



The United Nations
University

UNU-IIST

International Institute for
Software Technology

Formalising the translation from RSL to CSP

Abigail Parisaca Vargas 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

Formalising the translation from RSL to CSP

Abigail Parisaca Vargas and Chris George

Abstract

Our aim is to exploit the FDR model checker for descriptions of concurrent systems written in the RAISE Specification Language (RSL). To do this we define a translation from RSL to CSPM, show that the translation is a strong bisimulation, and then that various properties are preserved by strong bisimulation. This allows us to infer properties of the RSL description from the results of running FDR.

Abigail Parisaca Vargas is an Assistant Lecturer at San Pablo Catholic University in Arequipa, Peru. She was a UNU-IIST Fellow during May 2007 to April 2008. Her main research interest is formal methods, especially for concurrency.

Chis 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
1.1	Perspective	1
1.2	Organization of the report	2
2	Background	3
2.1	Concurrency	3
2.1.1	Communication events	3
2.1.2	Deadlock	3
2.1.3	Livelock	3
2.2	Process Algebra	4
2.3	Semantics	4
2.3.1	Operational Semantics	4
2.3.2	Denotational Semantics	5
2.3.3	Axiomatic Semantics	5
2.4	Semantics of RSL	5
2.5	Model checking	6
2.5.1	FDR model checker	6
2.5.2	Model checking in RSL	6
2.6	Using FDR for RSL	7
3	Differences between RSL and CSP	8
3.1	RSL data types and CSP_M	8
3.2	The fragments of RSL and CSP	8
3.3	Process algebras	9
3.3.1	Process, events and communications	9
3.3.2	Internal and external choice	10
3.3.3	Parallelism	10
3.3.4	Special processes	10
4	Our translation approach	12
4.1	Syntactic Normal Forms in RSL	12
4.2	Syntactic translations in RSL	13
4.3	Operational Semantics Comparison	13
4.3.1	Internal choice	14
4.3.2	External choice	14
4.3.3	Parallelism	14
4.4	Strong Bisimulation	18
4.4.1	Input	19
4.4.2	Output	19
4.4.3	Internal choice	19
4.4.4	External choice	20
4.4.5	Parallelism	21
4.4.6	if ... then ... else expressions	22
5	Properties	24
5.0.7	Traces	24
5.0.8	Deadlock	24
5.0.9	Refusals	24
5.0.10	Failures	25
5.0.11	Divergences	25
6	Conclusion	26
6.1	Future work	26

7 Appendix	27
7.1 CSP Operational rules	27
7.1.1 Hiding	27
7.1.2 Parallelism	27

Chapter 1

Introduction

Nowadays computer applications are becoming increasingly concurrent because more and more systems run on multicore machines and on distributed architectures. There are several formal methods which help us to model, describe and verify concurrent systems. All these formal methods have their own strengths and weaknesses. Therefore, it may often be convenient to use two or more of these methods to model a particular system. Since these methods are formal they have languages to express their models, and these languages by themselves have a formal design which has a formal syntax and semantics. This report explains an approach to develop a translation of RSL to CSP_M in a semantic way. In particular, we establish a semantic link between an RSL specification and its CSP_M translation by means of bisimulation. This allows us to apply the FDR model checker to RSL specifications.

1.1 Perspective

RAISE provides a full spectrum of specification features- parametrizable abstract data types, modularity, concurrency, non-determinism, subtypes- for full development from abstraction to programming languages like ADA and C++, and for formal correctness proofs.

Additionally, RSL allows specifications in some different styles like applicative or imperative; sequential or concurrent; explicit or implicit; with abstract data types or with concrete data types. [6]

On the other hand, CSP is a Process Algebra designed to be a notation and theory for describing and analyzing systems whose primary interest arises from the ways in which different components interact at the level of communication. [10]

Our purpose will be to translate the RSL applicative concurrent style process models to the suitable CSP ones as we wish to apply tools like FDR which can help us to model RSL processes.

Our approach will be, first, put the appropriate RSL syntax into its normal form. Next, taking into account the syntactic and semantic differences between RSL and CSP create a procedure to translate each syntactical RSL process expression into its corresponding CSP one. Third, by using these translations create a simulation relation and use bisimulation to prove it. Finally starting from the properties of the translation deduce the properties of the initial RSL process expressions.

1.2 Organization of the report

This report is organized as follows, in chapter 2 we describe some concepts which are necessary to have an idea about the focus of the project. In chapter 3 we detail what are the parts of each language which are going to be taken into account and in chapter 4 we describe which is going to be our translation approach ie. from the syntax to the semantics and the bisimulation. Additionally, Chapter 5 describes our approach to show that strong bisimulation preserves the properties of the semantic model, ie. traces, failures and divergences. Finally, our conclusions and the appendix.

Chapter 2

Background

2.1 Concurrency

Concurrency is concerned with the fundamental aspects of systems of multiple, simultaneously active computing agents that interact with one another. This notion is intended to cover a wide range of system architectures, from tightly coupled, mostly synchronous parallel systems, to loosely coupled, largely asynchronous distributed systems.

We can refer to a system as something that performs actions and events and has special behaviours where these behaviours could be called processes. The word 'algebra' denotes that we take an algebraic/axiomatic approach in talking about behaviour. That is, we use the methods and techniques of universal algebra [1].

2.1.1 Communication events

Generally, a communication is a synchronization between two processes along a channel at any one time. However in the CSP process algebra this synchronization can happen between two or more processes.

2.1.2 Deadlock

Deadlock refers to a specific condition when two or more processes are each waiting for another to do a communication but none of them can make any progress.

2.1.3 Livelock

Livelock refers to a process which communicates infinitely internally without any communication with any other external process.

2.2 Process Algebra

We can say that process algebra is the study of the behaviour of parallel or distributed systems by algebraic means. It offers means to describe or specify such systems, and thus it has means to talk about parallel composition. Besides this, it can usually also talk about alternative composition (choice) and sequential composition (sequencing). Moreover, we can reason about such systems using algebra and then we can do verification [1].

The basic operators, always present in some form or other, allow:

- parallel composition of processes
- specification of which channels to use for sending and receiving data
- sequentialization of interactions
- hiding of interaction points
- recursion or process replication

Following, the most basic static laws of process algebra [1],

It is stated,

$x + y = y + x$	(commutativity of alternative composition)
$x + (y + z) = (x + y) + z$	(associativity of alternative composition)
$x + x = x$	(idempotence of alternative composition)
$(x + y); z = x; z + y; z$	(right distributivity of + over ;)
$(x; y); z = x; (y; z)$	(associativity of sequential composition)
$x \parallel y = y \parallel x$	(commutativity of parallel composition)
$(x \parallel y) \parallel z = x \parallel (y \parallel z)$	(associativity of parallel composition)

Where,

+	alternative composition (binding weakest).
;	sequential composition (binding strongest).
x,y	atomic actions.

2.3 Semantics

The formal semantics of a programming language is a mathematical model which defines its meanings in a rigorous way ie the properties of the language and its essence beyond the syntax.

The major methods for defining the meaning of languages and programs are:

2.3.1 Operational Semantics

The operational semantics interprets programs as transition diagrams, with visible and invisible actions for moving between various program states and this interpretation is a mathematical formalization of some implementation strategy. [10]

2.3.2 Denotational Semantics

A denotational semantics maps a language into some abstract model in such a way that the value (in the model) of any compound program is determinable directly from the values of its immediate parts.

Using denotational semantics, we provide meaning in terms of mathematical objects, such as integers, truth values, tuples of values, and functions [11]

2.3.3 Axiomatic Semantics

An axiomatic semantics gives meaning to expressions through assertions and their correctness.

The relation between an initial assertion and a final assertion following a piece of code captures the essence of the semantics of the code [11].

2.4 Semantics of RSL

The RSL language integrates both concurrency and data in different levels of abstraction. Since it is a formal language, RSL has both a denotational and an axiomatic semantics. The models and the relevant information about the foundation can be founded in [2].

The structural operational semantics of the language was developed later and it is described in [3].

The description of the operational semantics in [3] includes a core syntax which is :

$$\begin{aligned}
 S &= D, \text{value } V, \text{axiom } A \\
 D &= \text{variable } x : \tau \mid \text{variable } x : \tau, D \mid \text{channel } c : \tau \mid \text{channel } c : \tau, D \\
 V &= c : \tau \mid c : \tau, V \\
 A &= E \equiv E \mid E \equiv E, A \\
 E &= () \mid \text{true} \mid \text{false} \mid \text{Number } n \mid id \mid x \mid \text{skip} \mid \text{stop} \\
 &\quad \mid \text{chaos} \mid \lambda id : \tau \bullet E \mid E E \mid \text{rec } id : \tau \bullet E \\
 &\quad \mid E \sqcap E \mid E \square E \mid E \parallel E \\
 &\quad \mid E ; E \mid x = E \mid c? \mid c!E \\
 &\quad \mid \text{if } E \text{ then } E \text{ else } E \mid \text{let } id = E \text{ in } E
 \end{aligned}$$

where

- E is Expressions
- x is Variables
- id is Identifiers
- c is Channels
- n is Integers
- τ is Types
- D is Declarations
- V is Value Definitions
- A is Axioms
- S is Specifications

Then the operational semantics corresponds to the denotational one, a description of the static semantics, directed by the side and communication effects, and operational semantics and its operators for internal and external choice and parallel composition is given in [3].

2.5 Model checking

Model checking is the process of checking whether a given structure is a model of a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is testing whether a given formula in the propositional logic is satisfied by a given structure.

Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula p , and a structure M with initial state s , decide if $M, s \models p$. If M is finite, as it is in hardware, model checking reduces to a graph search.

2.5.1 FDR model checker

CSP is a process algebra where the semantics of the language is based on mathematical models which are separated from the language itself. It was created to study processes and their interactions.

CSP is used as a notation for writing programs close to the implementation as well as a way to write specifications far from the implementation. Additionally, it is possible compare both and reason about it.

Some characteristics of CSP are [9] :

- basing the semantics, and equivalence, around the idea of refinement;
- separating the ideas of parallel composition and hiding, so that multiple processes can synchronise on events and enforce constraints, and hiding can be used as abstraction;
- the inclusion of a wide range of operators, both ones representing real modes of constructing processes and ones which, while pointless or difficult to implement, are useful in building specifications.

FDR (Failures-Divergence-Refinement) is a model-checking tool for state machines, with foundations in the theory of concurrency based around CSP, Hoare's Communicating Sequential Processes. Its method of establishing whether a property holds is to test for the refinement of a transition system capturing the property by the candidate machine. There is also the ability to check determinism of a state machine, and this is used primarily for checking security properties. FDR can be also be called a model checker/refinement-checker for CSP [8].

2.5.2 Model checking in RSL

RAISE is a formal method, with RSL as its specification language. Different tools have been built to be used in verification for RSL specifications, such as: test coverage analysis and mutation testing, translators to different languages and a translator to PVS which allows RSL specifications to be proved by the PVS theorem prover.

There are several techniques and strategies for doing development and proof in RAISE and the development process can be described as a step by step process going from an abstract specification to a more concrete one. Since it is suitable to apply model checking techniques to RSL specifications, one of the latest tools adds this functionality to RAISE using the Symbolic Analysis Laboratory (SAL) as a third party model checker.

SAL

SAL (Symbolic Analysis Laboratory) is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of

transition systems. A key part of the SAL framework is an intermediate language for describing transition systems. This language is intended to serve as the target for translators that extract the transition system description for other modeling and programming languages, and as a common source for driving different analysis tools. SAL describes transition systems in terms of initialisation and transition commands. These can be given by variable-wise definitions in the style of SMV or as guarded commands in the style of Murphy [4].

SAL and RSL

The SAL's symbolic model checker is sal-smc and it allows users to write assertions in LTL.

In order to use SAL as a third party model checker an extension to RSL was devised in order to allow transition system description inside RAISE specification code. Additionally, another extension was added in RAISE to add temporal logic operators to RSL and then to state properties about a given transition system. The temporal logic chosen for the extension is LTL. Temporal logic operators are only allowed in RSL's **ltl-assertions**.

2.6 Using FDR for RSL

FDR2 is a refinement checker, ie a model checker which uses the same language for the specification and the description of the system. Since we want to apply FDR in order to model RSL processes, we can state two things:

1. Implementation : There are syntactic and semantic differences between RSL and CSP which we need to cope with.
2. Specification : It is necessary to establish how to translate LTL properties from RAISE specifications into FDR.

Chapter 3

Differences between RSL and CSP

RAISE is a formal method to develop software systems with its specification language RSL.

The suggested method to specify is going from the applicative specification to the imperative specification and finally to the concurrent one. The strategy to go from one to another is "invent and verify", it means create a new specification from the former one and verify that it is correct. So, with this approach we can go from the more abstract to the more concrete specification.

In our case, we are going to work with the applicative-concurrent style of RSL.

3.1 RSL data types and CSP_M

RSL is a modular language with a very complete set of built-in types like: booleans, integers, natural numbers, real numbers, characters, texts and the Unit type. RSL also includes compound types like: products, sets, lists, maps. Moreover, functions, lambda expressions, subtype expressions, variant definitions, which includes constructor, destructor and reconstructor; case expressions, let expressions, variables and repetitive expressions.

On the other hand, CSP_M is the machine-readable dialect of CSP, it combines the CSP process algebra with a functional programming language, with the purpose of giving CSP support to describe concurrent systems in a way to be automatically manipulated. This dialect works with scripts where we find language expressions like numbers (integers), sequences or lists, sets, booleans, tuples, user-defined types, local definitions, pattern matching and lambda terms.

3.2 The fragments of RSL and CSP

RSL allows the specification of data and concurrency whereas CSP is mainly process modeling. The following grammars describe the process expressions which are going to be taken into account in the translation and which define the subsets which can be translated in both cases.

The RSL expressions are as follows:

$$P = \text{skip} \mid \text{stop} \mid E; P \mid P \square P \mid P \square P \mid P \parallel P \mid \text{if } v \text{ then } P \text{ else } P \text{ end}$$
$$E = c? \mid c!v$$

Where

- v is a pure value expression
- P is a process expression

The CSP expressions are as follows:

$$P = SKIP | STOP | E \rightarrow P | P \sqcap P | P \square P | P || P | \text{if } v \text{ then } P \text{ else } P$$

$$E = c? | c!v$$

Where

- v is a pure value expression
- P is a process expression

3.3 Process algebras

3.3.1 Process, events and communications

In RSL, we define a process as an expression with channel access and where within the scope of a channel c an expression can specify an input or an output.

So, if we have the following channel declaration,

channel $c : T$

then $c?$ is the input expression which waits for a value and $c!v$ an output expression which waits to put a value on the channel c [7]

Since in RSL we work just with communication events, the channel communication in RSL is point-to-point and synchronized.

In CSP, a process is defined as a pattern of events which are within a defined set called the alphabet of the process. An event can be a communication event if it belongs to a set X which contains all the events of the form $c.v$ where c is a channel used for a process P for output and for Q for input, where $P||Q$ and v is the value given by P and received by Q . For example:

$$(c!v \rightarrow P) || (c?x \rightarrow Q(x)) = c!v \rightarrow (P || Q(v))$$

$c!v$ will be the observable action in the communication event performed.

Since CSP is broadcasting if we want to create point to point communication we have to hide all the communications between P and Q :

$$P || Q \setminus Z$$

where $Z = X \cap Y$, $X \subseteq \alpha P$ and $Y \subseteq \alpha Q$

This is because of one of the principles mentioned in section 2.5.1: hiding and parallelism are separated ideas, so parallelism can be used to synchronise on several events. This is different to the idea of the RSL parallelism,

where communication always involves just two processes, and communication events are hidden as they occur. In RSL, one possible behaviour of

$$c!v ; P \parallel \text{let } x = c? \text{ in } Q(x) \text{ end}$$

(where v is pure, ie not itself a communicating or variable-dependent expression) is to become (via a τ event)

$$P \parallel Q(v)$$

This is always a *possible* behaviour. It is also a *necessary* behaviour iff there are no other processes running in parallel which are willing to input or output on channel c . If, for example, there were another parallel process waiting for input on channel c , then a possible behaviour would be for the first process to communicate and become P , and the second to continue to wait to input on c .

3.3.2 Internal and external choice

In the case of internal choice, both CSP and RSL use the combinator \sqcap which represents a nondeterministic choice (or an internal decision) between two processes, where in both cases the combinator is associative and commutative.

In the case of external choice, both CSP and RSL use the combinator \square which represents a choice made by the environment for the first event of the process, where in both cases the combinator is associative and commutative.

3.3.3 Parallelism

The operator for parallelism in CSP and in RSL is \parallel . However there are some differences between them when they model concurrent interactions. This is described in section 4.3.

3.3.4 Special processes

STOP, *SKIP*, *CHAOS* in CSP

1. *STOP* is a process that does not give a choice for an initial action. It is represented as follows,

$$x : \{\} \rightarrow P(x)$$

which means that the **unit** for external choice is *STOP*

2. *SKIP* is the process that terminates successfully and immediately. In sequential composition, we have the following rule:

$P \parallel Q$ terminates when both processes terminates and it is called *distributed termination*.

3. *CHAOS* can do anything at any time, and can refuse to do anything at any time. So, for any process P it will always be that $CHAOS \sqsubseteq P$ ($P \sqcap CHAOS = CHAOS$) but there is no **unit** for \sqcap .

stop, skip, chaos and swap in RSL

1. **stop** represents deadlock.
2. **skip** is the process that represents termination, and **skip** returns () of type **Unit**.
3. **chaos** represents the chaotic behaviour of non-termination, when a process fails to terminate and diverges.
4. **swap** is the process which can terminate or not, may deadlock, may behave non deterministically.

the **unit** for external choice is **stop**.

the **unit** for internal choice is **swap**.

the **unit** for parallelism is **skip**.

the **zero** for all of these combinator is **chaos**.

Chapter 4

Our translation approach

4.1 Syntactic Normal Forms in RSL

First we note that we can always obtain pure expressions (ones which do not access variables or channels) by introducing let expressions. For example:

$$\begin{aligned} & e1 \wedge e2 \\ \equiv & \mathbf{let\ } x1 = e1 \mathbf{ in\ } x1 \mathbf{ end\ } \wedge \mathbf{let\ } x2 = e2 \mathbf{ in\ } x2 \mathbf{ end} \\ \equiv & \mathbf{let\ } x1 = e1 \mathbf{ in\ let\ } x2 = e2 \mathbf{ in\ } x1 \wedge x2 \mathbf{ end\ end} \end{aligned}$$

provided we choose $x1$ so that it does not occur free in $e2$. This is particularly necessary in analysing concurrency, where, to take an extreme example:

$$\begin{aligned} & c1!(c2!e) \\ \equiv & \mathbf{let\ } x = c2!e \mathbf{ in\ } c1!x \mathbf{ end} \\ \equiv & \mathbf{let\ } x = \mathbf{let\ } y = e \mathbf{ in\ } c2!y \mathbf{ end\ in\ } c1!x \mathbf{ end} \end{aligned}$$

which clarifies the order of the evaluation of e and of the outputs.¹

We can therefore assume from now on when necessary that the constituent value expressions in our expressions are pure, and we shall represent such an expression by v .

We first show a table where each relevant syntactic expression is written down in its equivalent normal form; we are going to work with the normal form with bisimulation relations.

¹This example can in fact be simplified further to $\mathbf{let\ } y = e \mathbf{ in\ } c2!y; c1!() \mathbf{ end}$

Syntax	Normal Form
if e then $v.e$ end	if e then $v.e$ else skip end
if e_1 then $v.e_1$ elsif e_2 then $v.e_2$ else $v.e_3$ end	if e_1 then $v.e_1$ else (if e_2 then $v.e_2$ else $v.e_3$ end) end
$e_1 \wedge e_2$	if e_1 then e_2 else false end
$e_1 \vee e_2$	if e_1 then true else e_2 end
$e_1 \Rightarrow e_2$	if e_1 then e_2 else true end
$\sim e_1$	if e_1 then false else true end

4.2 Syntactic translations in RSL

There are some syntactic RSL expression which must be translated into another RSL expression in order to facilitate our translation.

We are focused on the applicative, concurrent and model-based specifications styles.

The syntactic translation from RSL to CSP_M is explained in Lizeth Tapia's report [12] and includes the following expressions

- **Built-in types**
 - Booleans
 - Integers
 - Natural Numbers
- **Compound types**
 - Products
 - Sets
 - Lists
 - Subtypes
 - Variant types
 - Record types
- **Constant values**
- **Functions**
 - if expressions
 - let expressions
 - case Expressions
- **Patterns**
- **Channels**
 - Channel definitions
 - Channel array definitions
- **Processes**

4.3 Operational Semantics Comparison

The operational semantics rules that are shown in the boxes are taken from [5] in the case of the RSL rules and from [10] in the case of the CSP rules.

4.3.1 Internal choice

RSL and CSP have the same operational semantics rules for internal choice. It is, when there is a nondeterministic choice between two expressions, one of them is selected for evaluation, via an internal action.

Below, the internal choice rules in both cases,

RSL	CSP
$(4.1) \frac{}{\rho \vdash P \sqcap Q \xrightarrow{\tau} P}$ $Q \sqcap P \xrightarrow{\tau} Q$	$(4.2) \frac{}{P \sqcap Q \xrightarrow{\tau} P}$ $\frac{}{P \sqcap Q \xrightarrow{\tau} Q}$

For internal choice we can see there is no difference between RSL and CSP.

4.3.2 External choice

In external choice in RSL and CSP, the environment influences the choice between the two expressions. It could also be the case that an argument of the process expression performs internal moves before the choice between the two arguments is done.

The following rules hold in both cases.

In the first table is shown when the environment makes the choice.

RSL	CSP
$(4.3) \frac{\rho \vdash P \xrightarrow{a} P'}{\rho \vdash P \sqcap Q \xrightarrow{a} P'}$ $Q \sqcap P \xrightarrow{a} Q' \quad a \neq \tau$	$(4.4) \frac{P \xrightarrow{a} P'}{P \sqcap Q \xrightarrow{a} P'} \quad a \neq \tau$ $\frac{Q \xrightarrow{a} Q'}{P \sqcap Q \xrightarrow{a} Q'} \quad a \neq \tau$

In the second table is shown the rules when while a process is waiting for the choice made by the environment, an internal action can be performed.

RSL	CSP
$(4.5) \frac{\rho \vdash P \xrightarrow{\tau} P'}{\rho \vdash P \sqcap Q \xrightarrow{\tau} P' \sqcap Q}$ $Q \sqcap P \xrightarrow{\tau} Q \sqcap P'$	$(4.6) \frac{P \xrightarrow{\tau} P'}{P \sqcap Q \xrightarrow{\tau} P' \sqcap Q}$ $\frac{Q \xrightarrow{\tau} Q'}{P \sqcap Q \xrightarrow{\tau} P \sqcap Q'}$

For external choice we can see there is no difference between RSL and CSP.

4.3.3 Parallelism

Since the parallel composition of two process expressions in RSL and CSP have differences in the semantics, it is necessary to analyze the operational semantics rules in both cases in order to proceed with the proper

translation from RSL concurrent processes to the corresponding CSP ones.

Synchronization

The operational semantics rule for synchronization of RSL is the following,

$$(4.7) \quad \frac{\rho \vdash P \xrightarrow{a} P' \quad , \quad Q \xrightarrow{\bar{a}} Q'}{\rho \vdash P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

where if a is an input ($c?x$) then \bar{a} is an output on the same channel ($c!e$), and vice versa.

The CSP rule is more general, in that both events may be inputs or outputs

$$(4.8) \quad \frac{P \xrightarrow{a} P' \quad , \quad Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'}$$

The notation $P \xrightarrow{a} P'$ for CSP includes the following cases:

1. a is an output on a channel c , say, so $P \xrightarrow{c!v} P'$ for some value v of the relevant type
2. a is an input on a channel c , say. In this case we can regard the input as an external choice over the possible values in the relevant type, and so for each such value, v say, $P \xrightarrow{c?v} P'$

Now we can see that the CSP rule includes 3 possible sub cases:

- Both events are outputs.

$$\frac{P \xrightarrow{c!v} P' \quad , \quad Q \xrightarrow{c!v} Q'}{P \parallel Q \xrightarrow{c!v} P' \parallel Q'}$$

- Both events are inputs.

$$\frac{P \xrightarrow{c?x} P'_{(x)} \quad , \quad Q \xrightarrow{c?y} Q'_{(y)}}{P \parallel Q \xrightarrow{c?z} P'_{(z)} \parallel Q'_{(z)}}$$

- One event is an output and one an input.

$$\frac{P \xrightarrow{c?x} P'_{(x)} \quad , \quad Q \xrightarrow{c!v} Q'}{P \parallel Q \xrightarrow{c!v} P'_{(v)} \parallel Q'}$$

Only the third of these corresponds to the assumptions of the RSL rule.

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

It is normal in RSL to adopt the following design rule, since in RSL channel communication is always between exactly two processes:

A process can only either input or output on any channel it accesses, and at most one other process can access each such channel, and accesses it in the opposite direction.

If we adopt this rule for RSL then we see that such RSL translated in the natural manner to CSP will produce CSP processes in which the two sub cases where both events are inputs, or both are outputs, cannot occur.

Finally we consider the conclusion of the rules for parallel processes. In RSL the resulting event is τ ; in CSP the resulting event is the event that was synchronized on (conventionally written as an output if the directions of the events were different). To get the same behaviour in the CSP as in the RSL we have to hide the synchronization events when we translate parallel combinations of processes. That is:

$$(P \parallel Q)_T = (P_T \parallel Q_T) \setminus \alpha P_T \cap \alpha Q_T$$

Where P_T is the CSP process translated from the RSL process P, and similarly for Q

Concurrency by agreement

The RSL rule when two processes are in parallel and permits one of them to proceed without communicating is the following :

$$\frac{\rho \vdash P \xrightarrow{\diamond} P'}{\rho \vdash P \parallel Q \xrightarrow{\diamond} P' \parallel Q}$$

where \diamond denotes an event which can be visible or silent.

In order to find the corresponding CSP rule, we are going to divide this event into any visible action **a** or in an internal action τ and we are going to work with these two cases, so we derive two rules:

1. with a visible event,

$$(4.9) \quad \frac{\rho \vdash P \xrightarrow{a} P'}{\rho \vdash P \parallel Q \xrightarrow{a} P' \parallel Q}$$

2. with an internal action,

$$(4.10) \quad \frac{\rho \vdash P \xrightarrow{\tau} P'}{\rho \vdash P \parallel Q \xrightarrow{\tau} P' \parallel Q}$$

In the first case, the corresponding CSP rules are:

$$(4.11) \quad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}$$

$$\frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}$$

Where $a \in \Sigma \setminus X$ and X represents $\alpha P \cap \alpha Q$.

However, the RSL rule is still more general, since there is not a given restriction about which alphabet a belongs to. A transition is performed from $P \xrightarrow{a} P'$ then $a \in \alpha P$, and we need to give a restriction in order to correspond to the CSP rule, which will be $a \notin \alpha Q$.

Now, we have two cases,

1) $\bar{a} \notin \alpha Q$: this case corresponds to the CSP rule (3.11).

2) $\bar{a} \in \alpha Q$: but P can not then perform a without involving Q since if $\bar{a} \in \alpha Q$ then our design rule prohibits \bar{a} from being in the alphabet of any other process.

1) So, only the first case is possible and the RSL rule corresponds to the CSP rule. 2) In the second case, the corresponding CSP rules will be,

$$(4.12) \quad \frac{P \xrightarrow{\tau} P'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \parallel Q \xrightarrow{\tau} P \parallel Q'}$$

and they are the same as the RSL rules.

So we can conclude by restating the translation rule for parallel processes:

$$(P \parallel Q)_T = (P_T \parallel Q_T) \setminus \alpha P_T \cap \alpha Q_T$$

where P_T is the CSP process translated from the RSL process P .

Termination in parallelism

In RSL and CSP \checkmark is an event that denotes termination, but this event is not seen by the environment so it is silent. Additionally the process expression to denote successful termination is *SKIP* in CSP and **skip** in RSL.

In RSL, the only event that **skip** performs is a \checkmark and after that **skip** performs it, the value $()$ is returned. The following RSL rule shows what it was described.

$$(4.13) \quad \overline{\rho \vdash \mathbf{skip} \xrightarrow{\checkmark} ()}$$

The corresponding CSP rule is shown below,

$$(4.14) \quad \overline{SKIP \xrightarrow{\checkmark} \Omega}$$

In RSL, when two processes are put in parallel and one of them performs a \checkmark the result will be a $()$ value in parallel with the other process. The following rules show this case,

$$(4.15) \quad \frac{\rho \vdash P \xrightarrow{\checkmark} ()}{\rho \vdash P \parallel Q \xrightarrow{\tau} () \parallel Q}$$

$$\frac{\rho \vdash Q \xrightarrow{\check{}} ()}{\rho \vdash P \parallel Q \xrightarrow{\tau} P \parallel ()}$$

The corresponding CSP rules in this case are shown below,

$$(4.16) \frac{P \xrightarrow{\check{}} P'}{P \parallel Q \xrightarrow{\tau} \Omega \parallel Q}$$

$$\frac{Q \xrightarrow{\check{}} Q'}{P \parallel Q \xrightarrow{\tau} P \parallel \Omega}$$

If two input values are in parallel means that the two processes compounding the parallel expression have terminated, a \checkmark event has to be performed to show that the main process has terminated.

$$(4.17) \overline{\rho \vdash () \parallel () \xrightarrow{\check{}} ()}$$

The corresponding CSP rule will be,

$$(4.18) \overline{\Omega \parallel \Omega \xrightarrow{\check{}} \Omega}$$

In brief:

$$(\mathbf{skip})_T = \mathit{SKIP}$$

$$(\mathbf{()})_T = \Omega$$

4.4 Strong Bisimulation

Our goal is to use CSP to model RSL, so that we can apply tools like FDR to CSP processes modeling RSL ones. We describe the modeling in CSP as a translation from RSL to CSP.

In order to show that results obtained from tools applied to the CSP model are valid for the original RSL we will show that the translation scheme is a strong bisimulation.

To start with, it is necessary to present the common events and processes between both process algebras.

	RSL	CSP
Events		
Input event	$c?x$	$c?x$
Output event	$c!v$	$c!v$
Internal action	τ	τ
Processes		
	skip	<i>SKIP</i>
	chaos	<i>CHAOS</i>
	stop	<i>STOP</i>

We proceed by structural induction over the syntax of RSL expressions.

For each construction we assume the component processes are strongly bisimilar with their translations, and show the constructed process to be strongly bisimilar with its translation. We use \sim as the symbol for strong bisimilarity.

4.4.1 Input

Suppose

$Q = \text{let } x = c? \text{ in } P \text{ end}$

The translation is $Q_T = c?x \rightarrow P_T$

We assume $P \sim P_T$ and need to show that $Q \sim Q_T$

We therefore need to show

(1) if $Q \xrightarrow{a} X$ then $Q_T \xrightarrow{a} Y$ where $X \sim Y$. This is immediate, taking a as $c?x$, X as P and Y as P_T \square

(2) if $Q_T \xrightarrow{a} Y$ then $Q \xrightarrow{a} X$ where $X \sim Y$. This is immediate, taking a as $c?x$, Y as P_T and X as P \square

4.4.2 Output

Recall our assumption that a value v is pure.

Suppose

$Q = c!v ; P \text{ end}$

The translation is $Q_T = c!v \rightarrow P_T$

We assume $P \sim P_T$ and need to show that $Q \sim Q_T$

we therefore need to show

(1) if $Q \xrightarrow{b} X$ then $Q_T \xrightarrow{b} Y$ where $X \sim Y$.

This is immediate, taking b as $c!v$ and X as P and Y as P_T \square

(2) if $Q_T \xrightarrow{b} Y$ then $Q \xrightarrow{b} X$ where $X \sim Y$.

This is immediate taking b as $c!v$ and Y as P_T and X as P \square

4.4.3 Internal choice

Suppose

$R = P \sqcap Q$.

The translation is

$R_T = P_T \sqcap Q_T$.

We assume that $P \sim P_T$ and $Q \sim Q_T$ and need to show that $R \sim R_T$.

There are two possible transitions for R

(1) In this case we have $R \xrightarrow{\tau} P$.

We argue as follows:

- (a) $R_T \xrightarrow{\tau} P_T$ from the internal choice rule for CSP (4.2).
- (b) Since $P \sim P_T$ we have that $R \sim R_T$. \square

Conversely, we have $R_T \xrightarrow{\tau} P_T$.

We argue as follows:

- (a) $R \xrightarrow{\tau} P$ from the internal choice rule for RSL (4.1).
- (b) Since $P_T \sim P$ we have that $R \sim R_T$. \square

(2) In this case we have $R \xrightarrow{\tau} Q$.

We argue as follows:

- (a) $R_T \xrightarrow{\tau} Q_T$ from the internal choice rule for CSP (4.2).
- (b) Since $Q \sim Q_T$ we have that $R \sim R_T$. \square

Conversely, we have $R_T \xrightarrow{\tau} Q_T$.

We argue as follows:

- (a) $R \xrightarrow{\tau} Q$ from the internal choice rule for RSL (4.1).
- (b) Since $Q_T \sim Q$ we have that $R \sim R_T$. \square

4.4.4 External choice

Suppose

$$R = P \square Q$$

The translation is

$$R_T = P_T \square Q_T$$

We assume that $P \sim P_T$ and $Q \sim Q_T$ and need to show that $R \sim R_T$

There are four possible transitions for R

(1) First, in this case we have $R \xrightarrow{\tau} P' \square Q$. This happens when $P \xrightarrow{\tau} P'$.

We argue as follows:

- (a) $P_T \xrightarrow{\tau} P'_T$ since $P \sim P_T$.
- (b) $R_T \xrightarrow{\tau} P'_T \square Q_T$ from (a) and the external choice rule (4.6). \square

Conversely, we have $R_T \xrightarrow{\tau} P'_T \square Q_T$. This happens when $P_T \xrightarrow{\tau} P'_T$

We argue as follows:

- (a) $P \xrightarrow{\tau} P'$ since $P \sim P_T$.
- (b) $R \xrightarrow{\tau} P' \square Q$ from (a) and the external choice rule (4.5). \square

(2) Second $R \xrightarrow{\tau} P \square Q'$. This happens when $Q \xrightarrow{\tau} Q'$. The argument is symmetric to (1).

(3) Third, in this case we have $R \xrightarrow{a} P'$. This happens when $P \xrightarrow{a} P'$.

We argue as follows:

- (a) $P_T \xrightarrow{a} P'_T$ since $P \sim P_T$.
- (b) $R_T \xrightarrow{a} P'_T$ from (a) and the external choice rule (4.4). \square

Conversely, we have $R_T \xrightarrow{a} P'_T$. This happens when $P_T \xrightarrow{a} P'_T$

We argue as follows:

- (a) $P \xrightarrow{a} P'$ since $P \sim P_T$.
- (b) $R \xrightarrow{a} P'$ from (a) and the external choice rule (4.3). \square

(4) Fourth, $R \xrightarrow{a} Q'$. This happens when $Q \xrightarrow{a} Q'$. The argument is symmetric to (3).

4.4.5 Parallelism

Suppose

$$R = P \parallel Q$$

The translation is

$$R_T = P_T \parallel Q_T \setminus B$$

where $B = \alpha P_T \cap \alpha Q_T$

$$\alpha P_T = \{c.e \mid e \in \alpha c\}$$

$$\alpha Q_T = \{c.e \mid e \in \alpha c\}$$

c is a channel

αc are all the events in that channel

e is an event

We assume $P \sim P_T$ and $Q \sim Q_T$ and need to show $R \sim R_T$.

There are 5 possible transitions for R

First, when $R \xrightarrow{\tau} R'$ we have three possibilities for R'

(1) $R \xrightarrow{\tau} R'$ where $R' = P' \parallel Q$. This happens when $P \xrightarrow{\tau} P'$.

In this case we argue as follows,

(a) $P_T \xrightarrow{\tau} P'_T$ since $P \sim P_T$.

(b) $P_T \parallel Q_T \xrightarrow{\tau} P'_T \parallel Q_T$ from (a) and rule for parallelism (4.8).

(c) $P_T \parallel Q_T \setminus B \xrightarrow{\tau} P'_T \parallel Q_T \setminus B$ from (b) and rule for hiding (7.3).

i.e. $R_T \xrightarrow{\tau} R'_T$. \square

Conversely, $R_T \xrightarrow{\tau} R'_T$ implies $P_T \parallel Q_T \setminus B \xrightarrow{\tau} P'_T \parallel Q_T \setminus B$. This happens when $P_T \xrightarrow{\tau} P'_T$ or $P_T \xrightarrow{a} P'_T$ so we have two possibilities for this case.

(1.1) In the first case we argue as follows,

(a) $P \xrightarrow{\tau} P'$ since $P \sim P_T$.

(b) $P \parallel Q \xrightarrow{\tau} P' \parallel Q$ from (a) and rule for parallelism (4.7).

i.e. $R \xrightarrow{\tau} R'$. \square

(1.2) In the second case, the transition $P_T \xrightarrow{a} P'_T$ is only possible if $a \in B$ (to become a τ transition) and $a \notin \alpha Q$ for design restrictions, so this transition is not possible.

(2) $R \xrightarrow{\tau} R'$ where $R' = P \parallel Q'$. This happens when $Q \xrightarrow{\tau} Q'$. In this case the argument is symmetric to (1).

(3) $R \xrightarrow{\tau} R'$ where $R' = P' \parallel Q'$. This happens when $P \xrightarrow{a} P'$ and $Q \xrightarrow{\bar{a}} Q'$.

In this case, we argue as follows,

(a) $P_T \xrightarrow{a} P'_T$ since $P \sim P_T$ and $Q_T \xrightarrow{\bar{a}} Q'_T$ since $Q \sim Q_T$.

(b) $P_T \parallel Q_T \setminus B \xrightarrow{\tau} P'_T \parallel Q'_T \setminus B$ from (a) and hiding rule (7.3) and parallelism rule (4.8). Where a and $\bar{a} \in B$

i.e. $R_T \xrightarrow{\tau} R'_T$. \square

The converse has three cases,

(a) $P_T \xrightarrow{\tau} P'_T$ and $Q = Q'_T$. This is immediate.

(b) $P = P'_T$ and $Q_T \xrightarrow{\tau} Q'_T$. This is immediate.

(c) $P_T \xrightarrow{a} P'_T$ and $Q_T \xrightarrow{\bar{a}} Q'_T$ where $a \in \alpha P_T$ and $\bar{a} \in \alpha Q_T$. This is also immediate. \square

Second, when $R \xrightarrow{a} R''$ we have four possibilities for R''

(1) In this case we have $R \xrightarrow{a} R''$ where $R'' = P' \parallel Q$. This happens when $P \xrightarrow{a} P'$ and $\bar{a} \notin \alpha Q$.

In this case, we argue as follows,

(a) $P_T \xrightarrow{a} P'_T$ since $P \sim P_T$.

(b) $P_T \parallel Q_T \setminus B \xrightarrow{a} P'_T \parallel Q_T \setminus B$ from (a) and rule for parallelism with ordinary events(7.4).

i.e. $R_T \xrightarrow{a} R''_T$. \square

Conversely, $R_T \xrightarrow{a} R''_T$ implies $P_T \parallel Q_T \setminus B \xrightarrow{a} P'_T \parallel Q_T \setminus B$. This happens when $P_T \xrightarrow{a} P'_T$.

In this case, we argue as follows,

(a) $P \xrightarrow{a} P'$ since $P \sim P_T$ and $\bar{a} \notin \alpha Q$.

(b) $P \parallel Q \xrightarrow{a} P' \parallel Q$ from (a) and rule (4.9).

i.e. $R \xrightarrow{a} R''$. \square

(2) In this case we have $R \xrightarrow{a} R''$ where $R'' = P \parallel Q'$. This happens when $Q \xrightarrow{a} Q'$ and $\bar{a} \notin \alpha P$. This argument is symmetric to (1).

(3) In this case we have $R \xrightarrow{a} R''$ where $R'' = P' \parallel Q$. This happens when $P \xrightarrow{a} P'$ and $\bar{a} \in \alpha Q$. In this case we have that this transition can not happen since our design restriction prevents \bar{a} being in the alphabet of any other process.

(4) In this case we have $R \xrightarrow{a} R''$ where $R'' = P \parallel Q'$. The argument is symmetric to (3).

4.4.6 if ... then ... else expressions

Suppose

R = if v then P else Q end

The translation is

$$R_T = \text{if } v \text{ then } P_T \text{ else } Q_T$$

where v is a boolean and $(v)_T = v$.

(1) We assume for now that v and its translation terminate and are deterministic.

We assume that $P \sim P_T$, $Q \sim Q_T$ and need to show that $R \sim R_T$

(a) In the case that v = true there is just one possibility $R=P$ and in the same way $R_T = P_T$ and $P \sim P_T$ \square

(b) In the case that v = false there is just one possibility $R=Q$ and in the same way $R_T = Q_T$, and $Q \sim Q_T$ \square

(2) If v does not terminate we assume that its translation will not.

(a) Then $R = \text{chaos}$, $R = CHAOS$ and $R \sim R_T$

(3) If v and its translation are both non-deterministic then

(a) $R = P \sqcap Q$,

$R_T = P_T \sqcap Q_T$ and we have dealt this in the case earlier in section 4.4.3

Chapter 5

Properties

After an RSL expression P has been translated into the CSP expression P_T , and it has been shown that P and P_T are strongly bisimilar it is necessary to check if the translation preserves the properties of the original process.

Theorem 5.1 *If $P \sim P_T$ then $P \xrightarrow{l} P' \Rightarrow P_T \xrightarrow{l} P'_T$ and vice versa, and in both cases $P' \sim P'_T$. Here the notation $P \xrightarrow{l} P'$ means P can do a trace l to become P' .*

5.0.7 Traces

Assert	P can do a trace l iff P_T can do a trace l .
Notation	P can do a trace l and become P' : $P \xrightarrow{l} P'$
	Proof by induction on l
Base case:	$l = \langle \rangle$ clearly $P \xrightarrow{\langle \rangle} P$ and $P_T \xrightarrow{\langle \rangle} P_T$
Inductive case:	$l = \langle h \rangle \wedge t$
$P \xrightarrow{l} P'$	$\Leftrightarrow P \xrightarrow{\langle h \rangle \wedge t} P'$
	$\Leftrightarrow P \xrightarrow{h} P''$ and $P'' \xrightarrow{t} P'$ for some P''
	$\Leftrightarrow P_T \xrightarrow{h} P''_T$ and $P''_T \xrightarrow{t} P'_T$ for some P''_T where $P'' \sim P''_T$
	$\Leftrightarrow P_T \xrightarrow{\langle h \rangle \wedge t} P'_T$
	$\Leftrightarrow P_T \xrightarrow{l} P'_T \quad \square$

5.0.8 Deadlock

P_T can deadlock	$\Leftrightarrow P_T \xrightarrow{l} P'_T \nrightarrow$ for some trace l and some P'_T
	$\Leftrightarrow P \xrightarrow{l} P' \nrightarrow$
	Theorem 5.1 and $P \sim P_T$
	$\Leftrightarrow P$ can deadlock. \square

5.0.9 Refusals

The definition of *refusals* of a process P is given below,

$$refusals(P) = \{X \mid X \subseteq \alpha P \wedge x \in X \wedge x \text{ is a refusal of } P\}.$$

where a *refusal* is an event that a process can fail to accept, no matter how long it is offered.

Theorem 5.2 *If $P \sim P_T$ then $refusals(P) = refusals(P_T)$*

Assert $refusals(P) = refusals(P_T).$

Proof

$$\begin{aligned} x \text{ is a refusal of } P &\Leftrightarrow P \xrightarrow{x} P' \\ &\Leftrightarrow P_T \xrightarrow{x} P'_T \text{ because } P \sim P_T \\ &\Leftrightarrow x \text{ is a refusal of } P_T \quad \square \end{aligned}$$

5.0.10 Failures

A *failure* is a pair (s, X) where $s \in traces(P)$ and $X \in refusals(P/s)$ and by convention a process refuses anything after \surd .

Here P/s is the process P after the trace s .

Then, the definition of failures is as follows,

$$failures(P) = \{(s, X) \mid \exists Q. P \xrightarrow{s} Q \wedge Q \text{ ref } X\} \cup \{(s, X) \mid \exists Q. P \xrightarrow{s} \surd \wedge Q\}$$

Theorem 5.3 *If $P \sim P_T$ then $failures(P) = failures(P_T)$*

Assert $failures(P) = failures(P_T).$

Notation The pair (l, X) is a failure of P where $X = refusals(P \setminus l)$

Proof

$$\begin{aligned} (l, X) \text{ is a failure of } P &\Leftrightarrow P \xrightarrow{l} P' \text{ and } refusals(P') = X \\ &\Leftrightarrow P_T \xrightarrow{l} P'_T \text{ where } P' \sim P'_T \text{ and} \\ &\quad refusals(P') = refusals(P'_T) \text{ and } refusals(P') = X \\ &\Leftrightarrow P_T \xrightarrow{l} P'_T \text{ and } refusals(P'_T) = X \\ &\Leftrightarrow (l, X) \text{ is a failure of } P'_T \quad \square \end{aligned}$$

5.0.11 Divergences

A process diverges if it can do an infinite sequence of τ transitions.

If R diverges, there is an infinite sequence

$$R_0 \xrightarrow{\tau} R_1 \dots R_i \xrightarrow{\tau} \dots \text{ where } R = R_0$$

Similarly there will be a sequence

$$R_{T_0} \xrightarrow{\tau} R_{T_1} \dots R_{T_i} \xrightarrow{\tau}, \text{ where } R_T = R_{T_0} \text{ and } R_i \sim R_{T_i}$$

If this sequence is infinite then R_T also diverges. Otherwise there is some j such that $R_{T_i} \xrightarrow{\tau}$, which could contradict $R_j \sim R_{T_j}$ and $R_j \xrightarrow{\tau}$.

So we conclude that R diverges iff R_T diverges.

Chapter 6

Conclusion

- It is always interesting combine the application of different formal methods to get different views of one system. To combine them implies deal not only with syntactic aspects of the languages but also with the semantics to do the translation consistently and correctly.
- For RSL system definitions, what it has been done is,
 - Taken a set of RSL expressions, first translated translate them into their normal forms, giving a *RSL Kernel* which was translated to a slightly restricted subset of CSP_M .
 - Shown that the translation is a strong bisimulation.
 - Shown that strong bisimulation preserves properties stated in terms of traces, deadlock, refusals, failures or divergences. ie the semantic properties of the original model are preserved.
- Lizeth Tapia's report and this one showed the semantic and syntactic translation from *RSL Kernel* to CSP_M which end with a tool with does the translation.
- Still it has not been explained which one will be the approach to translate the RSL systems specifications, since they are written in the way of LTL properties, into the respective CSP specifications.

6.1 Future work

Model checking is possible to apply to RSL specifications, in the applicative style, with SAL, and even an extension to RSL was added to allow transition system definitions in the code and part of it were add LTL operators and write assertions with them.

In order to apply model checking to the concurrent applicative style of RSL, a different approach was taken, it was to translate a set of RSL process expressions to CSP_M but this approach is not dealing with the translation from RSL *ltl-assertions* into CSP process-like specifications. To do this , first of all we need to find the way to express RSL *ltl-assertions* in a form that could be translated then into CSP specifications. Finally, the next step will be to integrate the complete translation of the implementation and then specification of a system written in RSL to its respective code in CSP. This integration will mean also the integration of two formal methods and give us means to express, analyze and compare systems expressed in RSL since the point of view of CSP.

Chapter 7

Appendix

What it is include here are the rest of the CSP operational semantics rules for hiding and parallelism taken from [10]. These rules are shown because they are the complementary rules for the ones used in this report.

7.1 CSP Operational rules

7.1.1 Hiding

$$1. (7.1) \quad \frac{P \xrightarrow{x} P'}{P \setminus B \xrightarrow{x} P' \setminus B} \quad x \notin B \cup \{\checkmark\}$$

$$2. (7.2) \quad \frac{P \checkmark P'}{P \setminus B \checkmark \Omega}$$

$$3. (7.3) \quad \frac{P \xrightarrow{a} P'}{P \setminus B \xrightarrow{a} P' \setminus B} \quad a \in B$$

7.1.2 Parallelism

$$1. (7.4) \quad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \quad a \in \Sigma \setminus B$$

$$2. (7.5) \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'} \quad a \in X$$

References

- [1] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, pages 131–146, 2005.
- [2] D. Boligano and M. Debabi. On the Foundations of the RAISE Specification Language Semantics. *BULL Research Report*, 1992.
- [3] D. Boligano and M. Debabi. RSL: An Integration of Concurrent, Functional and Imperative Paradigms. *Technical Report LACOS/BULL/MD/3/V12*, 1992.
- [4] L. de Moura, S. Owre, and N. Shankar. *The SAL Language Manual*, 2003.
- [5] M. Debabi. *The RSL semantic course*, 1993.
- [6] The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1992.
- [7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [8] Formal Systems (Europe) Ltd. *Failures-divergence refinement: Fdr2 user manual*, 2005.
- [9] A. W. Roscoe. *Model-checking CSP*, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.
- [10] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 2005.
- [11] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1994.
- [12] Lizeth Tapia and Chris George. *Model Checking Concurrent RSL with CSPM and FDR2*. Research Report 393, UNU-IIST, P.O.Box 3058, Macau, April 2008.