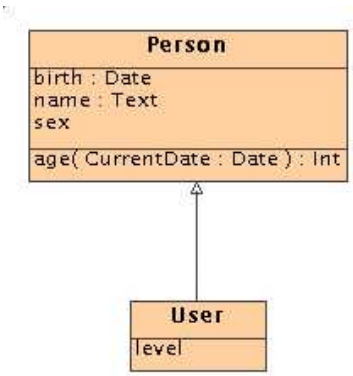

Figure 9: An example of generalization in UML

The type that denotes the set of all the instances of the subclass is generated as an RSL subtype of the class sort that corresponds to the superclass (`User` in the example). New attributes and operations added

to the subclass are translated as functions on the subclass type (`level` in the example).

```
PERSON_
object USER_ :
  with TYPES in
  class
    type
      Person = PERSON_.Person,
      User = {| o : Person • is_a_User(o) |}

    value
      level : User → level,

      update_level : level × User ~→ User
      update_level(at, o) as o′ post level(o′) = at
      pre preupdate_level(at, o),

      preupdate_level : level × User → Bool,
      is_a_User : Person → Bool,

      consistent : User → Bool
      consistent(o) ≡ PERSON_.consistent(o)
  end
```

The type corresponding to the type container of the subclass is generated as a subtype of the type used to specify the class container of the superclass, and the functions to operate on the class container are defined as usual.

```
    type
      Users =
        {| super : PERSONS.Persons •
            (∀ id : Person_Id •
                id ∈ dom super ⇒ USER.is_a_User(super(id))) |}
```

Figure 10 shows the dependency graph among the modules that have the specification for the superclass `Person` and the subclass `User`.

Dependency Because of the variety of meanings of Dependency, no specific semantics have been given in [9]. UML2RSL ignores all the dependencies found in a class diagram.

13.6.3 Advanced Class Features

UML2RSL supports also the translation of abstract, root, leaf and template classes. We treat each below.

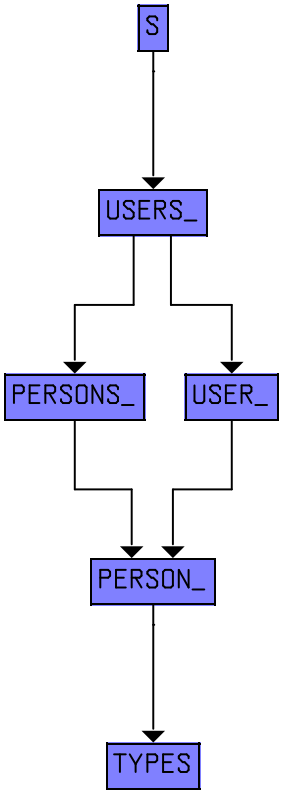

Figure 10: RSL module dependency graph for an example of generalization

Root Classes UML allows to constrain a class with the property {root}. This means that such a class may not have a superclass. This is checked by the tool during the syntactic analysis.

Leaf Classes Property {leaf} is used in UML to point out that a given class does not have children. In this case, the final structure of the RSL sort class can be fixed because no inheritance is possible from this class (see the semantics given for inheritance in [9]). Therefore, the type corresponding to the class is no more specified as a sort but as a more concrete RSL type: a record. For instance, if the class `Window` in figure 6 on page 97 was a leaf class, the RSL specification generated for an object of the class will be as follows:

```
TYPES
object WINDOW_ :
   with TYPES in
   class
      type
         Window ::
            size : Size ↔ replace_size
            default_size : Size ↔ replace_default_size
            max_size : Size ↔ replace_max_size
```

```
        visible : boolean ↔ replace_visible
        title : Char-set

    value
        update_size : Size × Window ~→ Window
        update_size(at, o) ≡ replace_size(at, o)
        pre preupdate_size(at, o),

        preupdate_size : Size × Window → Bool,
        ...
end
```

Note that the reconstructor for `title` is not generated because it is a frozen attribute.

Abstract Classes and Abstract Operations Whenever an abstract class is translated, since objects cannot be created, the functions that operate on the class container are not generated. Furthermore, a constraint to assure that no instances of the abstract class have been created is generated on the type `Sys`.

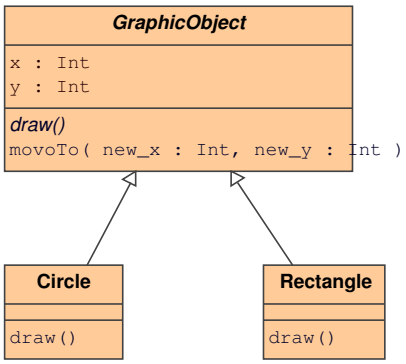

Figure 11: Example of abstract class

Let us consider the example in figure 11. The RSL code for this constraint, which is generated in the function `consistent` of module `S` is shown below.

```
consistent : Sys → Bool
consistent(s) ≡ ... ∧
    dom graphicobjects(s) =
        dom circles(s) ∪ dom rectangles(s)
```

It means that all the instances that can be in the class container corresponds to one of the concrete subclasses of the abstract class. This property can be expressed as the equality between the union of the abstract subclass containers and the abstract class container.

Abstract classes are classes that have at least one abstract operation. It means that the operation is incomplete and cannot be used, therefore an implementation must be given by a subclass. In RSL, the semantics of an abstract operation is given by hiding the operation name outside the module. So, outside the class module only references to the implementations given in the subclasses can occur. In the previous example, operation `draw` is abstract. To avoid the use of `draw` we hide its name.

```
TYPES
object GRAPHICOBJECT_ :
   with TYPES in
   hide draw in
   class
      type GraphicObject

      value
         x : GraphicObject → Int,
         y : GraphicObject → Int,
         ...
         draw : GraphicObject → GraphicObject,
         ...
   end
```

Template Classes UML2RSL deals with template classes too. It performs all the syntactic checks on them based in the syntax presented in [9] and produces the corresponding RSL code. An example of a template class and two different instantiations are shown in figure 12.

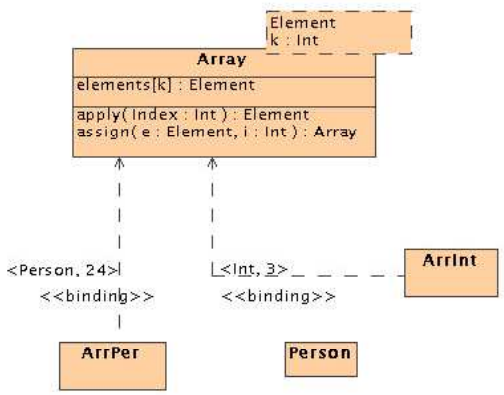

Figure 12: An example of template and instantiated classes

Figure 12 shows a template class for one array of k elements and the instantiation of one array of 3 integers and another of 24 persons.

A template class is translated to RSL in the same way as a concrete class but using a parameterized RSL scheme whose parameters corresponds to the class' parameters. In RSL when in a parameter expression only the name is given (as Element), it is assumed to be a type expression that resolves to a valid data

type (for example, Person or **Int**). Otherwise, it must resolve to a valid value expression (like 3 or 24 for **Int**).

```
TYPES
scheme ARRAY__(
    FPAR :
        with TYPES in
        class
            type Element

            value
                k : Int
        end) =
    with TYPES in
    class
        type Array, Elements = FPAR.Element-set

        value
            elements : Array → Elements,

            update_elements : Elements × Array ~→ Array
            update_elements(at, o) as o′ post elements(o′) = at
            pre preupdate_elements(at, o),

            preupdate_elements : Elements × Array → Bool,
            apply : Int × Array → FPAR.Element,
            assign : FPAR.Element × Int × Array → Array,

            consistent : Array → Bool
            consistent(o) ≡ card elements(o) = FPAR.k
    end
```

Since a template class cannot be used directly but only through its instantiations, the specification for its class container is not generated. However, for each instantiated class in the class diagram, UML2RSL generates the specification for its class container because they may have instances. These are generated as usual.

The semantics in RSL for the instantiation of a parameterized class is given by the instantiation of the corresponding parameterized scheme with its corresponding types and values. So, UML2RSL generates a new module to instantiate the array of 3 integers by instantiating the parameterized scheme ARRAY__ with Element equal to Int and k equal to 3 as follows:

```
ARRAY__
scheme ARRINT__ =
    with TYPES in  extend
      class

          object
              APAR_ArrInt:
```

```
            class
               type
                  Element = Int
               value
                  k: Int = 3
            end
   end
with extend ARRAY__(APAR_ArrInt) with class type ArrInt=Array end
```

and — as usual — UML2RSL generates an RSL object to represent the model corresponding to the specification of one object of class (`ArrInt` in this example).

```
ARRINT__
object ARRINT_: ARRINT__
```

As we can see, when we use template classes, the general structure of the resulting specification is sligthly different. In figure 13 we can see the RSL module dependency graph for the example of figure 12.

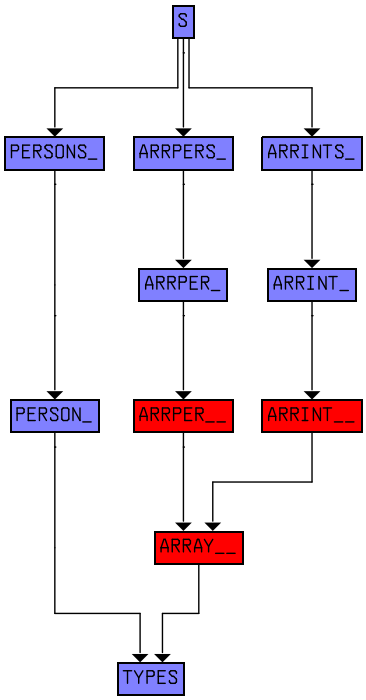

Figure 13: RSL module dependency graph for an example of template class

A more complete description of template class translation can be found in the section `Parameterized Classes` in [9].

Association Classes UML2RSL does not translate association classes. However, in [9] we propose a decomposition mechanism, which can be applied to all the association classes before the translation takes place.

13.6.4 Built-in types

Some standard UML data types are made equivalent to the corresponding RSL types (by means of abbreviation definitions in `TYPES.rsl`) and so can be freely used in UML classes: see table 13 below.

UML	RSL
boolean	Bool
char	Char
double	Real

Table 13: UML standard data types and equivalent RSL types

The UML data type `int` is not accepted by the translator because it clashes with the RSL built-in function **int**.

It is also possible to use the RSL built-in types **Unit**, **Bool**, **Int**, **Nat**, **Real**, **Char**, and **Text** in UML class diagrams. You will need to add them as new `Data Types` in the tool.

14 SAL Translator

14.1 Introduction

The SAL translator ws written by Juan Perna, as reported in [13]. This user guide was written by Ana Garis.

Rigorous Approach to Industrial Software Engineering (RAISE)[1] development method is carried out as a sequence of steps, starting from the specification of the system at a high level of abstraction and progressing by successively adding details towards a more concrete specification. To follow the RAISE development method, several tools are available such as code generators for several languages, test cases, and for model checking.

Model checking is a technique for verification of models, used for ensuring the correctness of hardware and software systems. Basically it consists of three steps: model specification, properties specification and verification (if model satisfies the specifications). The model is usually expressed as a transition system and properties are written as formal specifications, often using temporal logic formulas (CTL*, CTL or LTL).

Model checkers tool, such as SPIN, SMV and SAL, allow one to do the verification. For this first it is necessary to specify the model and the properties using the appropriate language (the language defined for each tool).

Regarding RAISE tools and in particular the tool for model checking (developed in 2006 by Juan Perna

[13]), it allows one to translate RSL (RAISE specification language) to SAL (see [14]). Later, verification can be done using the SAL model checker.

14.1.1 Why use the RSL-SAL translator

RSL-SAL translator enables the use of model checking for software components/systems verification. In particular, the tool provides model checking facilities for RSL.

Model checking is aligned with the RAISE development method, because it allows the verification of properties in early stages of the development process. Once verified, the RAISE development process warranties the preservation of the properties until the actual implementation of the system.

14.1.2 About the tool

The tool takes an RSL file and generates three different SAL translations. The difference between this three versions V1, V2 and V3 is that V1 does the verification under the assumption of confidence condition (CC) satisfaction, and V2 and V3 do not.

You must remember that "Confidence conditions are conditions that should probably be true if the module is not to be inconsistent, but that cannot in general be determined as true or false by an automatic tool" [15]. For example, a definition of a partial function without a precondition generates the CC "false".

Under assumptions in V1, all partial functions are considered as total and type well-formedness is taken for granted. But, in general, it is difficult to grant this, so it is important to use the automatic verification power of model checking to first verify the satisfaction of CC.

In order to check CC, V2 and V3 are generated. V2 (called "CC" version) allows to check CC. On the other hand, V3 (called "simple CC" version) allows to check CC but gives less diagnostic information.

When the tool translates an RSL file, it generates a file for each version. Also it generate other files, like `SAL_TYPES.sal`, `SAL_GLOBAL.sal`, `IT_AN.sal`, `NT_AN.sal` and `BT_AN.sal`. The last three contain the definitions of Integer type, Natural type and Boolean type respectively. More details about this can be found in the last section of the present report.

The following section shows which are the basic RSL constructs translatable to SAL. Section 14.3 shows how to write RSL transition systems and LTL assertions in order to use model checking technique, preceded by general overview about model checking. The specifications written following 14.3 will need to use the translatable basic RSL constructs described in section 14.3. Section 14.4 talks about V2 and V3: how RSL constructs are translated and how transition systems and LTL assertions are translated. Finally section 14.5 describes how to use the tool.

14.1.3 Known errors

- The translator does not deal properly with product types, product expressions (tuples) or product patterns, particularly in the CC version. Use records instead.

14.2 Translatable RSL constructs

Regarding V1, in the following subsections RSL constructs are listed, showing how they are translated to SAL. More implementation details about this, can be found in [13].

14.2.1 Declarations

Depending on the RSL kind of declaration, a declaration translates to one or more type, constant, function or variable declarations.

Scheme declarations The RSL *Scheme* constructor is translated to the SAL CONTEXT constructor.

Object declarations Object declarations are translated as instantiations of the SAL CONTEXT. If the applicative specification style is used, then object instantiations only introduce a name space in the current scheme.

Type declarations The *Boolean*, *Integer*, *Natural*, *Record*, *Variant* and *Collection* types are translated. But *Sort* and *Union* types are not accepted by the translator.

Sort Sort definitions are not accepted.

Boolean The RSL **Bool** type is translated to the BOOLEAN type in SAL.

Integer As integer is infinite by definition, it is necessary to impose a restriction over the possible values of the type. Then the translator uses a special integer type generated automatically during the translation as a subtype of the integer basic type of SAL (using the subrange structure in SAL). The subtyping used is, in RSL, of the form

type
Int_ = {| x : **Int** • x ∈ {DefaultIntLow .. DefaultIntHigh} |}

The tool generates a file (IT_AN.sal) where the Integer type definition is:

```
Int_:  TYPE = [SAL_GLOBAL!DefaultIntLow..SAL_GLOBAL!DefaultIntHigh];
```

SAL_GLOBAL.sal is the name of the SAL file (also generated by the tool) where `DefaultIntLow` and `DefaultIntHigh` are defined. By default, they are defined as -4 and 4 respectively, but this can be changed by the RSL specification, If a value definition of the form

value
IntHigh : T = n

is included in the specification being translated, where the type T is **Int** or any subtype of **Int**, and n is a literal, then n will be used as the value of `DefaultIntHigh` in SAL_GLOBAL.sal. A similar definition of IntLow will cause its value to be used as the value of `DefaultIntLow`.

It is necessary to make sure that DefaultIntHigh and DefaultIntLow are set to include any integer values generated when model checking the specification, since

- Integer arithmetic on range types in SAL is actually modulo arithmetic, to keep within within the bounds. Hence it only models RSL arithmetic properly when the bounds are wide enough.
- Exceeding the bounds for integers will cause errors to be generated in the CC version.

Natural Natural type is translated similarly to Integer type. The SAL subrange structure is used. The subtyping is, in RSL, of the form,

type
Nat_ = {| x : **Nat** • x ∈ {0 .. DefaultNatHigh} }|

The tool generates a file (NT_AN.sal) where the Natural type definition is:

```
Nat_:  TYPE = [0..SAL_GLOBAL!DefaultNatHigh];
```

DefaultNatHigh is by default set to 4, but this value can be changed in a similar fashion to DefaultIntHigh, by including in the specification a value definition of the form

value
NatHigh : T = n

where the type T is **Int** or any subtype of **Int**, and n is a literal.

Just as for integers, DefaultNatHigh needs to be large enough to include any natural values generated during model checking.

Variant type Variants are translated to the type declarator DATATYPE in SAL. For example, consider this example ([1], pg. 96)

type
List == empty | add(head : Elem ↔ replace_head, tail : List)

is translated to

```
List: TYPE = DATATYPE
  empty,
  add(head: Elem, tail: List)
END;

replace_head(z_: Elem, x_: List) : List =
  LET x1_ : Elem = head(x_) IN
  LET x2_ : List = tail(x_) IN
    add(z_ , x2_)
```

Note that reconstructor declarations (in this case "replace_head") are translated as explicit functions.

Record type RSL defines records as short variant definitions. Variants in SAL are defined with the type declarator "DATATYPE". Therefore record definition is translated to DATATYPE in SAL.

Union type Union type is not translatable into SAL.

Collection type (set, map, list) The strategy for translating sets and maps relies on an encoding based on total functions. The definition of the operations over sets and maps use LAMBDA functions.

- **Set type** The translation for sets uses an implementation based on a function from the domain of the set into a boolean value. For example, this type declaration:

  type
  Set1 = **Nat-set**

  will be translated into SAL as:

  ```
  Nat_set:  TYPE = [NT_AN!Nat_ -> BT_AN!Bool_];
  Set1:  TYPE = Nat_set;
  ```

- **Map type** Maps are also defined as functions but are in general not defined over all possible values in their domains. In this case, a map application over a value not in the map's domain will return the value *swap*.

  SAL does not provide partial function support, so partial constructions are not directly translatable. The translator modifies the map by creating a variant declaration. This declaration turns the map into a total function. For example, the following map definition

  type
  MyMap = T1 $\overrightarrow{m}$ T2

  is translated to

  ```
  T1_T2_map_range: TYPE = DATATYPE
    T1_T2_map_nil,
  ```

```
  T1_T2_map_range(T1_T2_map_val: T2)
END;

T1_T2_map: TYPE = [T1 -> T1_T2_map_Range];

MyMap: TYPE = T1_T2_map;

T1_set: TYPE = [T1 -> Bool_];
```

Non deterministic maps are not accepted in the translator. For example, the expression [x $\mapsto$ y | x, y : **Nat** • {x,y} $\subseteq$ {1, 2}] cannot be translated to SAL. Infinite maps (as in [n $\mapsto$ 2 $*$ n | n : **Nat** • is_a_prime(n)]) are not accepted for the translator.

- **List type** List definitions are not accepted.

Value declarations

Typings Value definitions of the form "identifier: type expression" (called *typings*) are not accepted by the translator.

Explicit value definitions Explicit value definitions are translated to a constant declarations in the model.

Implicit value definitions Implicit value definitions are not accepted by the translator.

Function definitions

- **Explicit function definitions** Explicit function definitions are translated to SAL explicit functions. The name of the function must be unique in the scheme. An example of a RSL declaration is

  value
  sum : **Int** $\times$ **Int** $\rightarrow$ **Int**
  sum(x,y) $\equiv$ x + y

  is translated to the SAL function

  ```
  sum(x:IT_AN!Int_ , y:IT_AN!Int_) :  IT_AN!Int_ = x + y;
  ```

  If the function name is overloaded in the specification, an error is reported during the translation. The same happens when operators are overloaded.

  SAL does not have predefined operators over sets and maps. The translator generates a file with macro declarations containing the names for the set and map operations. It will be expanded before model checking the specification. Table 14 and table 15 show function names for set operators and map operators.

Operator	Function name
x = y	=
x ≠ y	/=
x ⊃ y	strict_supset?(x, y)
x ⊂ y	strict_subset?(x, y)
x ⊇ y	supset?(x, y)
x ⊆ y	subset?(x, y)
x ∉ y	not_isin(x, y)
x ∩ y	intersection(x, y)
x \ y	difference(x,y)
x ∪ y	union(x, y)
x ∈ y	isin(x, y)

Table 14: Function names for set operators

Operator	Function name
x = y	=
x ≠ y	/=
x † y	override(x, y)
rng x	rng(x)
x / y	restriction_to(x, y)
x \ y	restriction_by(x, y)
dom x	dom(x)

Table 15: Function names for map operators

- **Partial function definitions**

  SAL does not support partial functions. The translator assumes that all preconditions of partial functions are verified. The tool, in V1, translates partial functions as SAL total functions. For example,

  value
  diff : **Nat** × **Nat** $\xrightarrow{\sim}$ **Nat**
  diff(x,y) ≡ x − y
  pre x ≥ y

  is translated to

```
diff(x:NT_!Nat_ , y:NT_!Nat_):  NT_!Nat_ = x - y;
```

- **Recursive function definitions** As in SAL, recursive function definitions are not accepted by the translator.
- **Implicit function definitions** Implicit function definitions are not accepted by the translator.

Variable declarations Variable declarations are not accepted by the translator.

Channel declarations Channel declarations are not accepted by the translator.

Axiom declarations Axiom declarations are not accepted by the translator.

Test case declarations Test case declarations are ignored by the translator.

14.2.2 Class expressions

The translation of a class expression results in the translation of its declarations and its statements.

Extending class expressions Extending class expressions are translated as a new class declarations. An extending class expression includes all the extended class declarations.

Hiding class expressions Hiding class expressions are ignored by the translator.

Renaming class expressions Renaming class expressions are ignored by the translator.

With expressions With expressions are ignored by the translator.

Scheme instantiations The translator only works with specifications written in an applicative style. So, the role of schemes is to provide type and value declarations.

14.2.3 Object expressions

An object expression which is an "object name" is accepted as a qualification. The following example, shows how object expressions are translated.

object N : NAMES
value
  f : N.Name → **Bool**
  f(n) ≡ N.to_bool(n)

is translated to

```
f(n: SAL\_TYPES!Name) : Bool_ = NAMES!to_bool(n);
```

Neither object array expressions nor fitting object expression are accepted by the translator.

14.2.4 Type expressions

RSL's basic type system uses the SAL type system for translating RSL type expressions.

Type names Type names translate to type names.

Product type expressions Product type expressions translate to SAL tuple declarations. For example,

MyProd = **Nat** × **Bool** × **Int**

is translated to

```
Prod :  TYPE = [NT_AN!Nat_ , BT_AN!Bool_ , IT_AN!Int_];
```

Set type expressions The translator generates a new set context for every set declaration/type expression found. Multiple set declarations of the same domain type are avoided. Set type expressions like,

MySet = (**Nat** × **Int**)-set

are rejected by the translator.

Only sets in which the domain is either a basic type or a defined type are accepted. So the expression in the previous example must be changed into:

MyData = **Nat** × **Int**,

MySet = MyData-**set**

List type expressions List type expressions are not accepted by the translator.

Map type expressions As in set type expressions, the translator generates a new finite map context for every map declaration/type expression found. Multiple map declarations of the same domain type are avoided.

Neither infinite maps nor non-deterministic map type expressions are allowed by the translator.

Function type expressions In general, the way to translate function type expressions is shown in 14.2.1. However, there are some exceptions:

- Curried functions are transformed into lambda functions.
- Function-type declared values are declared as function type and the value expression (a lambda abstraction expression) is assigned to it. For example,

  value
  sum: **Int** × **Int** → **Int** = λ (x,y) : **Int** × **Int** • x+y

  is translated to

  ```
  sum :[ [IT_AN!Int_, IT_AN!Int_] -> IT_AN!Int_ ] =
  LAMBDA (x:IT_AN!Int_, y:IT_AN!Int_):  x + y;
  ```

Subtype expressions Subtype expressions are translated to SAL subtype declarations. For example, the following subtype expression,

T = { | (x,y): **Int** × **Int** • x > y | }

is translated to

```
T: TYPE = {TypeId1_ :[IT_AN!Int_, IT_AN!Int_] | TypeId1_.1 > TypeId1_.2};
```

There are four special case for **Int** and **Nat**:

- T1 = { | x: **Int** • x ∈ {a..b} | }
  is translated to
  ```
  T1:  TYPE = [a..b]
  ```

- T2 = { | x: **Nat** • x ∈ {a..b} | }
  is translated to
  ```
  T2:  TYPE = [a..b]
  ```
- T3 = { | x: **Nat** • x <= a | }
  is translated to
  ```
  T3:  TYPE = [0..a];
  ```
- T4 = { | x: **Nat** • x < a | }
  is translated to
  ```
  T4:  TYPE = [0..a-1];
  ```

Bracketed type expressions A bracketed type expression translates as its constituent type expression.

14.2.5 Value expressions

Value expressions are translated of different forms.

Value literals The RSL value literals **Bool**, **Int** and **Nat** are translated but **Unit**, **Real**, **Char** and **Text** are not accepted by the translator.

Names A name translates as a name.

Pre names Pre names are not accepted by the translator.

Basic expression The RSL basic expression *skip* is ignored and *stop*, *chaos* and *swap* are not accepted in the translator.

Product expressions A product expression translates to a SAL tuple.

Set expressions All set expressions are accepted by the translator. Set expressions are modelled as total functions. They return true when they are applied to a member of the set, and false otherwise. The table 16, shows as set expressions are translated to SAL. The context name SET_OPS is used only for illustrative purposes.

List expressions List expressions are not accepted by the translator.

RSL	SAL
{}	`SET_OPS!emptySet`
{x, y}	`SET_OPS!add(x, SET_OPS!add(y, SET_OPS!emptySet))`
{x .. y}	`LAMBDA (z :IT_AN!Int_): x <= z AND z <= y`
{ b | b : T p(b) }	`LAMBDA (b :T): p(b)`
{ f(b) | b : T p(b) }	`LAMBDA (u :U): EXISTS (b :T) : p(b) AND f(b) = u`

Table 16: Set expressions

RSL	SAL
[ ]	`MAP_OPS!emptyMap`
[x ↦ p, y ↦ q]	`MAP_OPS!add(x,p,MAP_OPS!add(y,q,MAP_OPS!emptyMap))`
[b ↦ e | b : T . p]	`LAMBDA (b :T): IF p THEN m(e)=b ELSE nil ENDIF`

Table 17: Map expressions

Map expressions Generally map expressions are accepted, but they are not checked in order to verify if the resulting maps are deterministic. The table 17, shows as set expressions are translated to SAL. The context name MAP_OPS is used only for illustrative purposes.

Map expressions matching the pattern

[e1(x) ↦ e2(x) | x : T . p(x)]

(where e1 : T → U1, e2 : T → U2), are not accepted by the translator (because there is no way, in general, to generate the inverse function of e1).

List application expressions List application expressions are not accepted by the translator.

Function expressions Function expressions are translated to SAL's LAMBDA abstraction.

Application expressions An application expression is translated to a function call or a map application. List applications are not accepted by the translator.

Quantified expressions All quantified expressions are accepted. Regarding the translation mechanism, except ∃!, all quantifiers are directly supported by SAL. The translation of the ∃! expression is described following,

```
EXISTS (x:T) : p(x) AND ( FORALL (x1:T) : p(x1) => x = x1 )
```

Equivalence expressions Equivalence expressions are translated to SAL equalities.

RSL	SAL
$\Rightarrow$	`=>`
$\wedge$	`AND`
$\vee$	`OR`

Table 18: Axiom infix expressions

RSL	SAL
abs	`abs`
dom	`dom`
rng	`rng`

Table 19: Translation of built-in prefix operators

Post expressions Post expressions are not accepted by the translator.

Disambiguation expressions Disambiguation expressions are ignored (except set or map expressions which involve empty sets or empty maps).

Bracketed expressions Bracketed expressions are translated to SAL bracketed expressions.

Infix expressions Statement infix expressions are not accepted by the translator. On the other hand, the translation of the axiom infix expressions is straightforward. Table 18 shows how infix operators are translated.

Regarding value infix expressions, all expressions using infix operations over elements of any basic type are directly translated into their SAL counterparts. But infix expressions that use set or map operations are handled differently.

Equality/Inequality infix operations for set and map, remain as infix operations in the translated code (collections are implemented as functions).

The names of the operations showed in tables 14 and 15, are turned into prefix operations during the translation process (SAL does not support infix operator definitions for them).

Prefix expressions A prefix expression generally translates to the corresponding SAL expression. For example, the expression $\sim$ translates to `NOT`. The universal prefix expression, $\Box$, is not accepted by the translator.

The rest of the value prefix expressions translate to a function calls, using the function names described in Table 19. On the other hand, the prefix operators **int**, **real**, **card**, **len**, **inds**, **elems**, **hd** and **tl** are not accepted by the translator.

Comprehended expressions Comprehended expressions are not accepted by the translator.

Initialise expression Initialization expressions are not accepted by the translator.

Assignment expressions In the V1 of the translator, assignment expressions are only allowed when they describe transition systems (see 14.3.2). In this case, they are translated as SAL assignments.

Channel expressions expressions Channel expressions are not accepted by the translator.

Local expressions Local expressions are not accepted by the translator.

Let expressions SAL supports let expressions for simple bindings, so the translation mechanism for simple expressions is straightforward. For example,

let x = 0 **in** x + 1 **end**

is translated to

```
LET x :  IT_AN!Int = 0 IN x + 1;
```

Also it is possible translate more complex expressions, as you can see in the next example,

type
  Prod = **Int** × **Int**
value
  cons : **Int** × **Int** → Prod
  cons(a, b) ≡ **let** res = (a,b) **in** res **end**

is translated to

```
Prod: TYPE = [IT_AN!Int_, IT_AN!Int_];

cons(a :IT_AN!Int_, b :IT_AN!Int_) : SAL_TYPES!Prod =
  LET res :[IT_AN!Int_, IT_AN!Int_] = (a, b) IN res;
```

The type of the bounded name in the let definition must be explicitly stated in SALs model. On the other hand, the translation of binding involving products is more complex than previous cases. SAL imposes that there is only single binding in let expressions, preventing let expressions of the form "let (a,b) = P in" (where P is of product type). However, in SAL, product fields can be accessed by an index associated according the field position inside the product. The tool uses this feature to translate these expressions to SAL. For example,

value
  test : Prod → **Bool**
  test(p) ≡ **let** (a,b) = p **in** a > 1 **end**

is translated to

```
test(p :SAL_TYPES!Prod): BT_AN!Bool_ =
  LET LetId3_ : SAL_TYPES!Prod = p IN LetId3_.1 > 1;
```

If expressions The if expression translation is straightforward because SAL provides IF-THEN-ELSE and ELSIF constructions. For example, the following expression ([1], pg. 21)

if x > y **then** x - y **else** y - x **end**

is translated to

```
IF x > y THEN x - y ELSE y - x ENDIF
```

Case expressions A case expression is translated as a nested sequence of "if" expressions. For example, the expression

case x **of**
  1 → 10,
  2 → 20,
  _ → 0
end

is translated to

```
IF (x = 1) THEN 10
  ELSIF (x = 2) THEN 20
  ELSE 0
ENDIF
```

Iterative expressions Iterative expressions *while*, *until* and *for* are not accepted by the translator.

14.3 Writing transition systems and LTL assertions

Model checking is a formal method which consists basically in algorithmically verify if a model satisfies properties. The model is usually expressed as a transition system. On the other hand, properties are written as formal specifications, often using temporal logic formulas. Basically there are three kinds of temporal logic formulas: LTL , CTL and CTL^*.

In the context of RAISE development method, model checking is used early in development. Between the different kinds of RAISE specification styles, the applicative style is used. To use model checking specifying RSL code, it is necessary [16],

- To make a system finite. Model checking does an exhaustive check of the system. It needs a representation of the system as a finite set of all possible states. So, abstract types must be replaced by concrete types; also, types **Int** and **Nat** might be defined as small ranges.
- To define all the functions explicitly. Functions that can produce a new state of a system are specified for checking different aspects (that, perhaps provided these functions are applied according to some rules, or perhaps allowing them to be applied at any time, our system will evolve in particular ways).
- To add a transition system to express the rules controlling when functions can be applied (often just whenever their preconditions are true).
- To add definitions of the conditions to check. They are stated in the form of assertions in Linear Temporal Logic (LTL).

So, in order to use model checking techniques on RSL, it is necessary to represent transition systems and one of the kinds of temporal logic formulas (in particular, LTL assertions). Since none of these topics had a direct representation in RSL, it was extended adding features for writing transition systems and LTL assertions. Subsections 14.3.2 and 14.3.3 describe how write transition systems and LTL assertions respectively. Previously, a little background about model checking is presented in subsection 14.3.1

14.3.1 About Model Checking

Model checking is a very popular technique used for ensuring the correctness of hardware and software systems. Properties such as safety and liveness can be checked to assure correctness of systems, specially concurrent systems. It is a complete and automatic technique, but it only checks a model (not the real system).

Nowadays there are many model checkers, the main ones are SPIN, SMV and SAL. In particular, SAL is proposed for enabling model checking in RSL through the translator.

The model checking process consists of three steps: model specification, properties specifications and verification. The model specification is a mathematical model of the system, properties specification formally specify the desired behaviour of the system and finally the verification checks if model satisfies the specification.

Model specification The model is often expressed as a *transition system*, a directed graph consisting of nodes and edges (a computational tree). Each node has associated a set of atomic propositions. Nodes represent states of a system, edges represent possible transitions, and atomic propositions represent basic properties that hold at a point of execution.

Model checking is based on the calculation and the representation of all possible reachable states by the system, so the size of the computation increases exponentially with respect to the size of the problem. Several techniques have been developed to solve this issue. In particular, *symbolic model checking* and *abstraction* are two of the most used.

Specification of properties Desired behaviour of the system is specified with a formal language, a temporal logic. With temporal logic the static notion of truth is replaced by a *dynamic* one, in which the

formulas may change their truth values as the system evolves from state to state.

The Computation Tree Logic CTL^* is the most expressive temporal logic. It is used to describe properties of computation trees. CTL^* formulas are composed of path quantifiers and temporal operators.

- *Path quantifiers* are used to describe the branching structure in the computational tree. From a particular state, they allow one to specify that all of the paths or some of the paths starting at that state have some property.
  - A: "for all computational paths"
  - E: "for some computational path"
- *Temporal operators* describe properties of a path through the tree.
  - X: "next time"
  - F: "eventually" or "in the future"
  - G: "always" or "globally"
  - U: "until"
  - R: "release"

There are two types of formulas in CTL^*: *state formulas* and *path formulas*. State formulas are true in a specific state and path formulas are true along a specific path.

These formulas allow one to specify different requirements of systems, properties such as, reachability, safety, liveness and fairness. In particular, safety and liveness refer to,

- *Safety:* It is not possible to reach a particular dangerous state. Formula has the form: "$AG\ not\ s$", where s is a dangerous state.
- *Liveness:* For example, all requirements will have an acknowledgement. Formula has the form: "$AG[Req \rightarrow AF\ Ack]$"

There are two sublogics of CTL^*: *Branching-time logic* and *linear-time*. In branching-time temporal logic temporal, operators quantify over the paths that are possible from a given state. In linear-time temporal logic, operators are provided for describing events along a single computation path.

$ComputationTreeLogic(CTL)$ is a branching-time logic and it is a restricted subset of CTL^* in which each of the temporal operators X, F, G, U and R must be immediately preceded by a path quantifier.

$LinearTemporalLogic(LTL)$, on the other hand, is a linear time logic and consists of formulas that have the form (Af), where f is a path formula in which the only state subformulas permitted are atomic propositions.

14.3.2 Writing transition systems in RSL

To describe transition system inside RAISE's specification code, it is necessary to write following the grammar described below (an example about how to write a transition system can be found in subsection 14.3.4).

Transition_system_decl ::= “transition_system” {Base_module}$^{+}_{,}$
Base_module ::= “[“id“]” [“in” variable_decl] “local” variable_decl “in” Transition_decls “end”
Transition_decls ::= {Transition_decl}$^{+}_{[=]}$
Transition_decl ::= Single_guarded_command | Multiple_guarded_command
Single_guarded_command ::= [”[“id”]”] *logical*−value_expr “ ==>” Update_exprs |
[”[“ id “]”] “ else” “==>” Update_exprs
Multiple_guarded_command ::= “([=]” variable_decl “:-” Single_guarded_command “)”
Update_exprs ::= {id “:=” value_expr}$^{*}_{,}$

All variables declared as “in” are considered inputs to the transition system. The value for these variables will be initialized by the model checker. On the other hand, all variables declared as “local” represent the actual state of the transition system. If these variables are not initialized then the model checker will initialize them to any value. So the tool imposes the existence of an initial value for every local variable.

14.3.3 Writing LTL assertions in RSL

To describe LTL assertions inside RAISE’s specification code, it is necessary write following the grammar described below (an example about how to write LTL assertions can be found in subsection 14.3.4).

LTL_Property_decl ::= “ltl_assertion” {LTL_assertion}$^{+}_{,}$

LTL_assertion ::= “[“id”]” id “|−” LTL_expr

LTL_expr ::= *logical*_value_expr

The LTL temporal operators “G”, “F”, and “X” are allowed in LTL_exprs as function symbols.

14.3.4 An example

In some kind of the systems, such as a lift, it is important check certain properties, like safety and reliability. After modelling the system, it is possible to check these properties, using the model checking technique. Details of this problem, can be found in [16]. As was mentioned before, to use the model checking technique in the applicative RAISE style, it is necessary: to make a system finite, to define all the functions explicitly, to add a transition system, and to specify the LTL assertions.

First the example will be shown, a finite system will be done and all functions will be explicitly defined. Later the transition system and LTL assertions will be specified.

The problem Suppose that it is necessary to provide the control software for a lift in a building of 3 floors. The lift has to control different hardware components: cage, door, button, indicator, floor and motor.

Assumptions

- Hardware failures and need for maintenance are not considered.
- The time taken for the lift to move or the doors to open or close are not considered.
- It is assumed that lift doors (if any) and floor doors are operated indistinguishably (there are no differences between lift doors and floor doors).
- The management of the hardware components indicators and motors is ignored.
- Only the management of the hardware components (cages, doors and buttons) is considered.

Description *Cage* is single doored and it will presumably change its position, direction and speed. *Doors* are open or closed or perhaps in intermediate positions. *Buttons* are pressed (and lit) or cleared (and unlit).

The lift must serve a number of floors numbered consecutively. In each floor there are doors which must only be open when the lift is stationary at the floor. In each floor there are buttons. Except the top floor, there is a button to request the lift to stop there and then go up. Except the bottom floor, there is a a button to request the lift to stop there and then go down. Inside the lift also there is a button for each floor to request the lift to go to the floor.

RSL specification A lift is an example of an asynchronous system. This is, there are external stimuli that may arrive at any time (for example, buttons may be pressed at any time).

Following the RAISE development style, first it is necessary to consider the objects of the system and whether they will have dynamic state. In this case, the cage, the doors and the buttons will have dynamic state and hence be modelled as RSL objects.

The system must be specified and for this, it is necessary to formulate the type module for the system. This last one is called TYPES and later defined, it will instantiate as the global object "T".

```
scheme
  TYPES =
    class
      value
        min_floor : Nat = 0, max_floor : Nat = 2

      type
        Floor = {| n : Nat • n ∈ {min_floor .. max_floor} |},
        Door_state == open | shut,
        Button == bup0 | bup1 | bdown1 | bdown2 | blift0 | blift1 | blift2,
        Button_state == lit | clear,
        Direction == up | down,
        Movement == halted | moving,
        Requirement :: here : Bool  after : Bool  before : Bool

      value
        next_floor : Direction × Floor ~→ Floor
        next_floor(d, f) ≡
```

```
            if d = up then f + 1 else f − 1 end
            pre is_next_floor(d, f),

        is_next_floor : Direction × Floor → Bool
        is_next_floor(d, f) ≡
            if d = up then f < max_floor else f > min_floor end,

        invert : Direction → Direction
        invert(d) ≡ if d = up then down else up end
    end
```

Note that the type "Floor" has been defined as a subtype of Int. As it is supposed that there are three floors, it is possible to be explicit about what buttons there will be: two up buttons for floors 0 and 1), two down buttons for floors 1 and 2, and a lift button (inside the lift cage) for each floor.

The type "Requirement" is used to control the cage. It calculates, according to the button states, the current floor, and the current direction whether the cage is required to stop here, after or before.

Finally, note that the function "next_floor" indicates that f+1 is directly above floor f, and f−1 directly below it.

After defining the TYPE module and the object T, it is necessary to define modules for the cage, the buttons and the doors and from them to construct the lift system type. In particular, the definition of BUTTONS module and the CAGE module are shown.

```
scheme BUTTONS = hide required_here, required_beyond in
  class
    type
      Buttons ::
        up0 : T.Button_state ↔ re_up0
        up1 : T.Button_state ↔ re_up1
        down1 : T.Button_state ↔ re_down1
        down2 : T.Button_state ↔ re_down2
        lift0 : T.Button_state ↔ re_lift0
        lift1 : T.Button_state ↔ re_lift1
        lift2 : T.Button_state ↔ re_lift2

    value
      clear : T.Floor × Buttons →Buttons
      clear(f, bs) ≡
        case f of
          0 → re_up0(T.clear, re_lift0(T.clear, bs)),
          1 → re_down1(T.clear, re_up1(T.clear, re_lift1(T.clear, bs))),
          2 → re_down2(T.clear, re_lift2(T.clear, bs))
        end,

      press : T.Button × Buttons → Buttons
      press(b, bs) ≡
        case b of
          T.bup0 → re_up0(T.lit, bs),
```

```
        T.bup1 → re_up1(T.lit, bs),
        T.bdown1 →re_down1(T.lit, bs),
        T.bdown2 → re_down2(T.lit, bs),
        T.blift0 → re_lift0(T.lit, bs),
        T.blift1 → re_lift1(T.lit, bs),
        T.blift2 → re_lift2(T.lit, bs)
    end,

is_clear : T.Button × Buttons → Bool
is_clear(b, bs) ≡
    case b of
        T.bup0 → up0(bs) = T.clear,
        T.bup1 → up1(bs) = T.clear,
        T.bdown1 → down1(bs) = T.clear,
        T.bdown2 → down2(bs) = T.clear,
        T.blift0 → lift0(bs) = T.clear,
        T.blift1 → lift1(bs) = T.clear,
        T.blift2 → lift2(bs) = T.clear
    end,

check : T.Direction × T.Floor × Buttons → T.Requirement
check(d, f, bs) ≡
    T.mk_Requirement(
        required_here(d, f, bs),
        required_beyond(d, f, bs),
        required_beyond(T.invert(d), f, bs)),

required_here : T.Direction × T.Floor × Buttons → Bool
required_here(d, f, bs) ≡
    case f of
        0 → lift0(bs) = T.lit ∨ up0(bs) = T.lit,
        1 →
            lift1(bs) = T.lit ∨
                case d of
                    T.up →
                        up1(bs) = T.lit ∨
                            down1(bs) = T.lit ∧
                            lift2(bs) = T.clear ∧
                            down2(bs) = T.clear,
                    T.down →
                        down1(bs) = T.lit ∨
                            up1(bs) = T.lit ∧
                            lift0(bs) = T.clear ∧
                            up0(bs) = T.clear
                end,
        2 → lift2(bs) = T.lit ∨ down2(bs) = T.lit
    end,

required_beyond : T.Direction × T.Floor × Buttons → Bool
required_beyond(d, f, bs) ≡
    let f′ = T.next_floor(d, f) in
```

```
            required_here(d, f′, bs) ∨
            T.is_next_floor(d,f′) ∧
            let f′′ = T.next_floor(d, f′) in
              required_here(d, f′′, bs)
            end
        end
end
```

In the definition type of the BUTTONS module, it is possible to see that there are two "up" buttons on floors 0 and 1, and two "down" buttons on floors 1 and 2. There is a lift button inside the lift cage for each of the three floors.

```
scheme CAGE =
  class
    type
      Cage ::
        direction : T.Direction
        movement : T.Movement
        floor : T.Floor

    value
      /∗ generators ∗/
      move : T.Direction × Cage ~→ Cage
      move(d′, m) ≡
        mk_Cage(d′, T.moving, T.next_floor(d′, floor(m)))
      pre T.is_next_floor(d′, floor(m)),

      halt : Cage → Cage
      halt(m) ≡ mk_Cage(direction(m), T.halted, floor(m))
  end
```

After defining the CAGE, DOORS and BUTTONS modules, it is possible to construct the lift system type,

```
scheme LIFT =
  class
    object C : CAGE, DS : DOORS, BS : BUTTONS

    type
      Lift ::
        cage : C.Cage
        doors : DS.Doors
        buttons : BS.Buttons
```

Also, it is necessary to define functions to operate the lift system, such as "move", "halt", "check_buttons", "is_clear", and "press". The function "move" should specify that the lift is moved from a floor to the next floor in the given direction. The function "halt" should specify that the cage is halted, doors at the

current floor are opened and buttons for the current floor are cleared. The functions "check_buttons", "is_clear", and "press" should specify the access to the corresponding functions of BUTTONS. Finally the function "next" should calculate what to do next in any state, according to the current requirement. The specification of these functions is shown below.

```
value
  move : T.Direction × T.Movement × Lift ~→ Lift
  move(d, m, l) ≡
    mk_Lift(
      C.move(d, cage(l)),
      if m = T.halted
      then DS.close(C.floor(cage(l)), doors(l))
      else doors(l)
      end, buttons(l))
  pre T.is_next_floor(d, C.floor(cage(l))),

  halt : Lift → Lift
  halt(l) ≡
    mk_Lift(
      C.halt(cage(l)),
      DS.open(C.floor(cage(l)), doors(l)),
      BS.clear(C.floor(cage(l)), buttons(l))),

  check_buttons : Lift → T.Requirement
  check_buttons(l) ≡
    BS.check(
      C.direction(cage(l)), C.floor(cage(l)),
      buttons(l)),

  is_clear : T.Button × Lift → Bool
  is_clear(b, l) ≡ BS.is_clear(b, buttons(l)),

  press : T.Button × Lift → Lift
  press(b, l) ≡
    mk_Lift(cage(l), doors(l), BS.press(b, buttons(l))),

  next : Lift ~→ Lift
  next(l) ≡
    let
      c = cage(l),
      ds = doors(l),
      bs = buttons(l),
      r = check_buttons(l),
      d = C.direction(c)
    in
      case C.movement(c) of
        T.halted →
          case r of
            T.mk_Requirement(_, true, _) →
              move(d, T.halted, l),
```

```
                T.mk_Requirement(_, _, true) →
                    move(T.invert(d), T.halted, l),
                _ → l
            end,
        T.moving →
            case r of
                T.mk_Requirement(true, _, _) → halt(l),
                T.mk_Requirement(_, false, false) → halt(l),
                T.mk_Requirement(_, true, _) →
                    move(d, T.moving, l),
                T.mk_Requirement(_, _, true) →
                    move(T.invert(d), T.moving, l)
            end
    end
  end
pre can_next(l),

can_next : Lift → Bool
can_next(l) ≡
  let c = cage(l), r = check_buttons(l) in
    (T.after(r) ⇒
        T.is_next_floor(C.direction(c), C.floor(c))) ∧
    (T.before(r) ⇒
        T.is_next_floor(
            T.invert(C.direction(c)), C.floor(c)))
  end
```

The algorithm of the function "next" uses the current requirement. A requirement has three Boolean components: *here*, *after*, and *before*.

If the lift is halted then

- if *after* is true then move off in the current direction
- else, if *before* is true then move off in the opposite direction
- else no change

If the lift is moving then

- if *here* is true then halt
- else, if *after* and *before* are both false then halt
- else, if *after* is true then keep moving in the same direction
- else, if *before* is true, move in the opposite direction

As the function "move" in CAGE is partial, it is necessary to define the precondition "can_next". It ensures that the requirement only will move to another floor when such a floor exists.

Transition System The system has already been made finite and functions have already defined explicitly, so now the transition system will be specified. Such as was mentioned in subsection 14.3.2, to specify a transition system, it is necessary to decide about variables.

In this case, there is a single state variable Lift and it is possible to use just that. The initial state is chosen as: the lift halted with the doors open at floor 0, with all buttons clear.

Also, it is necessary to decide what the guarded commands for the transitions are. For this, the use of "next" with its precondition as guard is chosen. On the other hand, "press" is chosen too. A guard is needed for press: if this guard is not declared, the checking will fail when the lift must make progress, because the transition system will allow repeatable press transitions with no next transitions. So, "press" only is allowed when the button involved is clear.

```
transition_system
  [ L ]
  local
    lift : Lift :=
      mk_Lift(
        C.mk_Cage(T.up, T.halted, 0),
        DS.mk_Doors(T.open, T.shut, T.shut),
        BS.mk_Buttons(
            T.clear, T.clear, T.clear, T.clear, T.clear,
            T.clear, T.clear))
  in
    ([] b : T.Button •
      [ press ]
      is_clear(b, lift) ⟶ lift′ = press(b, lift))
    []
    [ next ]
    can_next(lift) ⟶ lift′ = next(lift)
  end
```

LTL assertions Now it is necessary to specify LTL assertions in order to verify properties that the lift must to have. An important property to verify is safety. To specify this property, suppose that a function "safe" in the scheme LIFT is defined as following,

```
safe : Lift → Bool
safe(l) ≡
  let c = cage(l), ds = doors(l) in
    (∀ f : T.Floor •
      (DS.door_state(f, ds) = T.open) =
        (C.movement(c) = T.halted ∧ C.floor(c) = f))
  end
```

The LTL assertion for safety is,

ltl_assertion

[safe] L ⊢ G(safe(lift)),
[req_safe] L ⊢ G(can_next(lift))

A LTL assertion will be useful to check that the lift eventually halts somewhere. This is written as,

ltl_assertion
[eventually_halts] L ⊢ G(F(C.movement(cage(lift)) = T.halted))

Another LTL assertion is defined to check that the lift is not permanently stationary on some floor. The specification intentionally asserts something we expect to be false, in this case that the lift never reaches floor 2,

ltl_assertion
[moves] L ⊢ G(C.floor(cage(lift)) < 2)

The assertion is invalid, so the model checker will generate a counter-example, such as the button lift2 is pressed and the lift moves from floor 0 to floor 1 and then continues to floor 2. Now it is possible to be sure that the lift is actually capable of useful behaviour.

Also it is important to specify liveness properties; for example, to specify for all floors that if a floor is requested from the lift then it must be eventually reached. This property for floor 0 is as follows,

ltl_assertion
[arrives0]
L ⊢
G(
BS.up0(buttons(lift)) = T.lit ∨
BS.lift0(buttons(lift)) = T.lit ⇒
F(DS.d0(doors(lift)) = T.open))

Note that all assertions must refer to a transition system (L in this case). Also note that "G" (globally in all states) and "F" (now or in the future) can be used as LTL temporal operators.

14.4 Confidence condition verification

The tool take an RSL file and it generates three SAL versions. The first version was described in previous sections, this section talks about the other two versions.

The first version allows one to do the verification under the assumption of CC satisfaction. But the tool allows one to use the automatic verification power of model checking to first certify the satisfaction of CC.

The tool generates therefore other two versions: A "CC" version that checks confidence conditions and another one "simple CC" that also checks CC but gives less diagnostic information. In order to provide CC verification, basically the translator does the following:

Type system A deeper embedding of type system is used, all types are lifted to new system types. This new type system includes in all SAL types a special datatype, called *Not_a_value_type* (also it includes user-defined ones).

The extended type system allows one easily to translate partial functions, to support explicit subtypes and to verify preconditions with model checking tools.

A new set of prefix functions must be used in the model. They provide the basic operations on each type, with the proper extensions to handle values of type Not_a_value_type (these values are called *navs*).

Partial Functions Code is added at the beginning of the function's body (i.e. the SAL conditional statement IF - THEN - ELSE) in order to verify the precondition satisfaction.

All functions take values of lifted types as arguments and return values of lifted types.

If any function's argument is a nav then the nav is returned. A nav is returned if any argument is not in its subtype, or if the precondition is violated, or if the result of the function is not in its subtype.

In the simple CC version Not_a_value_type is reduced to the single constant nav. This reduces the sizes of all the lifted types, and hence the size of the model, but provides less diagnostic information.

14.4.1 Model checking and confidence condition

A single LTL assertion is generated for each transition system, in order to check for CC violation. This says that all the local variables in the transition system are not navs, and if this is not true then SAL will produce a trace showing how the nav was generated. If a CC check succeeds then no CC violation occurs in the evolution of the transition system.

14.5 Using the tool

The RSL-SAL translator works on different operative systems. In particular, it is possible to use Linux and Windows platforms.

For using the tool on Windows platform it is necessary to:

- Install DJGPP for RAISE and RAISE setup
  (ftp://ftp.iist.unu.edu/pub/RAISE/rsltc/windows/)
- Install the Linux-like environment for Windows "Cygwin"
  (http://www.cygwin.com)
  In "Cygwin Setup" select "Interpreters Packages" and install "m4".
- Install SAL model checker for Windows (http://sal.csl.sri.com/)

For using the tool on Linux platform it is necessary to:

- Install "rsltc-2.5-1.i386.rpm" or "rsltc_2.5.1-1_i386.deb" (ftp://ftp.iist.unu.edu/pub/RAISE/rsltc/linux/)
- Install the Unix macro processor"m4" (from the Linux distribution)
- Install SAL model checker for Linux (http://sal.csl.sri.com/)

14.5.1 Activating the SAL translator

The translator's activation is through the Emacs editor, so for using the RSL-SAL translator

- Open a file with extension ".rsl" using Emacs editor
- Select the "RSL" option
- Execute the "Translate to SAL" option

When the "Translate to SAL" option is executed, the tool generate some files (they are in the same folder where .rsl file is). Some of these files are:

- **SAL_TYPES.sal** file contains the RSL type declarations translated to SAL types. This type declaration is used to avoid circular dependencies among SAL modules.
- **SAL_GLOBAL.sal** file contains the boundaries for Integer type and Natural type. These are values for *DefaultNatHigh* (default 4), *DefaultIntLow* (default -4) and *DefaultIntHigh* (default 4). See section 14.2.1 to see how the default values may be changed.
- **IT_AN.sal** file contains the definition of Integer type:

```
IT_AN : CONTEXT =
BEGIN
  Int_: TYPE = [SAL_GLOBAL!DefaultIntLow .. SAL_GLOBAL!DefaultIntHigh];

END
```

- **L_BUILTIN.sal** file contains the definition of Integer type for checking CC:

```
L_BUILTIN : CONTEXT =
BEGIN
  Not_a_value_type: TYPE = DATATYPE
    ...
  END;

  Bool__cc: TYPE = DATATYPE
    Bool__cc(Bool__val: BT_AN!Bool_),
    Bool__nav(Bool__nav_val: Not_a_value_type)
  END;

  Int__cc: TYPE = DATATYPE
```

```
    Int__cc(Int__val: IT_AN!Int_),
    Int__nav(Int__nav_val: Not_a_value_type)
  END;

END
```

- **L_BUILTIN_simple.sal** file contains the definition of Integer type for checking CC simple (it gives less diagnostic information):

```
L_BUILTIN_simple : CONTEXT =
BEGIN
  Not_a_value_type: TYPE = DATATYPE
    nav
  END;

  Bool__cc: TYPE = DATATYPE
    Bool__cc(Bool__val: BT_AN!Bool_),
    Bool__nav(Bool__nav_val: Not_a_value_type)
  END;

  Int__cc: TYPE = DATATYPE
    Int__cc(Int__val: IT_AN!Int_),
    Int__nav(Int__nav_val: Not_a_value_type)
  END;

END
```

- **NT_AN.sal** file contains the definition of Natural type:

```
NT_AN : CONTEXT =
BEGIN
  Nat_: TYPE = [0 .. SAL_GLOBAL!DefaultNatHigh];

END
```

- **BT_AN.sal** file contains the definition of Boolean type:

```
BT_AN : CONTEXT =
BEGIN
  Bool_: TYPE = BOOLEAN;

END
```

- <**file**>**.sal** file contains definitions of the model. This file only is generated if some valid RSL value declarations exist in the source RSL file.
- <**file**>**_cc.sal** file contains definitions of the model to allow checking of CC. This file only is generated if some valid RSL value declarations exist in the source RSL file.
- <**file**>**_cc_simple.sal** file contains definitions of the model to allow checking CC, giving less diagnostic information. This file only is generated if some valid RSL value declarations exist in the source RSL file.

- For each Set type or Map type declared, the following files are generated:

  - <NameType>_<Type>_OPS.sal

  - <NameType>_<Type>_cc_OPS.sal

  - <NameType>_<Type>_cc_OPS_simple.sal

  where <NameType> is the name in the declaration and <Type> is Map or Set. In these files there are functions such as "emptySet"function and "add" function (of Set type) used to translate RSL expressions to SAL (see section 14.2.5).

Once the "Translate to SAL" option has been executed, it is possible to check if the specification resulting from the translation is well formed. This one is activated in the Emacs editor, selecting the "RSL" option and the "Run SAL well-formed checker" option. This step also generates some .sal files from some .m4 files and so is essential.

After executing the "Run SAL well-formed checker" option, other utilities provided for the tool are enabled. Also they are activated using the Emacs editor. In this case, it is necessary to select the "RSL" option and some one of the following,

```
Run SAL deadlock checker
  Base
  CC
  Simple CC
Run SAL model checker
  Base
  CC
  Simple CC
```

In the next subsection an example is used to show how the tool works in more detail.

14.5.2 An example

The following code is in a file named "TOKENS.rsl",

```
scheme TOKENS =
  class
    type
      Token == a | b | c | d | e | f,
      State ::
        S1 : Token-set ↔ re_S1
        S2 : Token-set ↔ re_S2

    value
      init : State =
        mk_State({a, b, c}, {d, e, f}),

      give21 : Token × State → State
```

give21(t, s) ≡
re_S1({t} ∪ S1(s), re_S2(S2(s) \ {t}, s))

give12 : Token × State → State
give12(t, s) ≡
re_S2({t} ∪ S2(s), re_S1(S1(s) \ {t}, s)),

transition_system
[sys]
local state : State := init
in
([] tok : Token •
[give21]
tok ∈ S2(state) ⟶
state′ = give21(tok, state))
[]
([] tok : Token •
[give12]
tok ∈ S1(state) ⟶
state′ = give12(tok, state))
end

ltl_assertion
[consistent] sys ⊢ G(S1(state) ∩ S2(state) = {}),
[no_loss]
sys ⊢
G(S1(state) ∪ S2(state) = {a, b, c, d, e, f}),
[empty_S1_reachable] sys ⊢ G(S1(state) ≠ {}),
[empty_S2_reachable] sys ⊢ G(S2(state) ≠ {})
end

- Translating to SAL If the "TOKENS.rsl" file is opened using the Emacs editor and the "Translate to SAL" option is executed, the tool will generate these files,

```
Bool__cc_OPS.m4
Bool__cc_OPS_simple.m4
BT_AN.sal
Int__cc_OPS.m4
Int__cc_OPS_simple.m4
Int__OPS.m4
IT_AN.sal
L_BUILTIN.sal
L_BUILTIN_simple.sal
NT_AN.sal
SAL_GLOBAL.sal
SAL_TYPES.sal
SAL_TYPES_cc.sal
SAL_TYPES_cc_simple.sal
```

```
State_cc_OPS.m4
State_cc_OPS_simple.m4
Token_cc_OPS.m4
Token_cc_OPS_simple.m4
Token_set_cc_OPS.m4
Token_set_cc_OPS_simple.m4
Token_set_OPS.m4
TOKENS.sal
TOKENS_cc.sal
TOKENS_cc_simple.sal
```

- Running SAL well-formed checker Now, it is possible to check if "TOKENS.sal", "TOKENS_cc.sal" and "TOKENS_cc_simple.sal" are well formed. For that, the tool first copies prelude files, and later it generates .sal files from .m4 files.

The well formed checking is enabled selecting the "Run SAL well-formed checker" option. So, the compilation buffer shows,

```
sal_wfc_check TOKENS

sal-wfc TOKENS
Ok.
sal-wfc TOKENS_cc
Ok.
sal-wfc TOKENS_cc_simple
Ok.

Compilation finished at ...
```

And the following files are generated:

```
Bool__cc_OPS.sal
Bool__cc_OPS_simple.sal
Bool_cc_prelude
cc_type_prelude
Int__cc_OPS.sal
Int__cc_OPS_simple.sal
Int__OPS.sal
int_cc_prelude
int_prelude
map_cc_prelude
map_prelude
set_cc_prelude
set_prelude
State_cc_OPS.sal
State_cc_OPS_simple.sal
Token_cc_OPS.sal
```

```
Token_cc_OPS_simple.sal
Token_set_cc_OPS.sal
Token_set_cc_OPS_simple.sal
Token_set_OPS.sal
```

- Running SAL deadlock checker SAL model checking is only valid if there are no deadlock states. The tool allows one to check for no deadlock states, selecting the "Run SAL deadlock checker" option. Next, it is necessary to choose between "base", "CC", or "simple_CC" options. In all these cases, the Emacs minibuffer shows,

```
Transition system identifier:
```

Then it is necessary to specify a transition system identifier (in the TOKEN example its name is "sys"),

```
Transition system identifier:  sys
```

So, if previously the "base" option was selected, the compilation buffer will show,

```
sal-deadlock-checker TOKENS sys

ok (module does NOT contain deadlock states).
```

- Running SAL model checker The tool allows one to run the SAL model checker, selecting the "Run SAL model checker" option. Next, it is necessary to choose between "base", "CC", or "simple_CC" options. In all these cases, the Emacs minibuffer shows,

```
Assertion identifier (default all assertions):
```

If an assertion identifier is specified (for example, "consistent"), and previously the "base" option was selected, the compilation buffer will show,

```
sal-smc TOKENS consistent

proved.
```

But if no particular assertion identifier is specified, by default all assertions are checked. In this case, the result is

```
sal-smc TOKEN2

Counterexample for 'empty_S1_reachable' located at [Context: TOKEN2, line(49), column(0)]:
========================
Path
========================
Step 0:
--- System Variables (assignments) ---
```

```
state =
mk_State((LAMBDA (arg!9 : Token):
           (arg!9 /= f) and (arg!9 /= e) and (arg!9 /= d)),
         (LAMBDA (arg!10 : Token):
           (arg!10 = f) or (arg!10 = e) or (arg!10 = d)))
------------------------
Transition Information:
(module instance at [Context: TOKEN2, line(47), column(21)]
  (with tok = a at [Context: TOKEN2, line(40), column(0)]
    (label give12
      transition at [Context: TOKEN2, line(42), column(4)])))
------------------------
Step 1:
--- System Variables (assignments) ---
state =
mk_State((LAMBDA (arg!11 : Token):
           (arg!11 /= f) and
           (arg!11 /= e) and
           (arg!11 /= d) and
           ((arg!11 = c) or (arg!11 = b))),
         (LAMBDA (arg!12 : Token):
           (arg!12 = f) or
           (arg!12 = e) or
           (arg!12 = d) or
           (arg!12 /= c) and (arg!12 /= b)))
------------------------
Transition Information:
(module instance at [Context: TOKEN2, line(47), column(21)]
  (with tok = c at [Context: TOKEN2, line(40), column(0)]
    (label give12
      transition at [Context: TOKEN2, line(42), column(4)])))
------------------------
Step 2:
--- System Variables (assignments) ---
state =
mk_State((LAMBDA (arg!13 : Token):
           (arg!13 /= f) and
           (arg!13 /= e) and
           (arg!13 /= d) and
           (arg!13 /= c) and
           (arg!13 = b)),
         (LAMBDA (arg!14 : Token):
           (arg!14 = f) or
           (arg!14 = e) or
           (arg!14 = d) or
           (arg!14 = c) or
           (arg!14 /= b)))
------------------------
Transition Information:
(module instance at [Context: TOKEN2, line(47), column(21)]
  (with tok = b at [Context: TOKEN2, line(40), column(0)]
```

```
    (label give12
      transition at [Context: TOKEN2, line(42), column(4)])))
------------------------
Step 3:
--- System Variables (assignments) ---
state =
mk_State((LAMBDA (arg!15 : Token): false),
         (LAMBDA (arg!16 : Token): true))

Counterexample for 'empty_S2_reachable' located at [Context: TOKEN2, line(50), column(0)]:
========================
Path
========================
Step 0:
--- System Variables (assignments) ---
state =
mk_State((LAMBDA (arg!17 : Token):
           (arg!17 /= f) and (arg!17 /= e) and (arg!17 /= d)),
         (LAMBDA (arg!18 : Token):
           (arg!18 = f) or (arg!18 = e) or (arg!18 = d)))
------------------------
Transition Information:
(module instance at [Context: TOKEN2, line(47), column(21)]
  (with tok = d at [Context: TOKEN2, line(34), column(0)]
    (label give21
      transition at [Context: TOKEN2, line(36), column(4)])))
------------------------
Step 1:
--- System Variables (assignments) ---
state =
mk_State((LAMBDA (arg!19 : Token):
           (arg!19 /= f) and (arg!19 /= e)),
         (LAMBDA (arg!20 : Token): (arg!20 = f) or (arg!20 = e)))
------------------------
Transition Information:
(module instance at [Context: TOKEN2, line(47), column(21)]
  (with tok = f at [Context: TOKEN2, line(34), column(0)]
    (label give21
      transition at [Context: TOKEN2, line(36), column(4)])))
------------------------
Step 2:
--- System Variables (assignments) ---
state =
mk_State((LAMBDA (arg!21 : Token):
           (arg!21 = f) or (arg!21 /= e)),
         (LAMBDA (arg!22 : Token): (arg!22 /= f) and (arg!22 = e)))
------------------------
Transition Information:
(module instance at [Context: TOKEN2, line(47), column(21)]
  (with tok = e at [Context: TOKEN2, line(34), column(0)]
    (label give21
```

```
      transition at [Context: TOKEN2, line(36), column(4)])))
------------------------
Step 3:
--- System Variables (assignments) ---
state =
mk_State((LAMBDA (arg!23 : Token): true),
         (LAMBDA (arg!24 : Token): false))

Summary:
The assertion 'consistent' located at [Context: TOKEN2, line(47), column(0)] is valid.
The assertion 'no_loss' located at [Context: TOKEN2, line(48), column(0)] is valid.
The assertion 'empty_S1_reachable' located at [Context: TOKEN2, line(49), column(0)] is invalid.
The assertion 'empty_S2_reachable' located at [Context: TOKEN2, line(50), column(0)] is invalid.

Compilation finished ...
```

14.5.3 Confidence conditions

SAL is only sound when there are no deadlocks. So you have to check for deadlocks in the CC version as well as the basic one.

To illustrate this, and show how to get information from the CC version, we use the lift example from section 14.3.4. Suppose we change, in the BUTTONS module, the definition of "required_beyond" to

```
-- wrong version
required_beyond : T.Direction × T.Floor × Buttons → Bool
required_beyond(d, f, bs) ≡
   let f′ = T.next_floor(d, f) in
      required_here(d, f′, bs) ∨
       T.is_next_floor(d, f′) ∧
       let f′′ = T.next_floor(d, f′) in
         required_here(d, f′′, bs)
      end
   end ∧ T.is_next_floor(d, f)
```

Note that we gave moved the conjunct T.is_next_floor(d, f) from the beginning to the end. This looks logically equivalent to the original, because ∧ is normally commutative. But we have to remember RSL's left-to-right evaluation rule, and also recall that T.is_next_floor(d, f) is a precondition of T.next_floor(d, f). In the original version, if T.is_next_floor(d, f) is **false**, then T.next_floor(d, f) is not evaluated. In the new version, this is not so and we can get a precondition violation.

If we make this change to required_beyond, recompile to SAL, check well-formedness, and model check the CC version it reports all is well. But if we remember to also run the deadlock checker on the CC version it reports a deadlock:

```
Total number of deadlock states: 1.0
Deadlock states:
```

```
State 1
--- System Variables (assignments) ---
lift =
Lift_cc(mk_Lift_(mk_Cage_(up, halted, 0),
                 mk_Doors_(open, shut, shut),
                 mk_Buttons_(lit, lit, lit, lit, lit, lit, lit)))
```

This tells us that the "no errors" report from the CC model check cannot be trusted, but otherwise is not very helpful, because no Nav value is reported. We might notice that the buttons have all become lit, which means that the press transition is not enabled. The other transition must also not be enabled (since we have a deadlock), which means that can_next(lift) must not be true. In fact can_next contains a Nav. We can investigate what is happening by adding a variable to the system so that we can see the value of can_next:

- We add a variable cn of type **Bool** to the transition system. We initialise it to **true**: the chosen initial value does not matter since nothing will depend on it.
- We set cn to the value of can_next at the end of each transition, by adding the update

  $\text{cn}' = \text{can\_next}(\text{lift}')$

  to each of the two transitions. Note the use of lift' here so that cn contains the guard to be used in the *next* transition: we want to see what causes the deadlock before it happens.
- We retranslate to SAL, check well-formedness, and run the CC model check.

Now we get the result that the check on the variables is invalid: cn contains a Nav:

```
Step 1:
--- System Variables (assignments) ---
lift =
Lift_cc(mk_Lift_(mk_Cage_(up, halted, 0),
                 mk_Doors_(open, shut, shut),
                 mk_Buttons_(clear, lit, clear, clear, clear, clear, clear)))
cn =
Bool__nav(Precondition_of_function_T_next_floor_not_satisfied)

Summary:
The assertion 'LIFT_max_floor_cc_check' located at [Context: LIFT_cc, line(180), column(0)] is val

The assertion 'LIFT_min_floor_cc_check' located at [Context: LIFT_cc, line(181), column(0)] is val

The assertion 'LIFT_L_cc_check' located at [Context: LIFT_cc, line(209), column(0)] is invalid.
```

and we see that T.next_floor is the function whose precondition is violated. We can proceed in the same way if necessary, adding more variables to show how the values of particular expressions change in this run, until we find the error.

15 Use with emacs

This section describes the Unix/Linux version. If you use Windows you should also read section 15.1 for the differences.

emacs provides a convenient environment for using rsltc. Its "compile" facility allows a user to see error messages in a separate window and just click on them (middle button) to go straight to the source of the error in the appropriate file. Additional tools, like VCG (section 9) and the SML run-time tool (section 10) can be invoked automatically or from a menu.

There are a number of files supplied with the tool that support use with emacs:

rsl-mode.el	syntax highlighting; RSL menu
rsltc.el	rsltc interface
rslconvert.el, tokenise.el	conversion to LaTeX

Table 20: emacs files

These are best placed in a directory on your emacs load-path. Then put in your .emacs file

```
(load "rsl-mode")
(load "rslconvert")
```

This will load all the emacs files.

If the files are not in your emacs load-path, then you can use `load-file` instead of `load`. You will have to load all four files, and give a path name and the `.el` extension as part of the file name string, as in

```
(load-file "/home/me/rsltc.el")
(load-file "/home/me/rsl-mode.el")
(load-file "/home/me/tokenise.el")
(load-file "/home/me/rslconvert.el")
```

You will find `rslconvert.el` useful if you use LaTeX for documentation. See section 18.

When `rsl-mode.el` is loaded and you open an RSL file (with extension `.rsl`) in emacs then you will find an RSL menu from which the rsltc tool components can be invoked.

Apart from the pretty printer, emacs will start a second window to display output. Errors and confidence conditions start with a `file:line:column` string and clicking with the middle mouse button on a line starting with this string places the cursor at the relevant position in that file in the first window.

The pretty-printer puts its output in the same buffer as the input, but does not save it, and offers an "undo" option to go back to the the state before pretty printing.

There is also a menu item to run the SML run-time system on the output from the SML translator. See also section 16.

When the "Draw module dependency graph" menu item is selected, emacs will automatically start the VCG tool on the output.

15.1 Emacs on Windows

If you obtain a full version of emacs for Windows then things will run as described in section 15 and you need read no further, except to note that if you have a two-button mouse, pressing both buttons emulates the middle mouse button normal with Unix and Linux.

There is a cut down version of emacs, version 19.34, supplied with rsltc for Windows. This lacks the ability to run a sub-process, and so two features of the use with emacs described in section 15 are a little different:

- To run VCG, first use the menu item "Draw module dependency graph" from `X.rsl`. This will generate `X.vcg` but will not display it. Start a DOS shell, move to the directory where `X.rsl` is located, and use the following command to start VCG:

  ```
  <dir>/vcg X.vcg
  ```

  where `vcg.exe` is stored in `<dir>`.

- To run SML, first use the menu item "Translate to SML" from `X.rsl`. This will generate `X.sml` and `X_.sml`, where the latter is loaded by the former. Start a DOS shell, move to the directory where `X.rsl` is located, and use the following command to start SML:

  ```
  <dir>/sml
  ```

  where `sml.exe` is stored in `<dir>`.

  This starts the SML run-time system in its own window. In that window, after the prompt `-`, give the command

  ```
  use "X.sml";
  ```

  Note the semicolon ";" at the end of this command. If you forget it you will get a prompt `=` on the next line, and you can type it there.

16 Mutation testing

This section assumes you have installed emacs. It also assumes you are familiar with the test case feature recently added to RSL: see section 2.7.

The tool supports mutation testing with the SML translator and SML run-time system. Mutation testing is a technique to check the adequacy of test cases by seeing if a small change, a *mutation*, of the RSL source gives different test results. If the test results are different it shows that this mutation can be detected if made by error. If the test results are not different it suggests that more test cases might

be needed. This is a useful technique when the test cases are developed during specification and later used on the final software. Mutation testing can increase the thoroughness of the testing, and hence the quality of the test cases.

Note that the support for mutation testing currently depends on all the RSL files needed being stored in one directory.

Before you start mutation testing you need to run the SML translator on the original files. Open the file with the test cases in it, translate to SML (RSL menu `Translate to SML`), run the SML translator (`Run SML file`) and save the results (`Save results from SML run`). See section 10.2.2 on saving results.

To make a mutation in an RSL module, open the module file, `X.rsl`, say, in emacs. Select the expression you want to change with the mouse by dragging, or by clicking with first the left at the beginning and then the right mouse button just past the end. Then select the item `Make mutation` in the RSL menu. (This menu item will not appear until you have selected something.)

You will first be prompted for the replacement text. For example, you might have selected `<` and want to try replacing it with `<=`. You type `<=` and hit `Enter`.

Then you are prompted for the file to translate. This might be `X.rsl` if the test cases are in the same file, or it might be a different file, such as `TEST_X.rsl`. The first time you will be offered `X.rsl` and can change it to `TEST_X.rsl` if you need to. If you make this change and later in the same session make another mutation to `X.rsl` you will be immediately offered `TEST_X.rsl` as the file to translate. Having corrected the offered file name if necessary, hit `Enter`. We will assume in the rest of this section that the file to be translated is `TEST_X.rsl`.

The RSL files in the current directory are copied to a subdirectory `mutantn`, where the final `n` is initially `0` (zero), and the mutation is applied to `mutantn/X.rsl`. Then the SML translator is run on `mutantn/TEST_X.rsl`, and provided there are no errors the SML run-time system is run on the output. Then you can see if there are any differences from the output of the original run before you made any mutations.

If there are many test cases and you want some automated support with the comparison, go back to the file you just translated, `mutant0/TEST_X.rsl`, say, or the original `X.rsl`, or `TEXT_X.rsl`, and select `Compare with mutant` in the `RSL` menu. This will save the results from the latest SML run, and offer two results files to compare, in this case `TEST_X.sml.results` and `mutant0/TEST_X.sml.results`. You can change these if you wish, or just hit `Enter` for each immediately if they are what you want. The "ediff" file comparison tool is then run on the two selected results files. This tool generates a small command window which shows how many differences there are, as well as showing in two main buffers the results files with differences highlighted. With the point in the command window, the main command keys are space bar to highlight the next difference, `Delete` to highlight the previous one, `q` to quit ediff (confirmed by `y`), and the toggle `?` to show/hide all the commands available.

You can go back to `X.rsl` to make another mutation, which will be placed in subdirectory `mutant1`, starting by selecting an expression to change (perhaps the same expression) and using `Make mutation` again.

You can also tidy up by selecting the `RSL` menu item `Delete mutant directories` in the original `X.rsl` or `TEST_X.rsl` buffers. This will delete the subdirectories `mutantn` and their contents (including the results files: move these first if you need to preserve them).

17 Test coverage support

This section assumes you have installed emacs. It also assumes you are familiar with the test case feature recently added to RSL: see section 2.7.

The tool supports test coverage analysis with the SML translator and SML run-time system. When you run the translated SML code to execute some test cases you may see messages of the form

```
Unexecuted expressions in X.rsl
```

or

```
Complete expression coverage of X.rsl
```

There may be several such messages if the system being tested involves several RSL modules. The first indicates that not all expressions in X.rsl were executed; the second indicates that they all were.

In the first case, to see the unexecuted expressions, open the file `X.rsl` and select the RSL menu item `Show test coverage`. The unexecuted expressions will be highlighted in red. The highlighting can be cancelled again by selecting the RSL menu item `Cancel test coverage`. It is possible that several coverage files from different tests are being merged to create an overall coverage, and the result of the merge may be to show that all expressions were in fact executed, in which case instead of some highlighting you will see the message `Coverage is complete`.

It is possible to run several tests (i.e. translate and run several different RSL files containing test cases) and combine the results. This is done automatically. The coverage results from executing `X.rsl` are held in files `X.rsl.eln`, where `n` may be missing or may be an integer.

If the file `X.rsl` is edited in any way then the coverage results are redundant, as the behaviour may have changed. Even editing such as pretty printing, which makes no functional changes, invalidates the coverage results as they are based on the positions in the buffer of the unexecuted expressions. So as soon as you change `X.rsl` you will get the RSL menu item `Delete old coverage results`, which should be selected to remove the `X.rsl.eln` files. It is sensible but not critical to remove such old coverage files: any coverage file `X.rsl.eln` older than `X.rsl` is ignored when calculating the coverage.

Note that the support for test coverage analysis currently depends on all the RSL files needed being stored in one directory.

18 LaTeX support

This section assumes you have installed emacs.

If you use LaTeX you should get and install auctex `http://www.gnu.org/software/auctex` which provides very useful support for LaTeX in emacs. A good guide to getting and installing emacs, auctex, Miktex (a LaTeX for Windows) plus previewers is `NTTeXing.html` by Willem Minton (supplied with rsltc).

Then include in your .emacs file

```
(require 'latex)
(load "rslconvert")
(define-key LaTeX-mode-map "\C-cs" 'latex-symbol)
(define-key LaTeX-mode-map "\C-c\C-t" 'do-rsl)
(define-key LaTeX-mode-map "\C-c\C-u" 'undo-rsl)
```

Now you can do several useful things:

- typing the ASCII for an RSL symbol, like "`->`", and then immediately pressing Ctrl-c followed by s changes it to "`{\RIGHTARROW}`".
- You can put a larger piece of RSL be between `\RSLatex` and `\endRSLatex`. These look like LaTeX commands but are in fact not. For example, you type

  ```
  \RSLatex
  if a > b then c
    else d
  end
  \endRSLatex
  ```

  where `\RSLatex` and `\endRSLatex` must be in lines of their own and with no spaces in front of them. Ctrl-c Ctrl-t with the point anywhere after the `\RSLatex` changes this to

  ```
  %\RSLatex
  %if a > b then c
  %  else d
  %end
  %\endRSLatex
  \bp
  \kw{if} a {\GT} b \kw{then} c\\
  \>\kw{else} d\\
  \kw{end}
  \ep
  ```

  which will print with key words bold, the LaTeX versions of symbols, and with the original layout (it is best to use two spaces for each indentation). Ctrl-c Ctrl-u will revert to the original form for editing.
- The command `\RAISEIN{X}` will cause a LaTeX version of `X.rsl` to be included, provided you first use the emacs command `M-x mkdoc` on the LaTeX file that includes one or more such `\RAISEIN` commands. `mkdoc` generates a file `X.tex` for each such file, unless there is already such a file dated later than the corresponding `X.rsl` file. So it is easy to keep documents up to date with the latest versions of your RSL files.

  The file name in the `\RAISEIN` can include a path if `X.rsl` is in a different directory, but should not include `.rsl`.

`\RAISEINBOX` is an alternative to `\RAISEIN` that also puts a frame around the RSL.

The definitions of the RSL symbols are in `rslenv.sty`, and so your LaTeX document needs to include in the preamble the line

```
\usepackage{rslenv}
```

(If you still use `\documentstyle`, i.e. you are not using LaTeX2e, then you include `rslenv` as an option to `\documentstyle`.)

19 Installation

19.1 Unix and Linux

We assume you already have emacs installed. If you haven't, then rsltc just runs from the command line and there is nothing to do beyond installing the executable.

The files

```
rsl-mode.el
rsltc.el
rslconvert.el
tokenise.el
```

should be placed in a directory on your emacs load-path. Byte-compile them if you wish.

The files

```
rslenv.sty
boxedminipage.sty
```

should be placed where LaTeX will find them. If you or your system administrator don't know how to do that, then put them in the same directory as your LaTeX source. (You may already have boxedminipage.sty.)

19.1.1 SML

The files

```
rslml.cm
rslml.sml
```

should be placed anywhere you like. Make sure that whoever first runs SML on an sml file produced from RSL has write access to the directory, as the first load will compile the RSL library.

You also need to set the environment variable `RSLML_PATH` to the directory containing them, e.g. `/usr/local/sml/rslml`. You can get your system administrator to set this up, or you can do it individually as follows. Use the shell command

```
env | grep SHELL
```

to see what your login shell is: you should get something like `SHELL=/bin/csh`, showing it is csh, or some variant of it, or `SHELL=/bin/bash`, showing it is bash. In the first case, edit your `.cshrc` file in your home directory to include something like

```
setenv RSLML_PATH /usr/local/sml/rslml
```

In the second case, edit your `.bash_profile` file in your home directory to include something like

```
export RSLML_PATH=/usr/local/sml/rslml
```

In either case you will need to logout and login again for the environment variable `RSLML_PATH` to be set.

SML can be downloaded and unpacked from instructions and tar files obtainable from `http://www.smlnj.org`.

19.1.2 C++

The files

```
RSL_comp.h
RSL_list.cc
RSL_list.h
RSL_map.cc
RSL_map.h
RSL_prod.h
RSL_set.cc
RSL_set.h
RSL_text.h
RSL_typs.cc
RSL_typs.h
cpp_RSL.cc
cpp_RSL.h
cpp_io.cc
cpp_io.h
cpp_list.cc
cpp_list.h
```

```
cpp_map.h
cpp_set.cc
cpp_set.h
```

should be placed in a directory, and the file `rslcomp` edited to set CPP_DIR to that directory.

Put rslcomp somewhere on your path, and make it executable.

19.1.3 VCG

VCG is built from `vcg.1.30.r3.17.tar.gz`, supplied with rsltc. The executable is called `xvcg` and should be installed somewhere on your path.

19.1.4 rsltc

Finally install the executable `rsltc` somewhere on your path.

If rsltc and xvcg are not on your path, you can put the full names in rsltc.el by changing the definitions of `rsltc-command` and `vcg-command` respectively.

19.1.5 UML2RSL

See section 13.4.

19.2 Windows

There is a minimal version of emacs, version 19.34, supplied with rsltc. This was copied from the DJGPP distribution (`http://www.delorie.com/djgpp/`). It is better to get a full version of the latest version emacs from DJGPP, or from `http://www.gnu.org/software/emacs/emacs.html`. You can get it precompiled for Windows.

rsltc comes as a zip file `rsltc.zip`.

You need WinZip or equivalent to unpack the zip files. The dos tool pkunzip can be used with the -d option but you will afterwards need to correct the following file names in `gnu\emacs\lisp`, because the base names will have been truncated to 8 characters:

```
backquote.elc
case-table.elc
cc-compat.elc
dired-aux.elc
help-macro.elc
```

If you have a directory `c:\raise` you might want to rename it or vary the instructions below to avoid the possibility of overwriting files in it.

Extract rsltc.zip into `c:`, using the "Use folder names" option. This will create a directory `c:\raise`.

If you don't already have emacs:

- Unpack em1934b.zip into `c:\raise`, again using the "Use folder names" option. Note that rsltc.zip was unpacked into `c:`: now you are unpacking into a folder created by the first unpack.
- Use Windows Explorer to open `C:\raise\gnu\emacs\bin` and select emacs.exe.
- Optionally, use Properties under File to set the font to 7 x 12 and the screen to 43 lines. This gives a full length emacs window and a reasonable font size.
- Use CreateShortcut to make a shortcut.
- The file `C:\raise\gnu\emacs\lisp\rsltc.el` needs to know where the executable rsltc.exe is. If you have followed the directions above exactly it will be right: `"/raise/rsl/rsltc"`. Otherwise, start emacs with the shortcut you made and use it to edit `C:\raise\gnu\emacs\lisp\rsltc.el` so that `rsltc-command` is set correctly. Save rsltc.el. Exit emacs (Files menu) and start it again.

If you already have emacs, or have installed a version from elsewhere:

- Move the .el files in `C:\raise\gnu\emacs\lisp` to the emacs site-lisp directory (probably `C:\emacs\site-lisp`).
- Move the emacs start-up file `_emacs` from `C:\raise\gnu\emacs` to your "home" directory. The simplest method is to find out where emacs thinks "home" is and put it there. You can use the sequence

  ```
  Esc-x getenv Enter HOME Enter
  ```

  (where "Enter" means the Enter or Return key) in emacs to find out. Otherwise, see the section "Where do I put my .emacs, (or _emacs), file?" in `emacs_windows_faq.html`.
- The file `C:\emacs\site-lisp\rsltc.el` needs to know where the executable rsltc.exe is. If you have followed the directions above exactly it will be right: `"/raise/rsl/rsltc"`. Otherwise, use to edit `C:\emacs\site-lisp\rsltc.el` so that `rsltc-command` is set correctly. Save rsltc.el. Exit emacs and start it again.

19.2.1 SML

Install SML from the self-installing package from `http://www.smlnj.org`. We assume below that you have installed it in `C:/sml` as suggested.

In Windows 9X, include in autoexec.bat

```
SET RSLML_PATH=c:\raise\sml
```

In Windows NT, or later versions, get your system administrator to set this variable.

Make a folder `c:\raise\sml` and copy the files rslml.sml and rslml.cm from rsltc.zip file to there.

19.2.2 C++

The files

```
RSL_comp.h
RSL_list.cc
RSL_list.h
RSL_map.cc
RSL_map.h
RSL_prod.h
RSL_set.cc
RSL_set.h
RSL_text.h
RSL_typs.cc
RSL_typs.h
cpp_RSL.cc
cpp_RSL.h
cpp_io.cc
cpp_io.h
cpp_list.cc
cpp_list.h
cpp_map.h
cpp_set.cc
cpp_set.h
```

should be placed in a directory, and the file `rslcomp.bat` edited to set CPP_DIR to that directory.

Put `rslcomp.bat` somewhere on your path.

`rslcomp.bat` assumes that you are using the DJGPP (`http://www.delorie.com/djgpp/`) port of g++ (called `gxx`). If your C++ compiler is called something else `rslcomp.bat` is easy to change.

The translator output has been tested using DJGPP and also the Cygwin port of GNU tools to Windows: `http://sources.redhat.com/cygwin/`.

If you are using Cygwin with emacs 20 you might like to use the Cygwin bash shell within emacs. You should add to your _emacs file the following (taken from the Cygwin FAQ):

```
;; This assumes that Cygwin is installed in C:\cygwin (the
;; default) and that C:\cygwin\bin is not already in your
;; Windows Path (it generally should not be).
;;
(setq exec-path (cons "C:/cygwin/bin" exec-path))
(setenv "PATH" (concat "C:\\cygwin\\bin;" (getenv "PATH")))
```

```
;;
;; NT-emacs assumes a Windows command shell, which you change
;; here.
;;
(setq process-coding-system-alist '(("bash" . undecided-unix)))
(setq w32-quote-process-args ?\")
(setq shell-file-name "bash")
(setenv "SHELL" shell-file-name)
(setq explicit-shell-file-name shell-file-name)
;;
;; This removes unsightly ^M characters that would otherwise
;; appear in the output of java applications.
;;
(add-hook 'comint-output-filter-functions
          'comint-strip-ctrl-m)
```

You should also change the definition of `rsltc-command` in `rsltc.el` to `"/cygdrive/c/raise/rsl/rsltc"`, and change `vcg-command` similarly.

Limited tests have been done using Microsoft's Visual C++ compiler. The translator's output needs to be different, making much less use of templates. If you use this compiler you should use the `rsltc` option `-cpp` instead of `-c++`. The C++ output file will have extension `.cpp` instead of `.cc`. You can select this option in emacs by using the RSL menu item `Translate to Visual C++`.

We would be interested to hear of people using the translator with other C++ compilers.

19.2.3 VCG

VCG is installed from two zip files supplied with rsltc: `vcg_p1.zip` and `vcg_p2.zip`. Unpack them in that order using WinZip or pkunzip -d and just follow the instructions in the readme file in the first. You can place vcg where you like, but unless you unpack the two zip files into `C:\raise` you will need to edit the definition of `vcg-command` in `rsltc.el` in `site-lisp`.

19.2.4 UML2RSL

See section 13.4.

20 Making it yourself

rsltc is open source and you are welcome to build it yourself.

You will need gcc, flex, bison and make (GNU versions all available on the web, `http://www.gnu.org` or, for Windows, the DJGPP versions from `http://www.delorie.com/djgpp/`) plus the Gentle Compiler Construction System from `http://gentle.compilertools.net/`.

Build Gentle first, then select a make file from amongst those supplied, like make_dos or make_solaris (which will also do for Linux) and edit it as necessary, at least to set the path for Gentle. All the real work is in std_make, and the top level make files just set some system dependent things. If your new make file is `make_myos`, just run `make -f make_myos` and you should get an executable rsltc (or rsltc.exe on Windows). It only remains to move this to somewhere on your path and make sure you have execute access to it.

If you create a make file for another platform and are willing to share it with others we would be glad to get a copy and include it in future releases.

21 Help and bug-reporting

The email address for seeking help or reporting bugs is raise@iist.unu.edu.

If you have problems we will be glad to try to assist.

If you discover errors please report them to us Remember to include enough information about the platform, operating system, and any input files, for us to recreate the problem.

Acknowledgements

A number of other people have worked on the RAISE tool and this user guide:

Tan Xinming, Wuhan Jiaotong University, Wuhan, China
Ms He Hua, Peking University, Beijing, China
Ke Wei, Chinese Academy of Science, Beijing, China
Univan Ahn, Kim Il Sung University, Pyongyang, DPR Korea
Ms Ana Funes, University of San Luis, San Luis, Argentina
Aristides Dasso, University of San Luis, San Luis, Argentina
Juan Perna, University of San Luis, San Luis, Argentina
Ms Ana Garis, University of San Luis, San Luis, Argentina

References

[1] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992. Available from Terma A/S. Contact jnp@terma.com.

[2] The RAISE Method Group. *The RAISE Development Method*. Prentice-Hall International (UK) Limited, 1995.

[3] He Hua. A Prettyprinter for the RAISE Specification Language. Technical Report 150, UNU-IIST, P.O.Box 3058, Macau, December 1998.

[4] Ke Wei and Chris George. An RSL to SML Translator. Technical Report 208, UNU-IIST, P.O. Box 3058, Macau, August 2000.

[5] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML — Revised*. MIT Press, 1997.

[6] Univan Ahn and Chris George. C++ Translator for RAISE Specification Language. Technical Report 220, UNU-IIST, P.O. Box 3058, Macau, November 2000.

[7] Aristides Dasso and Chris George. Transforming RSL into PVS. Technical Report 256, UNU-IIST, P.O. Box 3058, Macau, May 2002.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[9] Ana Funes and Chris George. Formal Foundations in RSL for UML Class Diagrams. Technical Report 253, UNU-IIST, P.O. Box 3058, Macau, May 2002. Published as chapter VIII *Formalizing UML Class Diagrams* of *UML and the Unified Process*, Liliana Favre (ed.), IRM Press, 2003.

[10] Zhiming Liu. Object Oriented Software Development using UML. Technical Report 229, UNU/IIST, March 2001.

[11] H. Maruyama, K. Tamura, and N. Uramoto. *XML and Java, Developing Web Applications*. Addison-Wesley, 2000.

[12] Steve Holzner. *Inside XML*. New Riders, 2001.

[13] Juan Ignacio Perna and Chris George. Model checking RAISE specifications. Technical Report 331, UNU-IIST, P.O.Box 3058, Macau, December 2005.

[14] Leonardo De Moura and Sam Owre. The SAL Language Manual. Technical report, SRI-CSL, 2003.

[15] Chris George. Introduction to RAISE. Technical Report 249, UNU-IIST, P.O. Box 3058, Macau, April 2002.

[16] Chris George. *Domain Modeling and the Duration Calculus*, volume 4710 of *LNCS*, chapter Applicative Modelling with RAISE, pages 51–118. Springer, 2007.