



The United Nations
University

UNU-IIST

International Institute for
Software Technology

Model checking concurrent RSL with CSP_M and FDR2

Lizeth Tapia and Chris George

May 2008

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

Model checking concurrent RSL with CSP_M and FDR2

Lizeth Tapia and Chris George

Abstract

This Report shows the application of model checking techniques over formal specifications expressed in RSL using the FRD2 refinement checker, for which we have developed a first version of a translator from RSL to CSPM. We give an overview of the semantic and syntactic differences between these two languages, then we define a translation subset and finally we show the strategy used to find the respective equivalences in order to make the translation possible; we also briefly describe the development of the translator and show the use of this translator with some typical concurrent examples.

Silvia Lizeth Tapia Tarifa is working for the San Pablo Catholic University in Arequipa, Peru. She was a UNU-IIST Fellow during April 2007 to April 2008. Her main research interest is formal methods and logic.

Chis W. George is a Senior Research Fellow at UNU/IIST, since 1 September 1994. He is one of the main contributors to RAISE, particularly the RAISE method and that remains one of his main research interests. Before coming to UNU/IIST he worked for companies in the UK and Denmark.

Contents

1	Introduction	1
2	Translation issues	3
2.1	RSL specification styles	3
2.2	Syntactic viewpoint	5
2.2.1	RSL	5
2.2.2	CSP_M	8
2.3	Semantic viewpoint	10
2.4	Translating RSL subset to CSP_M	13
2.4.1	Built-in types	13
2.4.2	Compound types	15
2.4.3	Constant values	18
2.4.4	Functions	18
2.4.5	Patterns	21
2.4.6	Channels	21
2.4.7	Processes	23
3	CSP_M translator tool	32
3.1	Implementation	32
3.2	Structure of the tool	33
3.3	Limitations of the tool	33
4	Summary	35
A	Examples of translation from RSL to CSP_M	37
A.1	The Cyclic Scheduler Example	37
A.1.1	The problem	37
A.1.2	The model	37
B	Examples of application of model checking with RSL using FDR2	42
B.1	The Dining Philosophers Example	42
B.1.1	The problem	42
B.1.2	The model	42
B.1.3	Property verification with model checking and FDR2	46
B.2	The Multiplexed buffer example	47
B.2.1	The problem	47
B.2.2	The model	47
B.2.3	Refinement verification with model checking and FDR2	51
C	Other Examples	53
C.1	The railway level crossing Example	53
C.1.1	The problem	53

C.1.2	The model	53
C.1.3	Property verification with model checking	55
C.2	The Producer-Consumer Example	56
C.2.1	The problem	56
C.2.2	The model	56
C.2.3	Property verification with model checking	58
C.3	The Alternating Bit Protocol (ABP) Example	58
C.3.1	The problem	58
C.3.2	The model	59
C.3.3	Verifying the refinement with model checking	61
C.4	The Multiplexed Buffer using the ABP Example	62
C.4.1	The problem	62
C.4.2	The model	62
C.4.3	Verifying the refinement with model checking	66

Chapter 1

Introduction

In computer science and software engineering, *Formal Methods* refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. Engineering relies heavily on mathematical models and calculation to make judgements about designs. Then, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to the reliability and robustness of software[7].

RAISE is an acronym for “Rigorous Approach to Industrial Software Engineering”. It provides facilities for the industrial use of Formal Methods in the development of software systems. The aim of RAISE is to enable the construction of more reliable software, software with fewer errors, better documented software and more easily maintainable software[11].

The RAISE Specification Language (*RSL*) is a language with a formal - mathematical basis, intended to support the precise definition of software requirements and the reliable development from such definitions to executable implementations. Particular aims of the language were to support large, modular specifications, to provide a range of specification styles (axiomatic and model-based; applicative and imperative; sequential and concurrent), and to support specifications ranging from abstract to concrete[3]. Complete information can be found in the books on RSL [4], and the method [9].

On the other hand, *CSP* is a notation for describing concurrent systems whose component processes interact with each other by communication. Simultaneously, *CSP* is a collection of mathematical models and reasoning methods which help people understand and use this notation [10].

The machine-readable dialect of CSP (*CSP_M*) is one result of a research effort with the primary aim of encouraging the creation of tools — like FDR2 — for CSP. *CSP_M* combines

the CSP process-algebra with an expression language which, while inspired by languages like Miranda/Orwell and Haskell/Gofer, has been adapted to support the notation of CSP [6].

FDR2 is a model checker (refinement checker) for establishing properties of models expressed in CSP. It allows the automatic checking of deadlock and livelock freedom as well as general safety and liveness properties.

A *model checker* is a tool which seeks to verify that a system which is defined by transitions between (sometimes very large) finite sets of states satisfies some specification and which performs the verification by traversing the entire state space. In model checking tools, the specification is usually defined in a language other than that used for the implementation (often a specially defined logic). What characterises a *refinement checker* is that the specification is another description in the same language [10].

In this report, we describe the development of a tool that translates a subset of RSL into CSP_M in order to apply refinement checking techniques over RSL scripts using existing tools that understand CSP_M scripts like *FDR2*.

This report is organised as follows: Chapter 2 is focused on a syntactic and semantic description of RSL and CSP_M in order to establish the translation subset between RSL and CSP_M and explains the translation strategy used in the development of the translator tool from RSL to CSP_M ; chapter 3 is focused on describing the construction of the translator tool between RSL and CSP_M ; chapter 4 shows the summary of this report; and appendixes A, B and C include some typical examples which shows how the translation of a script is made and how the properties and refinement is checked with *FDR2*.

Chapter 2

Translation issues

RSL allows the specification of data and concurrency whereas CSP is mainly process modelling. Although the aims of both languages are different; it is possible to define some syntactic and semantic equivalences in order to make the translation possible.

This chapter is focused on describing the similarities and differences of both languages in order to find the subset which can be translated from RSL to CSP_M .

2.1 RSL specification styles

RSL has a rich set of features that provides us with a wide range of *specification styles* ; these are:

- **Applicative / Imperative:**

Applicative specification style is similar to the style used in applicative programming languages such as Lisp, Miranda and Haskell, where there is no concept of a global execution history or state, so there are no variables. It treats the specification as the evaluation of a set of functions and avoids states and mutable data.

Imperative specification style is similar to languages like C, Modula-2, Pascal etc., where the global execution history or state is important. This style uses variable definitions and in some cases loops, and the sequential order of application of procedures and functions has an important effect on the results.

- **Sequential / Concurrent:**

Sequential specification style uses the sequencing operator to compose expressions inside

a main single procedure or function (one thread), then the evaluation of expressions is one after the other.

Concurrent specification style is similar to the CSP language where besides sequencing operator, other combinators like internal and external choice combinators as well as concurrent and interlocked compositions are used to express a specification in order to allow the interaction between multiple tasks (threads) implemented as processes.

- Axiomatic / Model-based:

The axiomatic specification style is a more abstract method of specification than the model-based specification style. Here the data is described by using axioms. Within the RAISE method, an axiomatic specification is generally developed into a model-based specification style when more concrete data structures are designed.

The model-based specification style is characterized because any data available to a specification is modelled. In this style; operations on data are described by how they affect the model of the data.

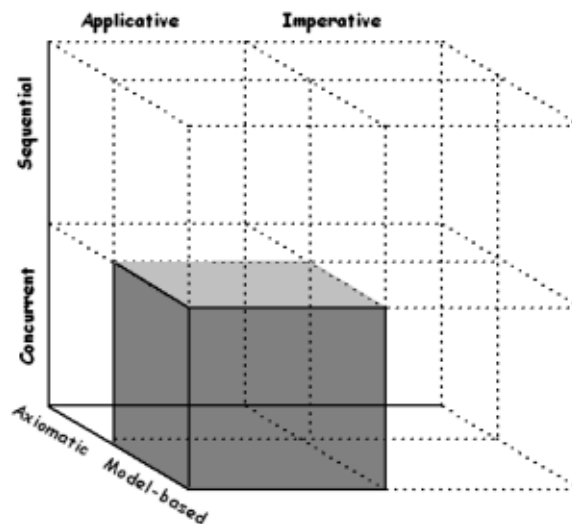


Figure 2.1: Subset of features being used

During the process of modelling a system in RSL a specification can be expressed as a combination of the different styles defined above. Since we are interested in the translation of RSL to CSP_M ; then the kind of specification which can be translated — as is showed in Figure 2.1 — will be a subset inside the applicative, concurrent and model-based specification styles.

The following script is an example of the specification of a vendor machine using the applicative, concurrent and model-based specifications styles:

```

scheme VENDOR_MACHINE =
  class
    channel
      coin, choc, toff: Unit
    value
      VMCT: Unit → in coin out choc, toff Unit
      VMCT() ≡ coin?;(choc!();VMCT() [] toff!();VMCT()),

      CSTM: Unit → out coin in choc, toff Unit
      CSTM() ≡ coin!();(choc?;CSTM() [] toff?;CSTM()),

      SYS: Unit → in coin,choc, toff out coin, choc, toff Unit
      SYS() ≡ VMCT()||CSTM()
  end

```

2.2 Syntactic viewpoint

2.2.1 RSL

RSL is a modular language where specifications are in general collections of (related) modules. There are two kinds of modules: schemes and objects. Schemes are essentially classes, and objects are instances of classes, then the basic concept is the class expression. These come in six forms: basic, extending, renaming, hiding, with, and instantiation.

Each class declaration can include the following declarations:

- Types: Built-in types and compound types
- Objects: Embedded modules
- Channels: For the input and output of data
- Values: Constants, functions and processes
- Variables: For storing values
- Axioms: Logical properties that must always hold
- Test cases: expressions to be evaluated by other translator tools
- Transition systems and temporal assertions: For the SAL model checker

No declarations are mandatory; any order and more than one occurrence of a kind of declaration are allowed [3].

RSL is a typed language. That is, it must be possible to associate each occurrence of an identifier representing a value, a variable or a channel with a unique type, and to check that the occurrence of the identifier is consistent with a collection of typing rules. The types that RSL supports are built-in types — Table 2.1 — and compound types — Table 2.2 and Table 2.3 — [3]

Compound types are types defined by the user; this type declaration consists of the keyword **type** followed by one or more type definitions separated by commas.

Type	Example values	Operators
Bool	true,false	$\wedge, \vee, \Rightarrow, \sim, =, \neq$
Int	..., -1, 0, 1...	$+, -, *, /, \backslash, \uparrow, <, \leq, >, \geq, \mathbf{abs}, \mathbf{real}$
Nat	0, 1...	Same as for Int
Real	..., -4.3, ..., 0.0, ...	$+, -, *, /, \backslash, \uparrow, <, \leq, >, \geq, \mathbf{abs}, \mathbf{int}$
Char	'a', ...	
Text	"", "Alice", ...	As for list of Char
Unit	()	

Table 2.1: RSL Built-in types [3]

Name	Type definitions
Product	Position = Real \times Real
Set	World = Country- set
List	Database = Data*
Map	Macao_residents = Id \xrightarrow{m} Name
Subtype	Gun = { i : Nat • i \in {0..15} }
Variant	ComplexColour == RGB(Red:Gun, Green:Gun, Blue:Gun) Black White
Record	Record_Person:: Id: Nat Info:Data

Table 2.2: RSL Compound Types - definition

Name	Example expressions	Operators
Product	(0.0,0.0), (-6.0,5.6)	
Set	{}, {China, Peru, Argentina}	hd , \in , \cup , \cap , \backslash , ...
List	$\langle \rangle$, $\langle D0, D1, D2, D1, D4 \rangle$	hd , tl , \in , \wedge , len , ...
Map	[], [543 \mapsto "Alice", 674 \mapsto "Wendy"]	dom , rng , hd , \in , \cup , \dagger , \backslash , $/$, ...

Table 2.3: RSL Compound Types - Examples and Operators

Object declarations consist of the keyword **object** followed by one or more object definitions separated by commas, as in the following example:

object

```
O1:class channel c:Unit end,
O2:class channel cInt:Int end
```

RSL supports concurrent system specification through message passing that uses channels for the communication between processes. Channel declarations consist of the keyword **channel** followed by one or more channel definitions separated by commas as in the following example:

channel

```
tick: Unit,
input, output: { | i : Nat • i ∈ {0..5} | },
ticket, cash, change: Unit
```

We define values within value declarations; a value declaration consists of the keyword **value** followed by one or more value definitions separated by commas.

A value definition can be a constant, a function or a function with channel access (process).

A constant is a value that does not change and it is defined as in the following example:

value

```
vInt: Int = 50,
vInt2: Int = 10\3,
vSet: Int-set = {4,-8},
vList: Int* = ⟨0..6⟩
```

A function is an entity that given an input value, executes some rules in order to generate an output value and it is defined as in the following example:

value

```
inv: Bool → Bool
inv(b) ≡ if b=true then false else true end,
```

```
list_sum : Int* → Int
list_sum(l) ≡
  case l of
    ⟨⟩ → 0,
    ⟨i⟩^l1 → i+list_sum(l1)
  end
```

RSL provides communication primitives, combinators — Table 2.4 — and terminators (**stop**, **skip**, **chaos**) to evaluate the concurrent communication between channels. Communication is synchronous in the sense that one process outputs data to a channel via ! while another one inputs data via ?, as is shown in the example:

value

TICKET: **Unit** → **in** cash **out** ticket **Unit**

TICKET() ≡ cash?;ticket!();TICKET(),

CHANGE: **Unit** → **in** change **out** cash **Unit**

CHANGE() ≡ cash!();change?;CHANGE(),

MACHINE: **Unit** → **in** cash,change **out** ticket,cash **Unit**

MACHINE() ≡ TICKET()||CHANGE()

Name	Combinator
Sequencing	;
External choice	□
Internal Choice	□
Parallel	
Interlocked	‡

Table 2.4: RSL Process Combinators

2.2.2 CSP_M

CSP_M includes a functional programming language, but its primary purpose is to support the description of parallel systems in a form which can be automatically manipulated. CSP_M scripts should, therefore, be regarded as defining a number of processes rather than a program in the usual sense [6].

The CSP_M expression language is rich and includes direct support for booleans, numbers (Table 2.5) sequences, sets, tuples (Table 2.6), user-defined types (Table 2.7), local definitions and pattern-matching.

In CSP_M , channel declarations consist of the keyword **channel** followed by one simple or multiple channel definition as in the following example:

channel input : {0..5}

Type	Example values	Operators
Booleans	<i>true, false</i>	$\wedge, \vee, \neg, ==, !=$
Numbers	$\dots, -1, 0, 1\dots$	$+, -, *, /, \%, <, \leq, >, \geq$
Void		

Table 2.5: CSP_M Built-in types

Name	Example expressions	Operators
Tuple	$(0,0), (-6,5)$	
Set	$\{1, 2, 3\}, \{m..n\}$	$\cup, \cap, -, \sqcup, \sqcap, \in, card, \dots$
Sequence	$\langle 1, 2, 3 \rangle, \langle m..n \rangle$	$\#, \wedge, head, tail, \dots$

Table 2.6: CSP_M tuples, sets and sequences - Examples and Operators

channel lift_piano, lift_table

CSP_M also allows the definition of constants, functions and processes; as is shown in the examples:

CONSTANTS $vNumber = suma(3, 7)$
 $vSet = \{2 * n \mid n \leftarrow \{2, 4, 6, 8\}\}$
 $vList = \langle 5, 6, 7 \rangle \wedge \langle 9 \rangle$

FUNCTIONS $suma(i, j) = i + j$
 $inc(i) = (i + 1) \% (vNumber + 1)$

PROCESSES $PETE = lift_piano \rightarrow PETE \sqcap lift_table \rightarrow PETE$

$DAVE = lift_piano \rightarrow DAVE \sqcap lift_table \rightarrow DAVE$

$TEAM = DAVE \parallel \{\{lift_piano, lift_table\} \mid \{lift_piano, lift_table\}\} \parallel PETE$

The language also supports the process terminators *SKIP*, *STOP* and *CHAOS()* and different process combinators as is shown in Table 2.8.

Since CSP_M does not support all of the syntactic features of RSL and has others features as well, then only the features supported by RSL that can have an equivalent translation to CSP_M will be included in the translation subset, as will be explained with more detail in Section 2.4.

Name	Type declaration
nametype	$Gun = \{0..15\}$
datatype	$ComplexColour = RGB.Gun.Gun.Gun \mid Black \mid White$

Table 2.7: CSP_M nametype and datatype - declaration

Name	Combinator
Simple and complex prefix	\rightarrow
Sequential composition	$;$
Interrupt	\triangle
Hiding	\backslash
External choice	\square
Internal Choice	\sqcap
Interleaving	\parallel
Parallel	$[[\mid]]$
...	

Table 2.8: CSP_M Process Combinators

2.3 Semantic viewpoint

Since our aim is to develop a tool that translates a subset of RSL into CSP_M in order to enable the application of model checking techniques where the concept of process is relevant and despite of the fact that the syntax inside the subset of processes of being translated (from RSL process to CSP_M process) is similar; an evaluation of the semantic meaning of a process in both cases is necessary.

A research about this evaluation was done by Abigail Parisaca in [8], where the following facts were reported:

1. Sequencing ($;$) in RSL and prefix (\rightarrow) in CSP_M are equivalent
2. The operational semantics of RSL [1] and the operational semantics of CSP_M [5] have the same rules for external and internal choice as is shown in Table 2.9.
3. Parallel combinator of two process expressions in RSL and CSP_M have differences in the semantic meaning.

The operational semantics rule for synchronization of RSL is shown in formula 2.8 below [1], where if a is an input ($c?x$) then \bar{a} is an output on the same channel ($c!e$), or vice

RSL [1]	CSP [5]
$(2.1) \quad \frac{\rho \vdash P \xrightarrow{a} P'}{\rho \vdash P \square Q \xrightarrow{a} P' \quad Q \square P \xrightarrow{a} Q' \quad a \neq \tau}$	$(2.2) \quad \frac{\frac{P \xrightarrow{a} P'}{P \square Q \xrightarrow{a} P'} \quad a \neq \tau}{Q \xrightarrow{a} Q'}{P \square Q \xrightarrow{a} Q'} \quad a \neq \tau$
$(2.3) \quad \frac{\rho \vdash P \xrightarrow{\tau} P'}{\rho \vdash P \square Q \xrightarrow{\tau} P' \square Q \quad Q \square P \xrightarrow{\tau} Q \square P'}$	$(2.4) \quad \frac{P \xrightarrow{\tau} P'}{P \square Q \xrightarrow{\tau} P' \square Q}$
$(2.5) \quad \frac{\rho \vdash P \square Q \xrightarrow{\tau} P \quad Q \square P \xrightarrow{\tau} Q}$	$(2.6) \quad \frac{}{P \square Q \xrightarrow{\tau} P}$ $(2.7) \quad \frac{}{P \square Q \xrightarrow{\tau} Q}$

Table 2.9: RSL and CSP_M - external and choice operational semantics

versa, and the resulting event after the communication is a τ .

$$(2.8) \quad \frac{\rho \vdash P \xrightarrow{a} P' \quad , \quad Q \xrightarrow{\bar{a}} Q'}{\rho \vdash P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

The CSP_M rule for parallel combinator [5] is more general, in that both events may be inputs or outputs as is shown in formula 2.9 below, and the resulting event after the communication is the synchronized event (conventionally written as an output if the directions of the events were different).

$$(2.9) \quad \frac{P \xrightarrow{a} P' \quad , \quad Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'} \quad ,$$

where it is possible to see that the CSP_M rule includes 3 possible sub cases:

- Both events are outputs.
- Both events are inputs.
- One event is an output and one an input.

RSL [1]	CSP [8]
$\frac{\rho \vdash P \xrightarrow{a} P' \quad , \quad Q \xrightarrow{\bar{a}} Q'}{\rho \vdash P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$	$\frac{P \xrightarrow{c?x} P' \quad , \quad Q \xrightarrow{c!v} Q'}{P \parallel Q \xrightarrow{c!v} P_{(v)'} \parallel Q'}$

Table 2.10: RSL and CSP_M - Parallel composition operational semantics

Only the third of these corresponds to the assumptions of the RSL rule as is showed in Table 2.10.

Therefore, the following design rule is adopted:

A process can only either input or output on a channel, and at most one other process can have access to that channel, and the access must be in the opposite direction.

This means that if two processes P and Q in RSL are running concurrently, the intersection of the input channels of P and Q must be empty, and the intersection of the output channels of P and Q must also be empty. This removes from the translated CSP the operational rules that do not exist in the RSL operational semantics.

If this rule is adopted for RSL then the translation from RSL to CSP_M will happen in a natural manner.

Also, since the resulting event after the communication is not the same; to get the same behaviour in CSP_M as in RSL, we have to hide the synchronization events when we translate the parallel combinator. That is, the translation of $P \parallel Q$ becomes $(P_T \parallel Q_T) \setminus \alpha P_T \cap \alpha Q_T$, where P_T is the CSP_M process translated from the RSL process P .

4. After an RSL expression has been translated into CSP_M ; traces refinement, failures refinement, failures-divergences refinement, deadlock-freedom property, livelock-freedom property, safety properties, liveness properties and determinism can be checked in the translated CSP_M script using the FDR2 model checker tool.

Where[6]:

- A process Q is a traces refinement of another P , if all the possible sequences of communications which Q can do are also possible for P (Traces model).
- A failure is a pair (s, X) , where s is a trace of the process and X is a set of events the process can refuse to perform at that point. Failures refinement means that the set of all failures of a refining process are included in those of the refined process (Failures model).
- Failures-divergences model adds the concept of divergences to the failures model. The divergences of a process are the set of traces after which the process may livelock.
- Deadlock-freedom property uses failures model.
- Livelock-freedom property uses failures-divergences model.

- Traces refinement is used for proving safety properties.
 - Failures-divergence refinement is used for proving safety, liveness and combination properties, and also for establishing refinement and equality relations between systems.
 - Failures refinement is normally used to prove failures-divergence refinement for processes that are already known to be divergence-free.
5. Finally, it was necessary to show that the results obtained from tools applied to the CSP_M scripts are valid for the original RSL. In order to show it, two things were established: that the translation scheme is a strong bisimulation and that strong bisimulation preserves the properties mentioned before.

Structural induction was used in [8] over the syntax of RSL expressions, where for each construction of input, output, sequence, internal choice, external choice, parallelism and if ... then ... else expressions; the assumption that the component processes are strongly bisimilar with their translations was made, and then it was shown that the constructed processes are strongly bisimilar with their translations. It was also shown that strong bisimilarity preserves properties based on traces, deadlock, refusals, failures and divergences.

With these, it is possible to conclude that the RSL script and the CSP_M script obtained by translation are strongly bisimilar and that properties of CSP_M processes derived from traces, deadlock, refusals, failures and divergences also hold for the RSL processes from which the CSP_M processes were translated. So what is checked in a CSP_M script also holds for the corresponding RSL script.

2.4 Translating RSL subset to CSP_M

This section presents the translation strategy used to develop the translator tool from RSL to CSP_M .

Since we are focus on the *applicative*, *concurrent* and *model-based* specifications styles; the following syntactic and semantic *equivalences* were found.

2.4.1 Built-in types

Booleans

Boolean is a type that contains two values and its operators are mainly connectives. The syntactic equivalence between RSL and CSP_M is as follows in Table 2.11:

	RSL	CSP_M
Boolean literals	true, false	true, false
Boolean and	\wedge	\wedge
Boolean or	\vee	\vee
Boolean not	\sim	\neg
Equality operations	$=, \neq$	$==, \neq$

Table 2.11: Booleans - Syntax equivalences

Integers

The integer type contains the negative and non-negative whole numbers. The syntactic equivalence from RSL to CSP_M is as follows in Table 2.12:

	RSL	CSP_M
Integer literals	$\dots, -1, 0, 1, \dots$	$\dots, -1, 0, 1, \dots$
Sum and Difference	$+, -$	$+, -$
Unary minus	$-$	$-$
Product	$*$	$*$
Integer division and modulo	$/, \backslash$	$/, \%$
Ordering operations	$<, \leq, >, \geq$	$<, \leq, >, \geq$

Table 2.12: Integers - Syntax equivalences

Natural Numbers

The natural number, Nat type, is a subtype of the integer type. All the integer operators showed in Table 2.12 are defined for natural numbers. (Though some operators have stronger precondition).

Since in CSP_M , natural numbers are not explicitly defined; a declaration of the Nat type will be translated to the open ranged set $\{0..\}$ in CSP_M

2.4.2 Compound types

Products

A product is an ordered finite collection of values of possible different types, enclosed in parentheses and separated by commas. A product in RSL is equivalent to a tuple in CSP_M as is shown in Table 2.3 and Table 2.6.

Sets

A set is an unordered collection of distinct values of the same type. The syntactic equivalence from RSL to CSP_M is as is shown in Table 2.13:

	RSL	CSP_M
Enumerated set	$\{1,2,3\}$	$\{1, 2, 3\}$
Ranged set	$\{m..n\}$	$\{m..n\}$
Comprehended set	$\{2*n n:Index \bullet n<3\}$	$\{2 * n \mid n \leftarrow Index, n < 3\}$
Set union	\cup	\cup
Set intersection	\cap	\cap
Set difference	\setminus	$-$
Set membership test	\in	\in
Set cardinality	card	<i>card</i>

Table 2.13: Sets - Syntax equivalences

Lists

A list is a sequence of values of the same type, possibly including duplicates. A list in RSL is equivalent to a sequence in CSP_M . The syntactic equivalence from RSL to CSP_M is as is shown in Table 2.14:

Subtypes

A type T_1 is a subtype of another type T_2 if all the values contained in T_1 are also contained in T_2 . The syntax of a subtype in RSL is:

$$\{| \text{single_typing} \bullet \text{pure-restriction} |\},$$

	RSL	CSP_M
Enumerated list	$\langle 1,2,3 \rangle$	$\langle 1, 2, 3 \rangle$
Ranged list	$\langle m..n \rangle$	$\langle m..n \rangle$
Comprehended list	$\langle 2*n n \text{ in } \langle 5,6,4 \rangle \bullet n < 6 \rangle$	$\langle n \mid n \leftarrow \langle 5, 6, 4 \rangle, n < 6 \rangle$
catenation	$\hat{\quad}$	$\hat{\quad}$
length	len	#
head	hd	<i>head</i>
tail	tl	<i>tail</i>

Table 2.14: Lists - Syntax equivalences

An example of a subtype in RSL is as follows:

$$\text{Index} = \{ | i : \mathbf{Nat} \bullet i \in \{0..5\} | \},$$

The kind of subtypes that can be translated are those in which the restriction contains a set expression. The CSP_M expression generated will be a named type, as in the following example:

```
nametype Index = {0..5}
```

Variant types

Variant types allow us to define types with a choice of values; perhaps with different structure. The simplest case is like an enumeration type found in some programming languages, such as:

```
Colour == red|yellow|blue
```

Other variant types have a more rich structure, such as:

```
Figure == box(length:Index ↔ new_length, width: Index ↔ new_width)
         |circle(radius: Index)
```

Variant types can be translated to a datatype in CSP_M ; since complex variants provide an implicit constructor, destructors, and sometimes (optional) reconstructors then an equivalent to these features have to be created during the translation to CSP_M , as follows:

```
datatype Colour = red | yellow | blue
```

```
datatype Figure = box.Index.Index | circle.Index
length(box.vlength.vwidth) = vlength
width(box.vlength.vwidth) = vwidth
radius(circle.vradius) = vradius
new_length(nlength, box.vlength.vwidth) = box.nlength.vwidth
new_width(nwidth, box.vlength.vwidth) = box.vlength.nwidth
```

Here, $box.\alpha.\beta$ and $circle.\gamma$ are constructors; $length(box.\alpha.\beta)$, $width(box.\alpha.\beta)$ and $radius(circle.\gamma)$ are destructors; and $new_length(\alpha1, box.\alpha.\beta)$ and $new_width(\beta1, box.\alpha.\beta)$ are reconstructors.

Record types

A record is a collection of data and its definition in RSL is as shown in the following example:

```
Shape::
  F:Figure
  C:Colour ↔ new_Colour
```

Records in RSL also have a constructor, destructors and optional reconstructs; then the translation to CSP_M is as follows:

```
datatype Shape = mk_Shape.Figure.Colour
F(mk_Shape.vF.vC) = vF
C(mk_Shape.vF.vC) = vC
new_Colour(nC, mk_Shape.vF.vC) = mk_Shape.vF.nC
```

Here, $mk_Shape.\alpha.\beta$ is a constructor; $F(mk_Shape.\alpha.\beta)$ and $C(mk_Shape.\alpha.\beta)$ are destructors and $new_Colour(\beta1, mk_Shape.\alpha.\beta)$ is a reconstructor.

2.4.3 Constant values

A constant is a value that does not change. In RSL a constant can be defined in an implicit way such as:

floors: **Int** • floors \geq 2

Or in an explicit way such as:

floors: **Int** = 10

Only explicit definitions of constants can be translated to CSP_M . The following example shows a translation of values in RSL to values in CSP_M :

RSL values:

iTask: **Int** = 50,
 vSet: **Int-set** = {0..6} \ {0..3},
 vList: **Int*** = <0..6>

CSP_M values:

iTask = 50
vSet = {0..6} - {0..3}
vList = <0..6>

2.4.4 Functions

A function is essentially a mapping from values of one type to values of another type. We will consider that a function returns a value of a type different to Unit and that it does not access channels. In RSL a function can be defined in an implicit way such as:

square_root:**Real** $\xrightarrow{\sim}$ **Real**
 square_root(x) **as** s
post s*s = x \wedge s \geq 0.0
pre x \geq 0.0

Or in an explicit way such as:

```
increment_one: Int → Int
increment_one(i) ≡ i+1
```

Only explicit definitions of functions can be translated to CSP_M ; pre and post conditions will be ignored. The following example shows a translation to CSP_M of the above RSL function:

$$increment_one(i) = i + 1$$

If Expressions

An if expression can be used to choose between two value expressions that should have the same type according to the evaluation of a boolean condition. The structure of an if expression is as follows:

```
if boolean-value-expr then value-expr1 else value-expr2 end
```

The following example shows a translation of an if expression in RSL to an if expression in CSP_M :

RSL if expression

```
if  $x > y$  then  $x - y$  else  $y - x$  end
```

CSP_M if expression

$$\text{if } x > y \text{ then } x - y \text{ else } y - x$$

RSL also allows a short hand structure for an if expression (nested if expression), as follows:

```

if boolean-value_expr1 then value_expr1
elsif boolean-value_expr2 then value_expr2
...
elsif boolean-value_exprn then value_exprn else value_exprn+1 end

```

Since CSP_M does not support elsif expressions, they have to be translated to nested if expressions in CSP_M . The following example shows a translation of an elsif expression in RSL to a nested if expression in CSP_M :

RSL if expression:

```

invert:Colour → Colour
invert(c)≡
  if c=black then white
  elsif c=white then black
  else grey
  end,

```

CSP_M nested if expression:

$$\text{invert}(c) = \text{if } c == \text{black then white else if } c == \text{white then black else grey}$$

Let Expressions

Definitions can be made local to an expression by enclosing them in a let expression. In RSL a let expression can be defined in an implicit or an explicit way. Only the explicit definition of a let expression can be translated to a let expression in CSP_M . The following example shows a translation of a function that uses a let expression in RSL to a function that uses a let expression in CSP_M :

RSL let expression:

```

plus: Int × Int → Int
plus(p) ≡ let (x,y)=p in x+y end,

```

CSP_M let expression:

$$\text{plus}(p) = \text{let } (x, y) = p \text{ within } x + y$$

Case Expressions

A case expression allows the selection of one of several alternative expressions, as shown in the following example:

```

exclusive_or: Bool × Bool → Bool
exclusive_or(b1,b2) ≡
  case (b1,b2) of
    (true,false) → true,
    (false,true) → true,
    _ → false
  end

```

CSP_M does not support case expressions; but they can be simulated using if expressions and let expressions as follows:

```

exclusive_or(b1, b2) =
  if let (x1_, x2_) = (b1, b2) within x1_ == true and x2_ == false
  then let (x1_, x2_) = (b1, b2) within true
  else
    if let (x1_, x2_) = (b1, b2) within x1_ == false and x2_ == true
    then let (x1_, x2_) = (b1, b2) within true
    else false

```

2.4.5 Patterns

A pattern can be a literal value of one of the built-in types, a wildcard, a named value (like the variant type *red* from the example in section 2.4.2), a constructor or destructor of variants and records or a list or product pattern as shown in Table 2.15:

2.4.6 Channels

Channel definitions

Channel definitions in RSL have one or more identifiers followed by a type expression as in the following example:

	RSL	CSP_M
Value literal	true	true
	50	50
Value name	yellow	<i>yellow</i>
	iTask	<i>iTask</i>
Wildcard	-	-
Product pattern	(x,y)	(x, y)
	(true,false)	$(true, false)$
Variant and record pattern	box(3,4)	<i>box.3.4</i>
List pattern	$\langle \rangle$	$\langle \rangle$
	$\langle h \rangle^t$	$\langle h \rangle^t$

Table 2.15: Patterns - Examples of syntactic equivalences

type

Index = $\{ | i : \mathbf{Nat} \bullet i \in \{0..Count\} | \}$,
 Data == D0|D1|D2|D3

channel

mess:Index×Data,
 ack:Index

The translation into CSP_M is as follows:

nametype Index = $\{0..Count\}$
datatype Data = D0 | D1 | D2 | D3
channel mess : (Index, Data)
channel ack : Index

Channel arrays definitions

In some situations it is useful to define an arbitrary number of channels as an array. RSL allows the definition of arrays of channels through the object array definition, as shown in the following example:

type Index = $\{ | i : \mathbf{Nat} \bullet i \in \{0..iPhil\} | \}$
object
 fork[p : Index, f : Index] : **class channel** pickup, putdown : **Unit end**,
 phil[p : Index] : **class channel** sitdown, getup, think, eat : **Unit end**

The translation to CSP_M channel definitions will be as follows:

```

nametype Index = {0..iPhil}
channel fork_pickup, fork_putdown : Index.Index
channel phil_sitdown, phil_getup, phil_think, phil_eat : Index

```

2.4.7 Processes

A process in RSL can be considered as an entity capable of communicating with other processes along events and channels. An RSL process is like a function, except that it accesses channels. To be translatable to CSP_M it must have **Unit** as the result type. The following example shows the translation of a process in RSL to a process in CSP_M .

RSL process:

```

DOORKEEPER: Unit → out open, close Unit
DOORKEEPER() ≡ open!(); close!(); DOORKEEPER()

```

CSP_M process:

```

DOORKEEPER = open → close → DOORKEEPER

```

Table 2.16 shows the equivalences of process terminators from RSL to CSP_M .

RSL	CSP_M
stop	stop
skip	skip
chaos	chaos (α)

Table 2.16: Process terminators - translation equivalences

A process can also have (non-Unit) parameters as we will see below and **stop**, **skip** and **chaos** are terminators of processes (Table 2.16).

Events, channels and the sequencing operator

Since a process interacts with other processes only through its interface, the important information in the description of a process concerns its behaviour on that interface. The interface of a process will be described as a set of events. Simple events are considered to be atomic and indivisible in their occurrence [6]. In RSL a simple event can be defined as a Unit channel. In the previous example, *open* and *close* are simple events.

However, some events that contain several pieces of information are necessary. So events can have some structure. One instance of a structured event is given by a communication channel which carries messages. If c is a channel name of type T , and v is a particular value of type T , then the RSL output expression $c!v; P()$ describes a process which is initially willing to output v along channel c , and subsequently behave as $P()$. If processes $P(x)$ are defined for each x that belongs to T then the RSL input expression **let** $x = c? \mathbf{in} P(x)$ **end** describes a process which is initially ready to accept any value x of type T along channel c [6]. The following example shows a process that has communication channels

```
COPY: Unit  $\rightarrow$  in input out output Unit
COPY()  $\equiv$  let  $x = \text{input?}$  in output! $x$ ; COPY() end
```

The translation of this example into CSP_M is as follows:

$$COPY = \text{input?}x \rightarrow \text{output!}x \rightarrow COPY$$

Another instance of a structured event is given by channel arrays in RSL. The following example shows a process that has channel arrays:

```
DOORKEEPER: Unit out  $\{\{\text{door}[i].\text{open}, \text{door}[i].\text{close}\} | i:\text{Index}\}$  Unit
DOORKEEPER( $i$ )  $\equiv$  door[ $i$ ].open!(); door[ $i$ ].close!(); DOORKEEPER( $i$ )
```

The translation of this example into CSP_M is:

$$DOORKEEPER(i) = \text{door_open.}i \rightarrow \text{door_close.}i \rightarrow DOORKEEPER(i)$$

To compose events and channels in order to evaluate them one after the other (in order); RSL uses the sequencing operator (;) or a let expression. The main patterns of a sequencing

RSL	CSP_M
$c!(); P()$	$c \rightarrow P$
$c?; P()$	$c \rightarrow P$
$c!v; P()$	$c!v \rightarrow P$
let $x = c? \mathbf{in} P(x) \mathbf{end}$	$c?x \rightarrow P(x)$
$c[i].ch!(); P()$	$c_ch.i \rightarrow P$
$c[i].ch?; P()$	$c_ch.i \rightarrow P$
$c[i,j].ch!v; P()$	$c_ch.i.j!v \rightarrow P$
let $x = c[i,j].ch? \mathbf{in} P(x) \mathbf{end}$	$c_ch.i.j?x \rightarrow P(x)$

Table 2.17: Sequencing operator - translation equivalences

operator and let expression using channels, and their respective equivalences in CSP_M are shown in Table 2.17.

A sequencing operator in RSL is equivalent to a prefix operator in CSP_M ; only when the expression on the left side is an input or output channel.

If process expressions

As in RSL functions; an if process expression can be used to choose between two processes according to the evaluation of a boolean condition. The following example shows a translation of an if process expression in RSL to an if process expression in CSP_M :

RSL if process expression

$$\text{TASK}(i) \equiv$$

$$\mathbf{if} \ i=0 \ \mathbf{then} \ \text{START_TASK}(i) \ \mathbf{else} \ \text{WAIT_TASK}(i) \ \mathbf{end}$$

CSP_M if process expression

$$\text{TASK}(i) =$$

$$\mathbf{if} \ i == 0 \ \mathbf{then} \ \text{START_TASK}(i) \ \mathbf{else} \ \text{WAIT_TASK}(i)$$

Let process expressions

A let expression inside a process is not just used with input channels; it can also be used to do local definitions. Only explicit let process expressions can be translated to CSP_M . The following example shows a translation of a process that uses a let expression in RSL to a process that uses a let expression in CSP_M :

RSL let process expression:

$$\text{PROC_PLUS}() \equiv \text{let } p = \text{input? in} \\ \quad \text{let } (x,y)=p \text{ in output!}x+y; \text{PROC_PLUS}() \text{ end} \\ \text{end}$$

CSP_M let process expression:

$$\text{PROC_PLUS} = \text{input?}p \rightarrow \text{let } (x, y) = p \text{ within } \text{output!}x + y \rightarrow \text{PROC_PLUS}$$

Case process expressions

A case process expression allows the selection of one of several alternative processes. CSP_M does not support case process expressions; but they can be simulated using if expressions and let expressions as shown in the following example:

RSL case process expression:

$$\text{BUTLER}(p) \equiv \\ \text{case } p \text{ of} \\ \quad 0 \rightarrow \{ \text{phil}[i].\text{sitdown?}; \text{BUTLER}(1) \mid i:\text{Index} \}, \\ \quad i\text{Phil} \rightarrow \{ \text{phil}[i].\text{getup?}; \text{BUTLER}(i\text{Phil} - 1) \mid i:\text{Index} \}, \\ \quad - \rightarrow \\ \quad \quad \{ \text{phil}[i].\text{sitdown?} ; \text{BUTLER}(p + 1) \mid i:\text{Index} \} \\ \quad \quad \{ \text{phil}[i].\text{getup?} ; \text{BUTLER}(p - 1) \mid i:\text{Index} \} \\ \text{end,}$$

CSP_M equivalent case process expression:

$$\begin{aligned}
BUTLER(p) = & \\
& \text{if } p == 0 \\
& \text{then } \square i : Index \bullet phil_sitdown.i \rightarrow BUTLER(1) \\
& \text{else} \\
& \quad \text{if } p == iPhil \\
& \quad \text{then } \square i : Index \bullet phil_getup.i \rightarrow BUTLER(iPhil - 1) \\
& \quad \text{else } \square i : Index \bullet phil_sitdown.i \rightarrow BUTLER(p + 1) \\
& \quad \quad \square \\
& \quad \quad \square i : Index \bullet phil_getup.i \rightarrow BUTLER(p - 1)
\end{aligned}$$

From channel accesses to Alphabets

In CSP_M , the alphabet of a process is the set of all events that the process uses. A CSP_M process is completely described by the way it can communicate with its external environment, so alphabets are very important for parallel composition.

A RSL process does not have intrinsic alphabets, but it can be calculated. When the process does not contain parallel composition; the alphabet can be calculated in two ways:

- Using the signature of a process.
- Using the access declaration.

In order to avoid the (mutual) recursion treatment, the second alternative was chosen, so the RSL keyword **any** used in access has to be avoided in order to make the calculation of an alphabet in a natural way.

Since our design rule states that RSL process can only either input or output on a channel, and at most one other process can have access to that channel in the opposite direction, it is necessary, in order to check the rule, to calculate an input alphabet (that contains all the input events and channels) and an output alphabet (that contains all the output events and channels).

The translation of a process from RSL to CSP_M and the calculation of its alphabets is shown in the following example:

RSL process:

CAR: **Unit** \rightarrow **out** car_run,car_approach, car_enter

$$\begin{aligned} & \mathbf{out} \text{ car_leave, crash, crash2 } \mathbf{Unit} \\ \text{CAR}() \equiv & \\ & \text{car_run!();car_approach!();car_enter!();} \\ & (\text{car_leave!();CAR}()) \\ & \square \\ & \text{crash!();crash2!();stop} \end{aligned}$$

CSP_M equivalent process and alphabet:

$$\begin{aligned} \text{Alph_in_CAR} &= \{\} \\ \text{Alph_out_CAR} &= \{\text{car_enter, car_approach, car_run, crash2, crash, car_leave}\} \\ \text{CAR} &= \text{car_run} \rightarrow \text{car_approach} \rightarrow \text{car_enter} \rightarrow \\ & (\text{car_leave} \rightarrow \text{CAR} \square \text{crash} \rightarrow \text{crash2} \rightarrow \mathbf{stop}) \end{aligned}$$

In the case of if process expressions and case process expressions, the alphabet is the union of all the alphabets of the processes inside those expressions.

The calculation of alphabets for parallel composition will be explained in following subsections.

External and internal choice operators

As is shown in Table 2.18 external and internal choice in RSL and CSP_M are equivalent. The following example shows a translation of processes with internal and external choice.

RSL processes

$$\begin{aligned} \text{VMCT: } & \mathbf{Unit} \rightarrow \mathbf{in} \text{ coin } \mathbf{out} \text{ choc, toff } \mathbf{Unit} \\ \text{VMCT}() \equiv & \text{coin?};(\text{choc!();VMCT}()) \square \text{toff!();VMCT}(), \end{aligned}$$

$$\begin{aligned} \text{CSTM: } & \mathbf{Unit} \rightarrow \mathbf{out} \text{ coin } \mathbf{in} \text{ choc, toff } \mathbf{Unit} \\ \text{CSTM}() \equiv & \text{coin!();}(\text{choc?;CSTM}()) \square \text{toff?;CSTM}(), \end{aligned}$$

CSP_M processes

$$\begin{aligned} \text{Alph_in_VMCT} &= \{\text{coin}\} \\ \text{Alph_out_VMCT} &= \{\text{toff, choc}\} \end{aligned}$$

$$VMCT = coin \rightarrow (choc \rightarrow VMCT \sqcap toff \rightarrow VMCT)$$

$$Alph_in_CSTM = \{toff, choc\}$$

$$Alph_out_CSTM = \{coin\}$$

$$CSTM = coin \rightarrow (choc \rightarrow CSTM \sqcap toff \rightarrow CSTM)$$

	RSL	CSP_M
External choice	$P() \sqcap Q()$	$P \sqcap Q$
Internal choice	$P() \sqcap Q()$	$P \sqcap Q$
Comprehended external choice	$\sqcap \{ P(i) \mid i:T \}$	$\sqcap i : T \bullet P(i)$
Comprehended internal choice	$\sqcap \{ P(i) \mid i:T \}$	$\sqcap i : T \bullet P(i)$

Table 2.18: Process combinators to be translated

Comprehended external and internal choice operators

As is shown in Table 2.18 comprehended external and internal choice in RSL and CSP_M are also equivalent.

The following example shows the translation of a process with comprehended external choice.

RSL process

$$\begin{aligned} \text{SndMess: } & \mathbf{Unit} \rightarrow \mathbf{in} \{ \text{snd_mess}[i].c \mid i: \text{Index} \} \mathbf{out} \text{ mess } \mathbf{Unit} \\ \text{SndMess()} & \equiv \\ & \sqcap \{ \mathbf{let} \ x = \text{snd_mess}[i].c? \ \mathbf{in} \ \text{mess}!(i,x); \text{SndMess}() \mathbf{end} \mid i : \text{Index} \} \end{aligned}$$

CSP_M process

$$\begin{aligned} Alph_in_SndMess & = \{ \text{snd_mess_c}.i \mid i \leftarrow \text{Index} \} \\ Alph_out_SndMess & = \{ \text{mess} \} \\ SndMess & = \sqcap i : \text{Index} \bullet \text{snd_mess_c}.i?x \rightarrow \text{mess}!(i, x) \rightarrow SndMess \end{aligned}$$

Parallel operator

Since parallel operator in RSL and CSP_M are not equivalent, and in order to preserve what we found in Section 2.3, the parallel operator in RSL:

$$PQ() \equiv P() \parallel Q()$$

is equivalent to the following process expression in CSP_M :

$$PQ = \begin{cases} \text{if } (in_{\alpha}P \cap in_{\alpha}Q == \{\} \wedge out_{\alpha}P \cap out_{\alpha}Q == \{\}) \text{ then} \\ \quad (P \parallel [in_{\alpha}P \cup out_{\alpha}P \mid in_{\alpha}Q \cup out_{\alpha}Q] \parallel Q) \setminus (in_{\alpha}P \cap out_{\alpha}Q \cup out_{\alpha}P \cap in_{\alpha}Q) \\ \text{else} \\ \quad \mathbf{chaos}(in_{\alpha}P \cup out_{\alpha}P \cup out_{\alpha}Q \cup in_{\alpha}Q) \end{cases}$$

Here, $in_{\alpha}X$ and $out_{\alpha}X$ are the input and output alphabets of the process X .

The input and output alphabets in CSP_M of the process PQ are as follows:

$$\begin{aligned} in_{\alpha}PQ &= (in_{\alpha}P \cup in_{\alpha}Q) - (in_{\alpha}P \cap out_{\alpha}Q \cup out_{\alpha}P \cap in_{\alpha}Q) \\ out_{\alpha}PQ &= (out_{\alpha}P \cup out_{\alpha}Q) - (in_{\alpha}P \cap out_{\alpha}Q \cup out_{\alpha}P \cap in_{\alpha}Q) \end{aligned}$$

Comprehended parallel operator

The comprehended parallel operator in RSL:

$$CP() \equiv \parallel \{ P(i) \mid i : T \}$$

is translated to the following process expression in CSP_M :

$$CP = \left(\parallel i : T \bullet in_{\alpha}P(i) \cup out_{\alpha}P(i) \circ P(i) \right) \setminus \left(\bigcup \{ in_{\alpha}P(i) \mid i \leftarrow T \} \cap \bigcup \{ out_{\alpha}P(i) \mid i \leftarrow T \} \right)$$

Here, $in_{-\alpha}X(i)$ and $out_{-\alpha}X(i)$ are the input and output alphabet of the process $X(i)$.

This rule is only sound if the input and output alphabets in CSP_M are pairwise disjoint, i.e. if

$$\forall i, j \leftarrow T, i \neq j \Rightarrow in_{-\alpha}P(i) \cap in_{-\alpha}P(j) = \{\} \wedge out_{-\alpha}P(i) \cap out_{-\alpha}P(j) = \{\}$$

The translator tool does not check that this rule is met which means that the user must check it manually.

The input and output alphabets in CSP_M of the process CP are as follows:

$$\begin{aligned} in_{-\alpha}CP &= \bigcup\{in_{-\alpha}P(i) \mid i \leftarrow T\} - (\bigcup\{in_{-\alpha}P(i) \mid i \leftarrow T\} \cap \bigcup\{out_{-\alpha}P(i) \mid i \leftarrow T\}) \\ out_{-\alpha}CP &= \bigcup\{out_{-\alpha}P(i) \mid i \leftarrow T\} - (\bigcup\{in_{-\alpha}P(i) \mid i \leftarrow T\} \cap \bigcup\{out_{-\alpha}P(i) \mid i \leftarrow T\}) \end{aligned}$$

Chapter 3

CSP_M translator tool

This chapter is focused on reporting the construction of the translation tool from RSL to CSP_M .

3.1 Implementation

The translator tool from RSL to CSP_M was constructed using the GENTLE Compiler Construction System [2].

A compiler is a computer program that translates text written in one language (the source language) into another language (the target language).

GENTLE Compiler Construction System is widely used in industry and research. It has been applied in large projects and for constructing various commercial products. GENTLE covers the full spectrum of translation. It supports language recognition, definition of abstract syntax trees, construction of tree walkers based on pattern matching, smart traversal, simple unparsing for source-to-source translation, and optimal code selection for microprocessors. Gentle provides a uniform framework for all of these tasks and it is freely available for open source projects, personal usage, and education[2].

There are commonly three general phases in the translation of a context free language into another with syntax and semantic differences. The first phase is called the parsing phase: the source language is parsed to check against the syntax and after applying semantic rules an abstract syntax tree is built. The second phase is called the translation phase: some semantic rules are applied in order to translate from one abstract syntax tree to another abstract syntax tree. The third phase is called the unparsing or generating phase: an output

is generated from the new syntax tree.

Our starting point on the implementation of the tool was the AST (abstract syntax tree) of RSL that was already implemented in GENTLE inside the RSL typechecker tool. That means that the parsing phase was already done and what we implemented was the translation and unparsing phases.

For the implementation of the translation phase; an AST of the subset of CSP_M to be translated was built and implemented in GENTLE, then the translation strategy designed and reported on chapter 2 and the unparsing was implemented using GENTLE. The result is a tool that takes as an input an RSL script (in a file); this script is transformed in an AST of RSL by the type checker tool, then applying many translation rules the AST of RSL is transformed in an AST of CSP_M and finally this AST is unparsed in a new output script (a new file) in CSP_M by our tool.

The generated script (with `.fdr2` extension) is ready for being uploaded into the FDR2 model checker tool for the verification of properties.

3.2 Structure of the tool

The general structure of the tool is as follows:

- `fdr.g` is the main module that calls the others.
- `fdr_ast.g` has the definition of the AST of CSP_M subset of translation.
- `fdr_gen_ast.g` does the translation of the AST of RSL to the AST of CSP_M .
- `fdr_gen_code.g` generates the CSP_M script from the AST of CSP_M .

3.3 Limitations of the tool

This first academic version of the translation tool from RSL to CSP_M has some limitation that were not taken into account during the development of tool; these limitation are:

- The translator tool only can translate specifications inside one module.
- The overloading of identifiers and operators will generate runtime errors
- Recursive variants will generate runtime errors

- Infinite types may generate runtime errors
- A process in RSL can be translated to a process in CSP_M only if it has Unit as the result type.
- The pairwise disjointness of the alphabets during the translation of the comprehended parallel operator must be checked manually by the user.

Chapter 4

Summary

In this report we have explained the syntactic and semantic differences between a formal language like RSL and a refinement checking language like CSP_M .

We have also covered the main problems faced during the translation process of these two languages, and when possible, we describe the translation technique applied in those cases with a justification of how the RSL semantics are preserved during these translation.

We have also constructed a tool that implements the translation of nearly all inside the applicative, concurrent and model-based specification styles of RSL. The implementation of this tool, has served itself as a verification of the suitability of the approach mentioned in this report and in the report of Abigail Parisaca [8], and now can be used to model check specifications inside the applicative, concurrent and model-based specification styles.

References

- [1] M. Debabi. The RSL semantic course, 1993.
- [2] The GENTLE Compiler Construction System. <http://gentle.compilertools.net/index.html>.
- [3] Chris W. George. Applicative Modelling with RAISE, Tutorial. In Zhiming Liu Chris W. George and Jim Woodcock, editors, *Domain Modeling and the Duration Calculus*, pages 51–118. UNU-IIST, Springer-Verlag LNCS, 2007.
- [4] The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1992.
- [5] C. A. R. Hoare. *Communicating Sequential Processes*. 1985.
- [6] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*. 2005.
- [7] LFM: Langley Formal Methods (NASA). <http://shemesh.larc.nasa.gov/fm/index.html>.
- [8] Abigail Parisaca Vargas and Chris George. Towards the translation from RSL to CSP, 2008. UNU-IIST research report 395.
- [9] The RAISE Method Group. *The RAISE Development Method*. Prentice-Hall International, 1995.
- [10] A.W. Roscoe. *The Theory and Practice of Concurrency*. 2005.
- [11] TERMA. RAISE - Rigorous Approach to Industrial Software Engineering. <http://spd-web.terma.com/Projects/RAISE/index.html>.

Appendix A

Examples of translation from RSL to CSP_M

A.1 The Cyclic Scheduler Example

A.1.1 The problem

Suppose there are six processes which need to be controlled. Each process can be started by a *start* event and uses a *finish* event to indicate that it has finished. Suppose also that the processes should start in order; returning to the first when the last has been started; when a process has finished it can be started again, but only when its turn comes.

A.1.2 The model

It is possible to model a scheduler, which uses *start* and *finish* events to control the processes. This scheduler will be implemented as a collection of *tasks*, each of which communicate with one of the processes being controlled and also with the next task.

For convenience during the modelling process:

- The relative position of the tasks will be as in Figure A.1
- The tasks will be labelled from 0 to 5: a suitable type is represented in RSL as follows:

value

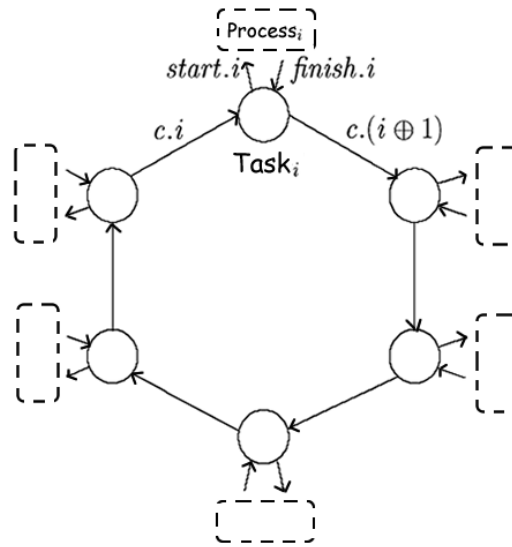


Figure A.1: The Cyclic Scheduler

```

iTask : Nat=5
type
Index = { | i : Nat • i ∈ {0..iTask} | }

```

The translation to CSP_M is:

```

iTask = 5
Index = {0 .. iTask}

```

- The value $inc(i)$ is *addition of 1 modulo 6*, written in RSL as follow:

```

inc : Index → Index
inc(i) ≡ (i + 1) \ (iTask+1),

```

The translation to CSP_M is:

```

inc(i) = (i + 1) mod (iTask + 1)

```

The idea is that each task waits for a signal in a *channel* c to its left, which means that the process to its left has been started. Then the task starts its own process and sends a signal

on the *channel* c to its right, to tell the next task that it can start its process. Also each task has to wait for its process to finish before starting it again.

Since this model is an example of a concurrent system; we will use channel arrays to represent the events that are illustrated in Figure A.1.

Since it is not necessary to pass a value in these channels, they will be Unit type (simple events).

object

```
start[t : Index] :class channel c : Unit end ,
finish[t : Index] :class channel c : Unit end,
c[t : Index] :class channel c : Unit end
```

The translation of the channel array declaration is as follows:

```
channel start : Index
channel finish : Index
channel c : Index
```

In order to start everything off one of the tasks must begin by starting its process instead of waiting.

```
START_TASK : Index → out {{start[i].c,c[inc(i)].c}|i:Index}
                    in {{finish[i].c,c[i].c}|i:Index} Unit
```

```
START_TASK(i) ≡
  start[i].c!() ;
  c[inc(i)].c!();
  (finish[i].c?;c[i].c?;START_TASK(i)
   □
   c[i].c?;finish[i].c?;START_TASK(i)),
```

The translation of this process is as follows:

$$\begin{aligned}
 \text{Alph}_{in_START_TASK}(i) &= \{ \text{finish}_{c.i}, c_{c.i} \} \\
 \text{Alph}_{out_START_TASK}(i) &= \{ \text{start}_{c.i}, c_{c.inc}(i) \} \\
 \text{START_TASK}(i) &= \text{start}_{c.i} \rightarrow c_{c.inc}(i) \rightarrow \\
 &\quad (\text{finish}_{c.i} \rightarrow c_{c.i} \rightarrow \text{START_TASK}(i)) \\
 &\quad \square c_{c.i} \rightarrow \text{finish}_{c.i} \rightarrow \text{START_TASK}(i)
 \end{aligned}$$

The other processes wait for the event $c[i].c?$:

$$\begin{aligned} \text{WAIT_TASK: Index} &\rightarrow \mathbf{out} \{ \{ \text{start}[i].c, c[\text{inc}(i)].c \} | i: \text{Index} \} \\ &\quad \mathbf{in} \{ \{ \text{finish}[i].c, c[i].c \} | i: \text{Index} \} \mathbf{Unit} \\ \text{WAIT_TASK}(i) &\equiv \\ &\quad c[i].c?; \text{START_TASK}(i), \end{aligned}$$

The translation of this process is as follows:

$$\begin{aligned} \text{Alph_in_WAIT_TASK}(i) &= \{ \{ \text{finish_c.i}, c_c.i \} \} \\ \text{Alph_out_WAIT_TASK}(i) &= \{ \{ \text{start_c.i}, c_c.\text{inc}(i) \} \} \\ \text{WAIT_TASK}(i) &= c_c.i \rightarrow \text{START_TASK}(i) \end{aligned}$$

Then the process TASK is defined as follows:

$$\begin{aligned} \text{TASK: Index} &\rightarrow \mathbf{out} \{ \{ \text{start}[i].c, c[\text{inc}(i)].c \} | i: \text{Index} \} \\ &\quad \mathbf{in} \{ \{ \text{finish}[i].c, c[i].c \} | i: \text{Index} \} \mathbf{Unit} \\ \text{TASK}(i) &\equiv \\ &\quad \mathbf{if } i=0 \mathbf{ then } \text{START_TASK}(i) \mathbf{ else } \text{WAIT_TASK}(i) \mathbf{ end}, \end{aligned}$$

The translation of the process TASK is as follows:

$$\begin{aligned} \text{Alph_in_TASK}(i) &= \{ \{ \text{finish_c.i}, c_c.i \} \} \\ \text{Alph_out_TASK}(i) &= \{ \{ \text{start_c.i}, c_c.\text{inc}(i) \} \} \\ \text{TASK}(i) &= \mathbf{if } i == 0 \mathbf{ then } \text{START_TASK}(i) \mathbf{ else } \text{WAIT_TASK}(i) \end{aligned}$$

Finally the process SCHED is defined by putting all the tasks in parallel

$$\begin{aligned} \text{SCHED : Unit} &\rightarrow \mathbf{out} \{ \{ \text{start}[i].c, c[\text{inc}(i)].c \} | i: \text{Index} \} \\ &\quad \mathbf{in} \{ \{ \text{finish}[i].c, c[i].c \} | i: \text{Index} \} \mathbf{Unit} \\ \text{SCHED}() &\equiv \parallel \{ \text{TASK}(i) \mid i: \text{Index} \}, \end{aligned}$$

The translation of the process SCHED is as follows:

$$\begin{aligned}
Alph_in_SCHED &= \bigcup(\{Alph_in_TASK(i) \mid i \leftarrow Index\}) - \\
&\quad \bigcup(\{Alph_in_TASK(i) \mid i \leftarrow Index\}) \cap \bigcup(\{Alph_out_TASK(i) \mid i \leftarrow Index\}) \\
Alph_out_SCHED &= \bigcup(\{Alph_out_TASK(i) \mid i < -Index\}) - \\
&\quad \bigcup(\{Alph_in_TASK(i) \mid i \leftarrow Index\}) \cap \bigcup(\{Alph_out_TASK(i) \mid i \leftarrow Index\}) \\
SCHED &= \text{if } \bigcap(\{Alph_in_TASK(i) \mid i \leftarrow Index\}) == \{\} \wedge \\
&\quad \bigcap(\{Alph_out_TASK(i) \mid i \leftarrow Index\}) == \{\} \text{ then} \\
&\quad \parallel i : Index \bullet Alph_in_TASK(i) \cup Alph_out_TASK(i) \circ TASK(i) \\
&\quad \setminus \bigcup(\{Alph_in_TASK(i) \mid i \leftarrow Index\}) \cup \bigcup(\{Alph_out_TASK(i) \mid i \leftarrow Index\}) \text{ else} \\
&\quad \mathbf{chaos}(\bigcup(\{Alph_in_TASK(i) \mid i \leftarrow Index\}) \cup \bigcup(\{Alph_out_TASK(i) \mid i \leftarrow Index\}))
\end{aligned}$$

Appendix B

Examples of application of model checking with RSL using FDR2

B.1 The Dining Philosophers Example

B.1.1 The problem

A *college* consists of *five philosophers* who think and eat. They *eat* at a circular dining table with five chairs. In the middle of the table there is a large tangled bowl of spaghetti, the table is set with five plates; there are also *five forks*, one between each pair of plates. When one of the *philosopher* need to *eat*, he enters to the dining hall, *sits down* on his chair *picks up* the *forks* on either side of his plate (first the one on the left, then the one on the right), *eats*, *puts down* the *forks*, *gets up* from his chair and leaves the dining hall (Since two forks are needed to eat the spaghetti; if one of the forks is already in use he has to wait).

B.1.2 The model

It is possible to model this story in various ways by choosing different episodes of philosophers lives as events, but the essential things from the point of view of interaction are when they *pick up* or *put down* their forks.

For convenience during the modelling process:

- The philosophers are labelled from 0 to 4.

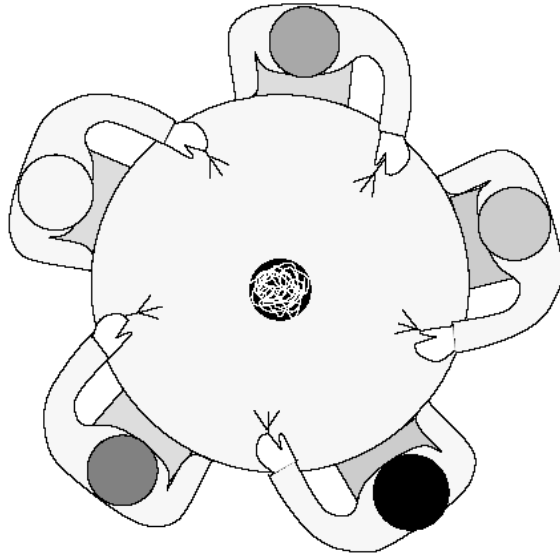


Figure B.1: The Dining Philosophers

- Each philosopher picks up two forks, also labelled from 0 to 4. It is possible to represent it in RSL as follow:

```

value
  iPhil : Nat=4
type
  Index = { | i : Nat • i ∈ {0..iPhil} | }

```

- The value $\text{inc}(i)$ is *addition of 1 modulo 5*, and $\text{dec}(i)$ is *subtraction of 1 modulo 5*; written in RSL as follow:

```

inc : Index → Index
inc(i) ≡ (i + 1) \ (iPhil+1),

dec : Index → Index
dec(i) ≡ (i + iPhil) \ (iPhil+1),

```

- Their relative positions are illustrated in Figure B.2.

The relevant components in this model are the *five philosophers*, who will be modelled as processes, and in order to make sure no fork can be held by two philosophers at once, a process is required to represent each *fork*.

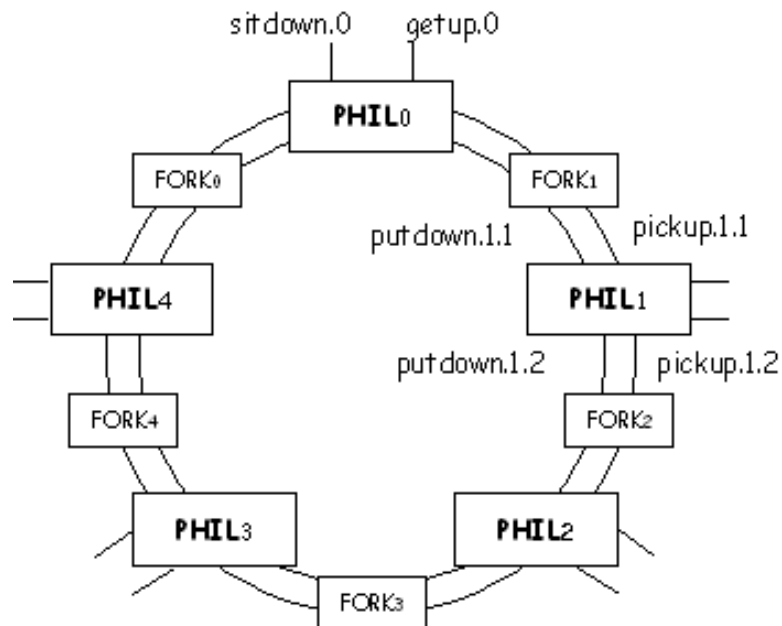


Figure B.2: The Dining Philosophers-relative positions

As in the previous example the Dining Philosophers is a concurrent system example, Then we can use the same scheme of Unit channels to model the relevant events of the *philosophers* and the *forks*

```
scheme CHANNEL = class channel c : Unit end
```

These relevant events can be represented with the following channel arrays

```
pickup[p : Index, f : Index] : CHANNEL,
putdown[p : Index, f : Index] : CHANNEL,
sitdown[p : Index] : CHANNEL,
getup[p : Index] : CHANNEL,
eat[p : Index] : CHANNEL
```

Each process Phil(i) can be model as a sequence of seven events

```
phil : Index → in any out any Unit
```

$$\begin{aligned} \text{phil}(i) \equiv & \\ & \text{sitdown}[i].c!() ; \text{pickup}[i, \text{inc}(i)].c!() ; \\ & \text{pickup}[i, i].c!() ; \text{eat}[i].c!() ; \\ & \text{putdown}[i, \text{inc}(i)].c!() ; \text{putdown}[i, i].c!() ; \\ & \text{getup}[i].c!() ; \text{phil}(i), \end{aligned}$$

Each process of the forks can be repeatedly modelled as a sequence of *pick up* and *put down* events, but there is a choice of who (philosopher) does these events

$$\begin{aligned} \text{fork} : \text{Index} &\rightarrow \text{in any out any Unit} \\ \text{fork}(i) \equiv & \\ & \text{pickup}[i, i].c? ; \text{putdown}[i, i].c? ; \text{fork}(i) \\ & \square \\ & \text{pickup}[\text{dec}(i), i].c? ; \text{putdown}[\text{dec}(i), i].c? ; \\ & \text{fork}(i), \end{aligned}$$

Now the possible behaviour of the *college* can be modelled by putting in parallel the *five philosophers* and the *five forks*.

$$\begin{aligned} \text{phils} : \text{Unit} &\rightarrow \text{in any out any Unit} \\ \text{phils}() \equiv & \parallel \{ \text{phil}(i) \mid i : \text{Index} \}, \\ \\ \text{forks} : \text{Unit} &\rightarrow \text{in any out any Unit} \\ \text{forks}() \equiv & \parallel \{ \text{fork}(i) \mid i : \text{Index} \}, \\ \\ \text{college} : \text{Unit} &\rightarrow \text{in any out any Unit} \\ \text{college}() \equiv & \text{phils}() \parallel \text{forks}(), \end{aligned}$$

Suppose that all philosophers sit down in order and the each one picks up the fork of his left, then is not possible to do a next event, because of that, this college is not *free from deadlock*.

In order to remove the possibility of deadlock, is possible to apply different strategies in this model. In this report we use the strategy of adding a *controller* (another process) in parallel that allows at most four philosophers to sit at the same time in the dining table

$$\begin{aligned} \text{butler} : \text{Index} &\rightarrow \text{in any out any Unit} \\ \text{butler}(i) \equiv & \\ & \text{case } i \text{ of} \\ & \quad 0 \rightarrow \square \{ \text{sitdown}[p].c? \mid p : \text{Index} \}; \text{butler}(1), \\ & \quad i\text{Phil} \rightarrow \square \{ \text{getup}[p].c? \mid p : \text{Index} \}; \text{butler}(i\text{Phil} - 1), \end{aligned}$$

```

- →
  [] {sitdown[p].c? |p:Index} ; butler(i + 1)
  []
  [] {getup[p].c? |p:Index} ; butler(i - 1)
end ,

```

```

newcollege : Unit → in any out any Unit
newcollege() ≡ college() || butler(0)

```

B.1.3 Property verification with model checking and FDR2

Is possible to verify with the FDR2 model checker tool that the process *college* is not *free from deadlock*, as we can see in Figure B.3.

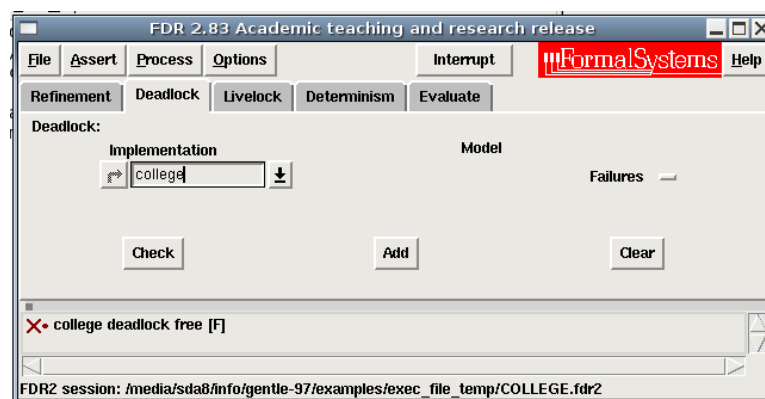


Figure B.3: FDR2 with the dining philosophers' college

As we mentioned before the process *college* is not free from deadlock because there is a trace that describe the situation when all philosophers sit down and the each one picks up the fork of his left; this situation can also be observed using FDR2 as is shown in Figure B.4

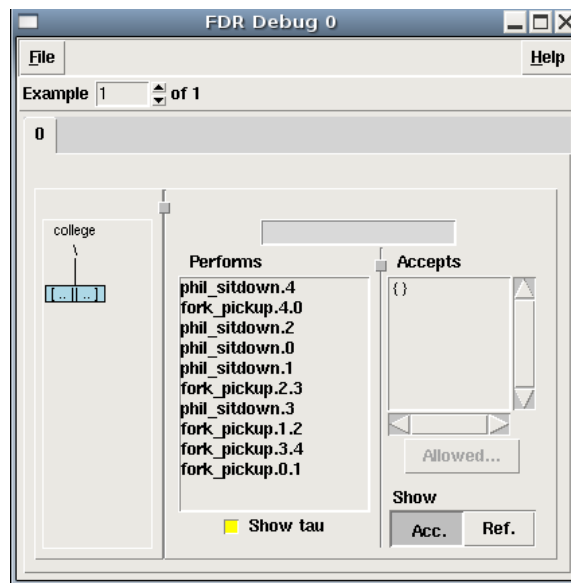


Figure B.4: Trace of the dining philosophers' college - deadlock

Finally it is possible to verify with the FDR2 model checker tool that the process *newcollege* is free from deadlock, as is shown in Figure B.5.

B.2 The Multiplexed buffer example

B.2.1 The problem

Consider the problem of *transmitting a number of message streams* over a *single data connection*. Additionally it is wished to *synchronize* the sending and receiving processes. The typical specifications of such a system might include the need to ensure that one line does not interfere with another, also the connection between each sender and the corresponding receiver acts as if it were a *simple single place buffer* like in the Figure B.6. Then the combination of N channels behaves like the *unsynchronized parallel composition of N simple buffers*.

B.2.2 The model

The requirement on the *multiplexed system* is that it refines the combination of N place buffer process. To avoid the introduction of extra buffering and interaction where one channel clogs the system when its receiver does not pick up messages soon enough, we introduce

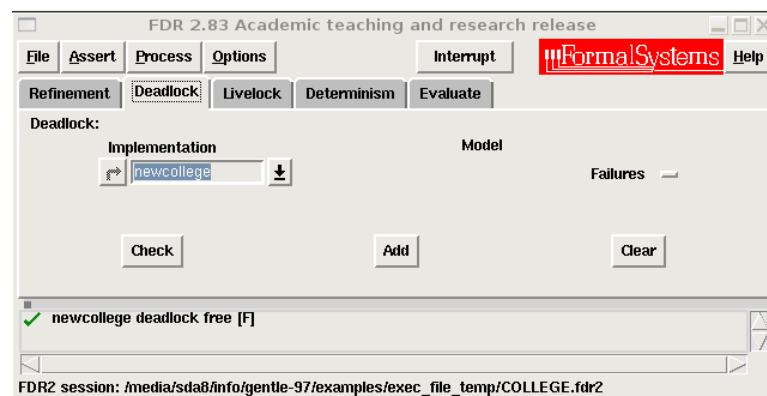


Figure B.5: FDR2 with the dining philosophers' new College

acknowledgement signals from the receivers to the senders. These can also be multiplexed through a single channel, as shown in Figure B.7.

The implementation will therefore consist of N transmitters $Tx(i)$, N receivers $Rx(i)$ and four processes which manage the forward and reverse channels: *SndMess* (*Send Message*) which multiplexes transmitted data and *RcvMess* (*Receive Message*) which demultiplexes it, together with *SndAck* (*Send Acknowledge*) and *RcvAck* (*Receive Acknowledge*) which perform similar functions for the acknowledgement channel.

For convenience during the modelling process:

- The buffers are labelled from 0 to N and the data is modelled in an explicit way, we can represent it in RSL as follows:

```

type Index = { | i : Nat • i ∈ {0..Count} | },
          Data == D0|D1|D2|D3
value
          Count : Nat = 2,

```

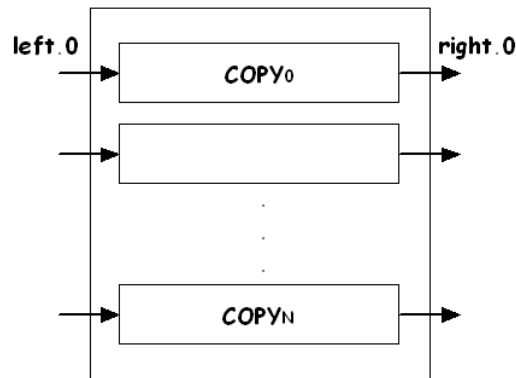


Figure B.6: parallel composition of N simple buffers

- The message will be a *Cartesian Product* of type Index and Data, and the acknowledgement will be just an Index value.

channel

```

mess:Index×Data,
ack:Index

```

- The channel arrays of communication will be represented as follows:

object

```

left[c : Index] : class channel c : Data end,
right[c : Index] : class channel c : Data end,
snd_mess[c : Index] : class channel c : Data end,
rcv_mess[c : Index] : class channel c : Data end,
snd_ack[c : Index] : class channel c : Unit end,
rcv_ack[c : Index] : class channel c : Unit end

```

Then we can express the four processes *SndMess*, *RcvMess*, *SndAck* and *RcvAck* in RSL as follows:

```

SndMess: Unit → in {snd_mess[i].c|i: Index} out mess Unit
SndMess() ≡

```

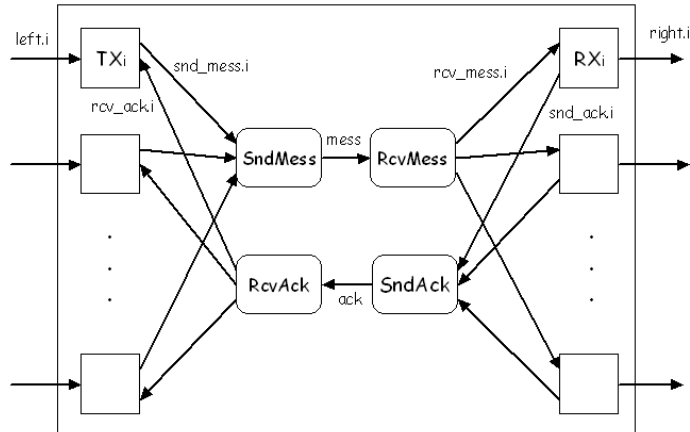


Figure B.7: The Multiplexed buffer

$$\square \{ \text{let } x = \text{snd_mess}[i].c? \text{ in } \text{mess}!(i,x); \text{SndMess}() \text{end} \mid i : \text{Index} \},$$

$$\text{RcvMess: Unit} \rightarrow \text{in } \text{mess} \text{ out } \{ \text{rcv_mess}[i].c \mid i : \text{Index} \} \text{Unit}$$

$$\text{RcvMess}() \equiv$$

$$\text{let } (i,x) = \text{mess}? \text{ in } \text{rcv_mess}[i].c!x; \text{RcvMess}() \text{end},$$

$$\text{SndAck: Unit} \rightarrow \text{in } \{ \text{snd_ack}[i].c \mid i : \text{Index} \} \text{out } \text{ack} \text{Unit}$$

$$\text{SndAck}() \equiv$$

$$\square \{ \text{snd_ack}[i].c?; \text{ack}!i; \text{SndAck}() \mid i : \text{Index} \},$$

$$\text{RcvAck: Unit} \rightarrow \text{in } \text{ack} \text{out } \{ \text{rcv_ack}[i].c \mid i : \text{Index} \} \text{Unit}$$

$$\text{RcvAck}() \equiv$$

$$\text{let } i = \text{ack}? \text{ in } \text{rcv_ack}[i].c!(); \text{RcvAck}() \text{end},$$

Also it is possible to represent the $Tx(i)$ and $Rx(i)$ processes in RSL as follows:

$$\text{Tx:Index} \rightarrow \text{in } \{ \{ \text{left}[i].c, \text{rcv_ack}[i].c \} \mid i : \text{Index} \} \text{out } \{ \text{snd_mess}[i].c \mid i : \text{Index} \} \text{Unit}$$

$$\text{Tx}(i) \equiv$$

B.2.3 Refinement verification with model checking and FDR2

Refinement relations can be defined for systems described in CSP_M in several ways, depending on the semantic model of the language which is used. In FDR2 we can check three main forms of refinement, corresponding to the traces, failures and failures-divergences refinement models. FDR2 tool allows us to verify if an implementation is a refinement of a given specification.

We can express the specification of this example that is: "The N channels should behave like the *unsynchronized parallel composition of N simple buffers*", as follows:

Copy: $\text{Index} \rightarrow \mathbf{in} \{ \text{left}[i].c \mid i : \text{Index} \} \mathbf{out} \{ \text{right}[i].c \mid i : \text{Index} \} \mathbf{Unit}$
 $\text{Copy}(i) \equiv \mathbf{let} \ x = \text{left}[i].c? \ \mathbf{in} \ \text{right}[i].c!x; \ \text{Copy}(i) \mathbf{end},$

Copies: $\mathbf{Unit} \rightarrow \mathbf{in} \{ \text{left}[i].c \mid i : \text{Index} \} \mathbf{out} \{ \text{right}[i].c \mid i : \text{Index} \} \mathbf{Unit}$
 $\text{Copies}() \equiv \parallel \{ \text{Copy}(i) \mid i : \text{Index} \}$

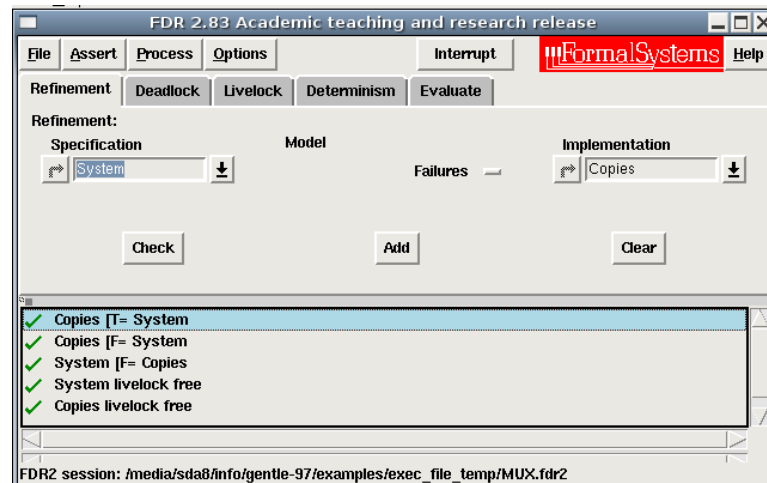


Figure B.8: FDR2 with the multiplexed buffer - refinement verification

As is shown in Figure B.8 it is possible to check that *Copies* is refined by *System* in the traces and failures refinement models. It is not necessary to check the failure-divergences model

because *Copies* and *System* are free from livelock. Finally is possible to check that *System* is refined by *Copies*, then we can conclude that the processes *System* and *Copies* are equivalent.

Appendix C

Other Examples

C.1 The railway level crossing Example

C.1.1 The problem

*One road and one railway line cross each other, and as a usual there is a *gate* which can be lowered to prevent *cars* crossing the railway. If the *gate* is *raised*, then *cars* can *freely cross* the track. A *Train* can cross the road regardless of whether the gate is up or down.*

C.1.2 The model

It is possible to use the following *Unit channels* to represent the behaviour of the system:

```
class
  channel
    car_run,car_approach, car_enter, car_leave: Unit,
    train_run,train_approach, train_enter, train_leave: Unit,
    gate_raise, gate_lower: Unit,
    crash,crash2: Unit
```

The processes *car* and *train* can be represented as follows:

```
car: Unit → out car_run,car_approach, car_enter
```

```

                                out car_leave, crash, crash2 Unit
car() ≡
  car_run!(); car_approach!(); car_enter!();
  (car_leave!(); car()
   □
   crash!(); crash2!(); stop),

train: Unit → in crash
                                out train_run, train_approach, train_enter, train_leave Unit
train() ≡
  train_run!(); train_approach!(); train_enter!();
  (train_leave!(); train()
   □
   crash?; stop),

vehicle: Unit → in crash
                                out car_run, car_approach, car_enter, car_leave, crash, crash2,
                                train_run, train_approach, train_enter, train_leave Unit
vehicle() ≡ car() || train(),

```

The *gate* can be modelled in the following way:

```

gate: Unit → out gate_raise, gate_lower Unit
gate() ≡
  gate_lower!(); gate_raise!(); gate(),

```

Then the behaviour of the *system* can be modelled as follows:

```

system: Unit → in crash
                                out car_run, car_approach, car_enter, car_leave, crash,
                                train_run, train_approach, train_enter, train_leave, crash2,
                                gate_raise, gate_lower Unit
system() ≡ vehicle() || gate(),

```

Suppose that a *car* and a *train* pass the *cross point* at the same time, then a *crash* will occur. In order to prevent this situation, a process *control* – which is placed in parallel with the *system* – will be defined as follow:

```

control: Unit → in car_approach, car_enter, car_leave,

```

```

train_approach, train_enter, train_leave,
gate_raise, gate_lower Unit
control() ≡
  train_approach?;gate_lower?; train_enter?;
  train_leave?; gate_raise?; control()
  []
  car_approach?; car_enter?; car_leave?; control(),

```

Then the behaviour of the *new system* will be as follows:

```

system1: Unit →in car_approach,car_enter,car_leave,
          train_approach, train_enter, train_leave,
          gate_raise, gate_lower, crash
          out car_run,car_approach,car_enter,car_leave, crash,crash2,
          train_run,train_approach, train_enter, train_leave,
          gate_raise, gate_lower Unit
system1() ≡ system()||control(),

```

C.1.3 Property verification with model checking

There is a *safety* and a *liveness* property that can be checked in this model which are:

- There should *never* be a *train* and a *car* on the *cross point* at the same time and
- Whenever a *car* or a *train* approaches the crossing they should eventually be able to cross.

We can express these two properties as follows:

```

safe: Unit→ out car_run,car_approach,car_enter,car_leave,
          train_run,train_approach, train_enter, train_leave,
          gate_raise, gate_lower Unit
safe() ≡ car_run!();safe()
          []
          car_approach!();safe()
          []
          car_enter!();safe()
          []
          car_leave!();safe()

```

```

[]
train_run!();safe()
[]
train_approach!();safe()
[]
train_enter!();safe()
[]
train_leave!();safe()
[]
gate_raise!();safe()
[]
gate_lower!();safe() ,

```

```

live: Unit → out car_run,train_run Unit
live() ≡ car_run!();live()
[]
train_run!();live()

```

Is possible to check that the processes *safe* is refined by *system1* in the traces refinement model because traces model is used for proving safety properties and *live* is refined by *system1* in the failures-divergences refinement model because failures-divergences refinement model is used for proving liveness properties, then these two properties are checked.

C.2 The Producer-Consumer Example

C.2.1 The problem

There are two processes, *the producer* and *the consumer*; which share a common *fixed-size buffer*. The producer's job is to generate an element, put it into the buffer and start again. At the same time the consumer is consuming the elements. The problem is to make sure that *the producer will not try to add elements into the buffer if it is full* and that *the consumer will not try to remove elements from an empty buffer*.

C.2.2 The model

It is possible to model this problem in this way:

The elements and the place to store the elements can be modelled as follows:

```

type
  Elem == E1|E2|E3,
  Buffer = Elem*

```

Then it is possible to model the *events* and *channels* of communication as follows:

```

object
  produce[e : Elem] : class channel c : Unit end,
  consume[e : Elem] : class channel c : Unit end
channel
  left, right: Elem

```

The *fixed-size buffer* can be modelled as follows:

```

N: Nat = 4,

BUFF: Buffer → in left out right Unit
BUFF(s) ≡
  if len s ≠ N then let x = left? in BUFF(s ^⟨x⟩) end
  else stop end
  []
  if s ≠ ⟨⟩ then right!(hd s);BUFF(tl s) else stop end,

BUFFini: Unit → in left out right Unit
BUFFini() ≡ BUFF(⟨⟩),

```

The behaviour of the *producer* and *consumer* processes will be modelled as follows:

```

PROD: Unit → in {produce[e].c|e:Elem} out left Unit
PROD() ≡ [] { produce[e].c?;PROD1(e) | e:Elem },

PROD1: Elem → in {produce[e].c|e:Elem} out left Unit
PROD1(e) ≡ left!e;PROD(),

CONS: Unit → in right out {consume[e].c|e:Elem} Unit
CONS() ≡ let e = right? in consume[e].c!();CONS()end,

PC: Unit → in {produce[e].c|e:Elem} out left
  in right out {consume[e].c|e:Elem} Unit
PC() ≡ PROD()||CONS(),

```

Finally putting all the processes in parallel, is possible to represent the complete behaviour of the *system*:

$$\begin{aligned} \text{SYS: Unit} &\rightarrow \mathbf{in} \{ \text{produce}[e].c|e:\text{Elem} \} \mathbf{out} \text{ left} \\ &\quad \mathbf{in} \text{ right} \mathbf{out} \{ \text{consume}[e].c|e:\text{Elem} \} \\ &\quad \mathbf{in} \text{ left} \mathbf{out} \text{ right} \mathbf{Unit} \\ \text{SYS}() &\equiv \text{PC}() \parallel \text{BUFFini}() \end{aligned}$$

C.2.3 Property verification with model checking

Is possible to check that the process *SYS* is *free from deadlock and livelock*.

C.3 The Alternating Bit Protocol (ABP) Example

C.3.1 The problem

We want to implement a buffer between two distant points that only have *unreliable channels* available; these unreliable channels can *lose* or *duplicate* as many messages as they wish, but preserve the value and order of those they transmit. There are a number of protocols available to overcome this sort of error, the simplest of which is known as the *alternating bit protocol (ABP)*.

The structure of the network used is shown in Figure C.1, where the two *error-prone* channels are *C1* and *C2*.

The basic idea is to add an *extra bit* to each message sent along the leaky channels which *alternates* between 0 and 1. The sending process sends multiple copies of each message until it is acknowledged. As soon as the receiving process gets a new message it sends repeated acknowledgements of it until the next message arrives. The two ends can always spot a new message or acknowledgement because of the alternating bit.

C.3.2 The model

The complete system will be modelled as in Figure C.2

First of all the types Bit, Data and Message are modelled in RSL as follows.

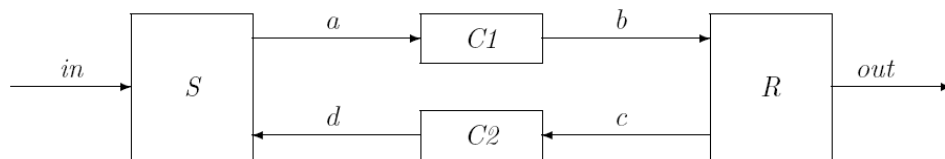


Figure C.1: The Alternating Bit Protocol

```

type
  Bit == ZERO|ONE,
  Data == D0|D1|D2|D3,
  Message ::
    bt:Bit
    dt:Data
  
```

Then it is possible to model the channels and unreliable channels as follow:

```

channel
  input,output:Data,
  a,b:Message,/* a:input, b:output*/
  c,d:Bit /* c:input, d:output*/
  
```

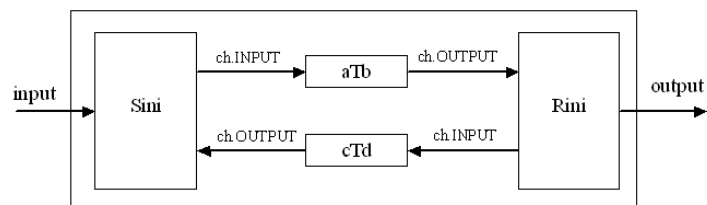


Figure C.2: The Alternating Bit Protocol model

It is possible to model the *environment* as a sort of *erroneous medium* that may lose as many messages as it likes, and repeat any message as often as it wishes. Also it is required to create a system using such media which works as long as they *do not commit an infinite unbroken series of errors*. In order to easily avoid this infinite unbroken series of errors, a counter r will be introduced. Finally the model of these unreliable channels is as follows:

```

N: Int = 5,
aTb : Int → in a out b Unit
aTb(r) ≡ let x = a? in aTb1(r, x) end ,

aTb1 : Int × Message → in a out b Unit
aTb1(r, x) ≡
  case r of
    0 → b!x ; aTb(N),
    _ →
      b!x ; aTb(N)
      ∥
      b!x ; aTb1(r - 1, x)
      ∥
  
```

```

        aTb(r-1)
    end,

aTb_ini : Unit → in a out b Unit
aTb_ini() ≡ aTb(N) ,

cTd : Int → in c out d Unit
cTd(r) ≡ let x = c? in cTd1(r, x) end ,

cTd1 : Int × Bit → in c out d Unit
cTd1(r, x) ≡
    case r of
        0 → d!x ; cTd(N),
        _ →
            d!x ; cTd(N)
            []
            d!x ; cTd1(r - 1, x)
            []
            cTd(r-1)
    end,

```

It is necessary to model a function that inverts one to zero and vice versa as follows:

```

inv : Bit → Bit
inv(bit) ≡ if bit = ZERO then ONE else ZERO end,

```

The *sender* (*Sini*) and *receiver* (*Rini*) processes are modelled as follows:

```

S : Bit → in input,d out a Unit
S(i) ≡ let x = input? in S1(i,mk_Message(i,x)) end,

S1 : Bit × Message → in input,d out a Unit
S1(i, x) ≡
    a!x;S1(i, x)
    []
    let bit=d? in if bit=i then S(inv(i)) else S1(i, x)end end,

Sini : Unit → in input,d out a Unit
Sini() ≡ S(ZERO),

R1 : Bit → in b out output,c Unit

```

```

R1(i) ≡
  let x = b? in if bt(x) = i then output!dt(x);cli; R1(inv(i))
                else clinv(i); R1(i) end
  end,

Rini : Unit → in b out output,c Unit
Rini() ≡ R1(ZERO),

```

If $C1$ and $C2$ behave as described above, then the *complete system* behaves like a reliable buffer.

```

ENV: Unit → in a,c out b,d Unit
ENV() ≡ aTb_ini()||cTd_ini(),

SYS_0: Unit → in a,c out b,d in input,d out a Unit
SYS_0() ≡ Sini()|| ENV(),

SYS: Unit → in a,c out b,d in input,d out a in b out output,c Unit
SYS() ≡ SYS_0()||Rini(),

```

C.3.3 Verifying the refinement with model checking

We can express the specification of this example as follows:

```

Copy: Unit → in input out output Unit
Copy() ≡ let x = input? in output!x; Copy() end

```

It is possible to check that *Copy* is refined by *SYS* in the traces and failures refinement models. It is not necessary to check the failure-divergences model because *Copy* and *SYS* are free from livelock. Finally it is possible to check that *SYS* is refined by *Copy*, then we can conclude that the processes *SYS* and *Copy* are equivalent.

C.4 The Multiplexed Buffer using the ABP Example

C.4.1 The problem

Consider the same problem of the *Multiplexed Buffer example* where a number of message streams are sent over a single data connection between two distant points that only have unreliable channels available, as in the *Alternating Bit Protocol (ABP) Example*.

C.4.2 The model

It is possible to model the multiplexed buffer using two processes *Sini (the sender)* and *Rini (the receiver)* and including the unreliable environment modelled in the ABP example as in the Figure C.3

First of all the types Bit, Data and Message will be slightly modified as follows.

```

type
  Index = {| i : Nat • i ∈ {0..Count} |},
  Data == D0|D1|D2|D3,
  Bit == ZERO|ONE,
  Message ::
    mb:Bit
    mi:Index
    md:Data,
  Acknowdge::
    ab:Bit
    ai:Index

```

The reliable and unreliable channels used in this example are the following:

```

channel
  a,b:Message,/* a:input, b:output*/
  c,d:Acknowdge /* c:input, d:output*/
object
  left[c : Index] : class channel c : Data end,
  right[c : Index] : class channel c : Data end,
  snd_mess[c : Index] : class channel c : Data end,
  rev_mess[c : Index] : class channel c : Data end,
  snd_ack[c : Index] : class channel c : Unit end,

```

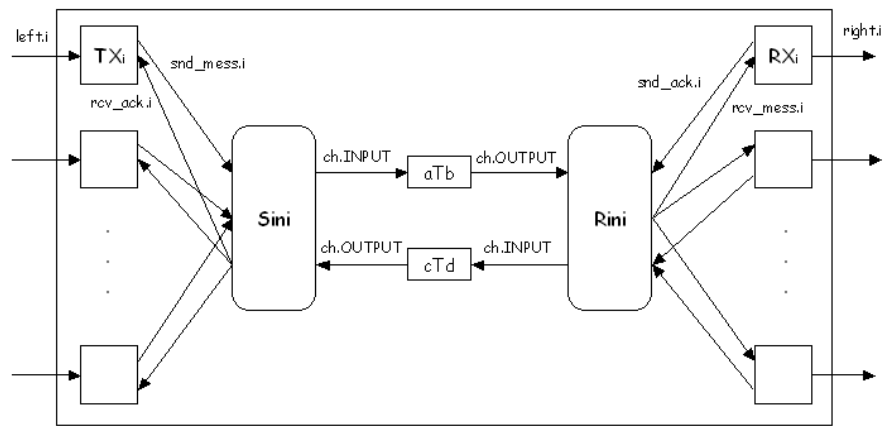


Figure C.3: Multiplexed Buffer using the ABP model

```
rcv_ack[c : Index] : class channel c : Unit end
```

The processes that model the *unreliable environment* used in the *ABP example* can be reused in this example

```
N: Int = 2,
Count : Nat = 2,
aTb : Int → in a out b Unit
aTb(r) ≡ let x = a? in aTb1(r, x) end ,

aTb1 : Int × Message → in a out b Unit
aTb1(r, x) ≡
  case r of
```

```

    0 → b!x ; aTb(N),
    — →
        b!x ; aTb(N)
        ∥
        b!x ; aTb1(r - 1, x)
        ∥
        aTb(r-1)
end,

aTb_ini : Unit → in a out b Unit
aTb_ini() ≡ aTb(N) ,

cTd : Int → in c out d Unit
cTd(r) ≡ let x = c? in cTd1(r, x) end ,

cTd1 : Int × Acknowledge → in c out d Unit
cTd1(r, x) ≡
    case r of
        0 → d!x ; cTd(N),
        — →
            d!x ; cTd(N)
            ∥
            d!x ; cTd1(r - 1, x)
            ∥
            cTd(r-1)
    end ,

cTd_ini : Unit → in c out d Unit
cTd_ini() ≡ cTd(N),

```

The function *inv* used in the *ABP example* that inverts one to zero and vice versa, is reused here:

```

inv : Bit → Bit
inv(bit) ≡ if bit = ZERO then ONE else ZERO end,

```

The *sender* and *receiver* processes are modelled as follows:

```

S : Bit → in {snd_mess[i].c | i : Index}, d out a, {rev_ack[i].c | i : Index} Unit
S(bit) ≡
    ∥ {let x = snd_mess[i].c? in S1(bit, mk_Message(bit, i, x)) end | i : Index } ,

```

```

S1 : Bit × Message → in {snd_mess[i].c | i : Index},d out a,{rcv_ack[i].c | i : Index} Unit
S1(bit, x) ≡
  a!x;S1(bit, x)
  []
  let b1=d?
  in if bit=ab(b1) then rcv_ack[ai(b1)].c!(); S(inv(bit)) else S1(bit, x)end
end,

```

```

Sini: Unit → in {snd_mess[i].c | i : Index},d out a,{rcv_ack[i].c | i : Index} Unit
Sini() ≡ S(ZERO),

```

```

R1 : Bit → in {snd_ack[i].c | i : Index},b out c,{rcv_mess[i].c | i : Index} Unit
R1(bit) ≡
  let x = b?
  in if bit=mb(x)then rcv_mess[mi(x)].c!md(x); snd_ack[mi(x)].c?;
    c!mk_Acknowledge(bit,mi(x)); R1(inv(bit))
    else c!mk_Acknowledge(inv(bit),mi(x)); R1(bit) end
end,

```

```

Rini : Unit → in {snd_ack[i].c | i : Index},b out c,{rcv_mess[i].c | i : Index} Unit
Rini() ≡ R1(ZERO),

```

The $Tx(i)$ and $Rx(i)$ processes will be reused from the *Multiplexed buffer example*:

```

Tx:Index → in {{left[i].c,rcv_ack[i].c} | i : Index} out {snd_mess[i].c | i : Index} Unit
Tx(i) ≡
  let x = left[i].c? in snd_mess[i].c!x ;rcv_ack[i].c?;Tx(i)end,

```

```

TxS: Unit → in {{left[i].c,rcv_ack[i].c} | i : Index} out {snd_mess[i].c | i : Index} Unit
TxS() ≡ ||{Tx(i) | i : Index },

```

```

Rx:Index → in {rcv_mess[i].c | i : Index} out {{right[i].c,snd_ack[i].c} | i : Index} Unit
Rx(i)≡
  let x = rcv_mess[i].c? in right[i].c!x;
  snd_ack[i].c!();Rx(i) end ,

```

```

RxS: Unit → in {rcv_mess[i].c | i : Index} out {{right[i].c,snd_ack[i].c} | i : Index} Unit
RxS() ≡ ||{Rx(i) | i : Index },

```

The next step is to put in parallel the *left* part of the system and the *right* part of the system as follows:

$$\begin{aligned} \text{LHS: Unit} &\rightarrow \mathbf{in} \{ \{ \text{left}[i].c, \text{rcv_ack}[i].c \} \mid i : \text{Index} \} \mathbf{out} \{ \text{snd_mess}[i].c \mid i : \text{Index} \} \\ &\quad \mathbf{in} \{ \text{snd_mess}[i].c \mid i : \text{Index} \}, d \mathbf{out} a, \{ \text{rcv_ack}[i].c \mid i : \text{Index} \} \mathbf{Unit} \\ \text{LHS}() &\equiv \text{Txs}() \parallel \text{Sini}(), \\ \text{RHS: Unit} &\rightarrow \mathbf{in} \{ \text{rcv_mess}[i].c \mid i : \text{Index} \} \mathbf{out} \{ \{ \text{right}[i].c, \text{snd_ack}[i].c \} \mid i : \text{Index} \} \\ &\quad \mathbf{in} \{ \text{snd_ack}[i].c \mid i : \text{Index} \}, b \mathbf{out} c, \{ \text{rcv_mess}[i].c \mid i : \text{Index} \} \mathbf{Unit} \\ \text{RHS}() &\equiv \text{Rxs}() \parallel \text{Rini}(), \end{aligned}$$

Finally, the total *system* will be represented as follows:

$$\begin{aligned} \text{sys: Unit} &\rightarrow \mathbf{in} \{ \{ \text{left}[i].c, \text{rcv_ack}[i].c \} \mid i : \text{Index} \} \mathbf{out} \{ \text{snd_mess}[i].c \mid i : \text{Index} \} \\ &\quad \mathbf{in} \{ \text{snd_mess}[i].c \mid i : \text{Index} \}, d \mathbf{out} a, \{ \text{rcv_ack}[i].c \mid i : \text{Index} \} \\ &\quad \mathbf{in} \{ \text{rcv_mess}[i].c \mid i : \text{Index} \} \mathbf{out} \{ \{ \text{right}[i].c, \text{snd_ack}[i].c \} \mid i : \text{Index} \} \\ &\quad \mathbf{in} \{ \text{snd_ack}[i].c \mid i : \text{Index} \}, b \mathbf{out} c, \{ \text{rcv_mess}[i].c \mid i : \text{Index} \} \\ &\quad \mathbf{in} a \mathbf{out} b \mathbf{in} c \mathbf{out} d \mathbf{Unit} \\ \text{sys}() &\equiv \text{LHS}() \parallel a\text{Tb_ini}() \parallel c\text{Td_ini}() \parallel \text{RHS}() \end{aligned}$$

C.4.3 Verifying the refinement with model checking

We can express the specification of this example modelling the same problem but with a reliable connection as in Figure C.4

We can express this model in RSL as follows:

```

channel
  mess: Index × Data,
  ack: Index
value

  SndMess: Unit → in { snd_mess[i].c | i : Index }, ack
                out mess, { rcv_ack[i].c | i : Index } Unit
  SndMess() ≡
    [] { let x = snd_mess[i].c? in mess!(i,x); RcvAck() end | i : Index },

  RcvAck: Unit → in { snd_mess[i].c | i : Index }, ack
                out mess, { rcv_ack[i].c | i : Index } Unit
  RcvAck() ≡
    let i = ack? in rcv_ack[i].c!() ; SndMess() end,

```

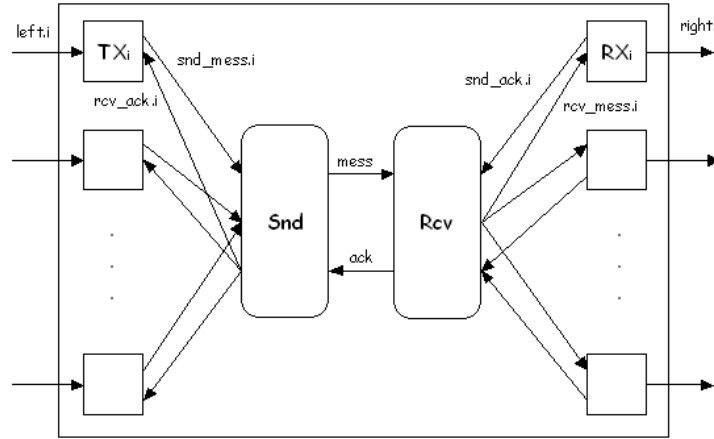


Figure C.4: Multiplexed Buffer using the ABP model

RcvMess: **Unit** \rightarrow **in** mess, {snd_ack[i].c | i : Index}
out ack, {rcv_mess[i].c | i : Index} **Unit**

RcvMess() \equiv
let (i,x) = mess? **in** rcv_mess[i].c!x ; SndAck()**end**,

SndAck: **Unit** \rightarrow **in** mess, {snd_ack[i].c | i : Index}
out ack, {rcv_mess[i].c | i : Index} **Unit**

SndAck() \equiv
 \square { snd_ack[i].c? ; ack!i ; RcvMess() | i : Index },

LHS_0: **Unit** \rightarrow **in** {{left[i].c, rcv_ack[i].c} | i : Index} **out** {snd_mess[i].c | i : Index}
in {snd_mess[i].c | i : Index}, ack **out** mess, {rcv_ack[i].c | i : Index} **Unit**

LHS_0() \equiv
 Txs() || SndMess(),

RHS_0: **Unit** \rightarrow **in** {rcv_mess[i].c | i : Index} **out** {{right[i].c, snd_ack[i].c} | i : Index}
in mess, {snd_ack[i].c | i : Index} **out** ack, {rcv_mess[i].c | i : Index} **Unit**

RHS_0() \equiv
 Rxs() || RcvMess(),

$$\begin{aligned}
\text{System: } \mathbf{Unit} &\rightarrow \mathbf{in} \{ \{ \text{left}[i].c, \text{rcv_ack}[i].c \} \mid i : \text{Index} \} \mathbf{out} \{ \text{snd_mess}[i].c \mid i : \text{Index} \} \\
&\quad \mathbf{in} \{ \text{snd_mess}[i].c \mid i : \text{Index} \}, \mathbf{ack} \mathbf{out} \text{ mess}, \{ \text{rcv_ack}[i].c \mid i : \text{Index} \} \\
&\quad \mathbf{in} \{ \text{rcv_mess}[i].c \mid i : \text{Index} \} \mathbf{out} \{ \{ \text{right}[i].c, \text{snd_ack}[i].c \} \mid i : \text{Index} \} \\
&\quad \mathbf{in} \text{ mess}, \{ \text{snd_ack}[i].c \mid i : \text{Index} \} \mathbf{out} \text{ ack}, \{ \text{rcv_mess}[i].c \mid i : \text{Index} \} \mathbf{Unit} \\
\text{System}() &\equiv \text{LHS}_0() \parallel \text{RHS}_0()
\end{aligned}$$

It is possible to check that *System* is refined by *sys* in the traces and failures refinement models. It is not necessary to check the failure-divergences model because *System* and *sys* are free from livelock. Finally it is possible to check that *sys* is refined by *System*, then we can conclude that the processes *sys* and *System* are equivalent.