COMMUNICATION AND PROTOCOL SATISFACTION IN ERASMUS

Nima Jafroodi

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

July 2013

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Mr. Nima Jafroodi**
Entitled:	**Communication and Protocol Satisfaction in Erasmus**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Martin Pugh	____________________	Chair
W. Du	____________________	External Examiner
Mourad Debbabi	____________________	Examiner
Constantinos Constantinides	____________________	Examiner
Joey Paquet	____________________	Examiner
Peter Grogono	____________________	Supervisor

Approved ____

Chair of Department or Graduate Program Director

____________ 2013 ______ ______________________________

Christopher Trueman,

Interim Dean of Engineering and Computer Science

Abstract

Communication and Protocol Satisfaction in Erasmus

Nima Jafroodi, Ph.D.

Concordia University, 2013

Over the last few years, the major chip manufactures have shifted from single core towards multicore architectures, because they realized the difficulties of increasing the clock speed of processors. The spread of multicore architectures have a pervasive effect on the performance of software. In the past, application programs would effectively speed up by itself over time, but this free ride is over. With the advent of multicore processors, enhancement in the performance of applications depends upon making effective use of hardware parallelism. As a result, parallel programming has suddenly become relevant for all computer systems.

Unfortunately, parallel programming is very hard. Instead of doing everything in a sequential fashion, programmers need to ensure that their programs are designed in a way that is able to do many tasks simultaneously. As an example, in a computer game, one can't just put every game character in a separate process, running on different CPUs. What if one processor is a little faster than another, resulting in one game character moving faster than another? Somehow programmers have to ensure that all the elements of their game are synchronized, even if they are running on different threads, across multiple cores.

Programming languages can make it much easier for developers to write error free parallel programs. But the problem is that most mainstream languages do not provide suitable abstractions for expressing and controlling concurrency. Specifically, object oriented programming languages, the currently dominant paradigm, which provide concurrency through multi-threading. Object oriented programming languages are already very complex. For instance, Java provides fourteen different ways of controlling access to a variable. Adding concurrency to that makes it even harder for programmers to keep track of all concurrency issues such as shared variables, critical regions control, data races, and . . . .

The primary parallel programming language with a strong and safe support for concurrency is the Occam that is based on Tony Hoare's CSP (*Communicating Sequential Processes*). CSP is a process calculi that fully specifies process synchronization by mathematical notations. Despite its simple formalism, CSP turned out to be hard to implement efficiently. Several programming languages based on CSP appeared quickly, but they placed various restrictions on communication protocols in order to make the implementation efficient.

This thesis contributes to the Erasmus project. A process oriented programming language that aims at making the CSP paradigm more practical. Erasmus addresses concurrency by providing *processes* as the primary abstraction. Processes interact with one another through *synchronous channels*. Channels and processes are associated with *protocols* that specify the interprocess communication pattern. In this thesis we focus our attention on two problems. First, the efficient implementation of the CSP *generalized alternative* construct that allows a process to non-deterministically choose between several possible communication. Second, the design and implementation of *protocol satisfaction* that allows the Erasmus compiler to statically check the safety of interprocess communication, and hence the safety of a program.

Acknowledgments

This thesis puts an end to twenty five years of my life as a student. Twenty five years filled with wonderful memories. Memories that are identified with people who played a significant role in my life.

Foremost, I would like to express my sincere gratitude to my advisor Prof. Peter Grogono for giving me the opportunity to work with him. I am very grateful for his support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I am also grateful for his friendship and concern for things not related to work. I am sure I would not have been able to finish this thesis without his help and remarkable ideas.

Furthermore, I would like to thank the members of my thesis committee Drs. M. Debbabi, J. Paquet, C. Constantinides, and W. Du for reading the thesis, and providing useful comments and being present in my defense session. It is my privilege to have them in my thesis committee.

I would also like to highly thank Halina Monkiewicz, the graduate program advisor, for all her helps throughout all these years. My roommate Ashkan for his warm company in the last year of my studies. My childhood friends Amirali, Shahab, Alireza, Avideh, Azadeh, Farid, Mehrali and his brother Alireza, the so called "Kafoo brothers". My college friends Nima, Debbi, Siavash, Tamer, Maryam, George, Hootan, Kaveh, Nader, Saman, Mohammad, Soudeh, Shauheen, Shahab, and Yasaman. Thank you all very much!

My special thanks go to my girlfriend Mona Mehrandish whose constant supports have been so heartwarming through many cold moments. This is certainly never forgotten!

Last but certainly not least comes my family: my lovely mom Fereshteh, my lovely dad Bahram, and my lovely sister Nelia. I think that now, at the end of my PhD studies, would be the right moment to express my deepest gratitude to them for their unconditional support, encouragement and faith in me throughout my whole life and in particular, during the last few years. I can only hope that one day I would be able to return, even if only in part, the love and kindness they have extended to me. I dedicate this thesis to them, with love and gratitude.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Ugly Truth

Historically, desktop computing has been completely dominated by single core machines, but now this is changing. Until recently, advances in technology meant advances in clock speed, so software would effectively speed up by itself over time. While Moore's law [Moo65] may still be technically valid, but it is no longer true for all intents and purposes. It is still true that each year more and more transistors are fit into the same space, but the performance enhancements that this has traditionally led to has ceased years ago. Instead of increasing the clock speed, the major chip manufactures are now turning into multi-core architectures, in which parallel processors communicate directly with one another.

Multiprocessor architectures enhances performance by exploiting *parallelism*: allowing multiple processors to work on a single task. Since parallelism involves substantial communication and coordination among parallel components, the improvement in performance gained by it depends very much on the software algorithms used and their implementations [HS08]. In particular, the possible gains are limited by the fraction of the software that can be run independently in parallel. This effect is described by *Amdahl's law*, which formulates the maximum speedup that can be achieved by using n processors based on the portion of the program that can be made parallel (p) and the portion that can not be parallelized ($1 - p$):

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Some problems are "embarrassingly parallel": they can be easily divided into components that can be simultaneously executed in parallel. Such problems, if executed on multi-core architectures, realize speed up factors near the number of available cores. The ugly truth about Amdahl's law is that most problems are not accelerated so much, because they can not be completely parallelized. For example, for a given problem and a 10-processor machine, if one manages to parallelize 90% of the solution, but not the remaining 10%, then the speedup is:

$$S(10) = \frac{1}{(1 - 90\%) + \frac{90\%}{10}} \simeq 5$$

The remaining 10% that wasn't parallelized cut down the speed by half, because this part requires substantial communication and coordination among parallel processes.

1.2 The CSP Paradigm

All the points explained in the previous section, suggest that in order to achieve true performance gains software must be carefully written to take advantage of hardware parallelism. Instead of doing everything in a sequential fashion, programmers need to ensure that their programs are able to do several tasks concurrently. But concurrent programming is hard, specially for ordinary programmers. Experts agree that parallel programming is hard, because mainstream programming languages do not provide suitable abstractions for expressing and controlling concurrency [Lee06, Sut05].

Concurrency, if presented, has usually been provided by libraries, although this is known to be unsafe [Boe05]. Two of the mainstream programming languages, C and C++, completely lack any support for concurrency at the language level. Java provides support for parallel programming through threads, monitors, sockets and Remote Method Invocation (RMI) classes, but there are many concerns expressed about the way in which this support is provided. Improper implementation of monitors and semaphores, and difficulty of programming with threads are examples of such concerns [SHW00]. The primary language with strong and safe support for concurrency built-in is Occam [Cor84] which is based on Hoare's *Communicating Sequential Processes* (CSP) [Hoa78] model that fully specifies process synchronization by mathematical notations.

The CSP model has three main advantages. First, it provides *safe concurrency* by allowing the creation of large scale applications that are based on processes, compositions, and channel communications. Thus, ordinary programmers can write efficient and trustworthy concurrent programs without worrying about the correct use of concurrency primitives. Second, it provides *network transparency* of channels and *anonymity* of the communications. Thus, processes can communicate in the same way regardless of their locations, i.e., whether they are located on the same machine or distributed around a network. Third, it provides a mathematical notation for describing patterns of communication using algebraic expressions, and contains formal proofs for analyzing, verifying, and eliminating undesirable conditions such as race hazards, deadlocks, livelocks, and starvations.

Despite its simple formalism, CSP turned out to be hard to implement efficiently. Several programming languages based on CSP appeared quickly, but they placed various restrictions on communication protocols in order to make the implementation efficient: a process may only choose amongst receive operations; only a single pair of processes can be connected to a channel; or the protocol may be subject to deadlock or lack of fairness.

Joyce was developed by Brinch Hansen as a programming language for distributed systems [Han02]. It is based on Pascal and the principles of CSP. The program components are *agents* which exchange messages via synchronous, typed channels. *Polling* statements of Joyce are based on CSP's guarded alternatives, but they only allow polling on input channel statements. Ada [Led83] is one of the few industrial-strength languages that provides secure concurrency. It is an object oriented programming language, which uses CSP primitives to provide concurrency. The *selective wait* of Ada is based on CSP's guarded alternatives (only input channel statements are allowed). Occam demonstrates the possibility of efficient execution of many small processes. Occam-π [Pet05], Occam's descendant, provides mobile processes, which are processes that may be suspended, sent to another site, and resumed. The *alt* construct of Occam implements the CSP's guarded alternatives, which only allows input operations.

Considerations such as those described above led us to undertake a research project with the goal of designing a programming language that aims at making the CSP paradigm more practical. The project and the programming language are both called Erasmus. Erasmus addresses concurrency by providing *processes* as the primary abstraction. The process model provides a strong foundation and is complemented by a structuring mechanism called *cells*. Cells provide structure while processes define activities. Processes communicate by exchanging messages through synchronous channels. Each channel and processes' ports are associated with a *protocol* that determines the types of the messages that may be sent through the channel and their allowed sequences. The compiler uses protocols (*protocol satisfaction*) to analyze, verify, and eliminate undesirable conditions, e.g., *safe communication* between processes.

1.3 Problem Statement

In order to prove that the approach taken by Erasmus is viable, we must show that it can be implemented efficiently. In this thesis, we focus our attention on two areas:

1. The design and implementation of *synchronous communication* and the CSP *generalized alternative* construct,

2. The design and implementation of the *client-server* protocol.

1.3.1 Generalized Alternative Construct

A CSP program consists of a collection of processes $P_1, P_2, \ldots P_n$ that interact by exchanging messages. These message passing primitives, called input and output commands, are *synchronous*: a sender process performing an output primitive must wait until a receiver process executes the corresponding input primitive.

An important feature of CSP is the *alternative* construct which is based on Dijkstra's guarded commands [Dij75]. This construct allows a process to non-deterministically choose between several possible communications. For example, the CSP statement:

$$[P_1?m \rightarrow S_1 \;\Box\; P_2?n \rightarrow S_2]$$

means "either read m from process P_1 and perform sequence S_1, or read n from process P_2 and perform sequence S_2.

In Erasmus, port names rather than process names are used for communication, and ports have fields. Assuming that p and q are the appropriate ports, the corresponding statement would be:

```
select{
  || m := p.x; S1
  || n := q.x; S2
}
```

For uniformity, we will refer to statements of this kind as **select** statement (or alternative statement) and the task they perform as *selection* (or alternation).

All of the languages mentioned place various restrictions on selection. One restriction is to allow selection only for receiving. This restriction prevents a sender and a receiver from polling the same channel simultaneously. However, it is an asymmetry and can lead to awkward code. Another restriction is that, if one end of a channel is handled by selection, the other end of the channel must be an unconditional communication. This is a natural restriction because it is difficult to find an efficient implementation that allows selection at both ends of a channel. However, it is a serious restriction for large-scale programming because it prevents independent compilation: in order to compile a process, the compiler must inspect the code of other processes.

Buckley and Silberschatz [BS83] provide four criteria which can have a significant effect on the efficiency of the **select** statements:

i) The number of processes contributing to a single communication should be small;

ii) Processes shouldn't have too much information about the system and other processes they wish to communicate with;

iii) The number of messages required to make a communication should be small;

iv) If two processes in the system have matching send and receive commands, and they are not synchronized with any other processes, they should eventually synchronize.

The first two criteria ensure locality, the third ensures efficiency, and the last ensures progress.

One of the objective of this thesis is, therefore, to present a design and implementation for the Erasmus **select**-construct that overcomes the above restrictions, and satisfies the criteria mentioned by Buckley and Silberschatz.

1.3.2 Client-server Protocol

Similar to other process oriented languages, Erasmus follows the *client-server* relationships between processes: Server processes usually offer some services to their environment, and may themselves act as clients to other servers. Clients are processes that require some services and will obtain these services by sending requests to server processes.

A client-server communication is called *safe*, if every message sent by the client can be received by the server process, and *vice-versa*. Safety property allows the construction of client-server systems of processes that are guaranteed to be free from deadlock and livelock properties. The problem is to check programs for safety. As much safety checking as possible should be done at compile-time (static-checking). Safety checking can also be done at run-time (dynamic-checking), but this is less desirable.

To ensure safety, Erasmus channels and processes' ports (channel ends) are augmented with protocols. Protocols define both the structure of messages and allow the patterns of communication between processes to be specified. The compiler checks that each server *satisfies* its clients. Satisfaction is defined as a *relation* on labeled transition systems (LTS). Given a server and a client processes, their corresponding protocols, and a channel protocol, the compiler constructs the respective LTSs and checks whether the server process satisfies the client process with respect to the channel. A program is said to be *safe* if each of its client-server communications is safe. Otherwise, it is said to be *unsafe*.

The second objective of this thesis is, therefore, to explore some general mechanisms and structures which can be used for specifying client-server communications in Erasmus language. Particularly, we would like to define and implement satisfaction relation that can be served as a basis for static safety checking of client-server communications.

1.4 Contributions

This thesis makes the following contributions to the Erasmus project:

- The design and implementation of the **select**-construct that overcomes the restrictions previously put by other CSP like programming languages. Two models have been proposed. The first one, which we refer to as the initial design, presents a fair distributed protocol that can be used for non-deterministic message passing between Erasmus processes (see Appendix A). This work has been published in [GJ10]. The second one presents a more efficient design of the **select** construct (see Chapter 3). The concepts of *closing channels* and *priorities* are also considered within this model. This model has been validated using mCRL2 toolset: a model checker that is based on ACP process algebra. This work has been accepted to be published in [JG13].

- Defining protocol as a formalism to specify client-server communication in the Erasmus programming language (see Chapter 4). *Protocol satisfaction* is also defined as a mean for statically safety checking of programs with respect to communications. Deciding safety of programs is based on constructing a binary relation, called *satisfaction* relation, over the states of labeled transition systems corresponding to processes and protocols.

1.5 Thesis Overview

The remainder of this thesis is organized as follows: Chapter 2 presents the background. Chapter 3 describes the communication problem of Erasmus in more details, explains our initial design, and presents our final design and implementation of the **select**-construct. Chapter 4 describes our approach for specifying the client-server communication in the Erasmus programming language, and presents the definition and implementation of the satisfaction relation that can be served as a basis for the static safety checking of Erasmus programs. Finally, Chapter 5 presents the conclusion and the future work.

Chapter 2

Background

2.1 Overview

This chapter is organized as follows. Section 2.2 describes the Erasmus project in more details and presents its syntax. Section 2.3 formally defines labeled transition systems that we will be using to model Erasmus protocols and processes. Section 2.4 gives the history of process algebra, and explains the syntax of the ACP process algebra that we will be using to design the CSP generalized alternative construct. Section 2.5 describes the modal μ-calculus that we will be using to specify safety and liveness properties of our system. Finally, Section 2.6 describes the mCRL2-toolset that we will be using to validate our ACP models.

2.2 The Erasmus Project

Erasmus [GS08a, GS08b, GS08c] is a programming language that is being developed by Dr. Peter Grogono at Concordia University and Brian Shearing at the Software Factory in England. Erasmus is designed specifically for the development of concurrent systems, and is based on *Communicating Sequential Processes* (CSP) process algebra.

A *program* in Erasmus is a collection of *cells*, *processes*, and *protocols*. *Cells* are first-class citizens[1] that define structure of programs. A cell may contain processes and other cells. Processes are also first-class citizens that define activities. A process is always defined inside a cell. The code of an individual process is sequential, but processes execute concurrently and communicate by exchanging messages through *synchronous* channels. *Protocols* define interfaces of processes and specifies the communication patterns: the type and allowable sequence of messages that can be transmitted

[1]A first-class citizen is a program entity that can be named, defined, used in expressions, and passed as an argument.

through a channel. As a simple example, consider the following code:

```
protocol prot = {word: Text}
process P = p:−prot{
 p.word = "Hello World!";
}
process Q = q:+prot{
 scrln(q.word);
}
cell main = {C:prot; P(C); Q(C);}
main();
```

The program prints the "Hello World!" to the standard output device. It comprises of a protocol named *prot*, and two processes P and Q, followed by a cell definition and an instantiation of the cell. When cell *main* is instantiated, it creates a synchronous channel C, connects the ports of processes P and Q to channel C, and executes P and Q in parallel. Figure 1 shows the high-level structure of the program.

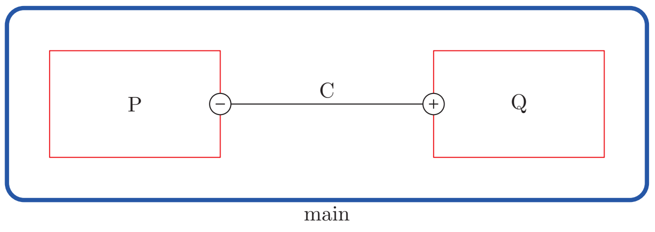


Figure 1: High-level structure of the Hello World program

Similar to other process oriented languages, Erasmus follows the *client-server* relationships between processes, where servers provide some services and clients require some services. Server and client processes are connected to each other using synchronous channels. In Erasmus a server process with respect to a channel is a process that has a port with prefix '+' before the name of the protocol associated with the port, and a client is a process that has a port with prefix '-' .

Communication betweem a client and a server processes is performed by message passing, which in turn is controlled by assignment statements of the form $v := e$. Here v, or e (or both), may be port expressions of the form $p.f$. Thus $p.f := e$ illustrates a send of data e to another process through field f of port p, and $v := p.f$ illustrates a receive of data from another process. We can also have expressions like $p1.f1 := p2.f2$, which illustrates a combined send and receive operation.

Each channel along with its processes' ports are associated with a protocol that determines the types of the message that may be sent through the channel, their directions, and their allowed sequences. For example, the protocol

protocol *prot* = {*start*; ∗(*query*: *Text*; ↑*reply*: *Integer*); *finish*}

specifies that a client process can send a *start* signal to a server, followed by repeatedly send a *query* message of type *Text* and receive a *reply* of type *Integer* from it, and then stop the conversation by sending a *finish* signal to it. The same protocol, however, specifies that a server process can receive a *start* signal from a client, followed by repeatedly receive a *query* message of type *Text* and send a *reply* of type *Integer* to it, and then stop the conversation by receiving a *finish* signal.

Protocols define interfaces of processes and specify communication patterns. Complex protocols can be defined using several operators (e.g, *sequential composition*, *alternative composition*, *repetition*, and . . .). For example, a server that provides two services has the following protocol:

protocol *prot* = {*start*; ∗((*q1*: *Text*; ↑*r1*: *Integer*) | (*q2*: *Text*; ↑*r2*: *Integer*)); *finish*}

in which the operator | specifies a choice. Protocols do not have to match exactly. Compiler checks the requirement that a server protocol *satisfies* a client protocol. Satisfaction ensures that the server can do everything that the client needs, see Chapter 4.

Erasmus includes a CSP construct, called **select**-construct, to provide a choice between communicating on different channels. For example, a server process that provides two services to two different clients (on two channels) can be defined by the following code:

```
process server = p1:+prot; p2:+prot{
 select{
   || t: Text := p1.q; p1.r:=1;
   || t: Text := p2.q; p2.r:=2;
 }
}
```

Processes and protocols can simulate functions and objects in OO languages. As an example, a process can imitate a function by providing a protocol of the form:

protocol *prot* = {∗(*I1*; *I2*; ...; *Im*; *O1*; *O2*; ...; *On*)}

in which the *I*s correspond to inputs and the *O*s correspond to outputs. Similarly, a process can imitate an object by providing a protocol of the form:

protocol *prot* = {∗(*M1* | *M2* | ... | *Mn*)}

in which the *M*s specify the behavior of the object's methods. (The choice is made by the client; the server does whatever it is asked to do.)

Alternating Bit Protocol We illustrate the salient features of Erasmus with a program that implements the specification of a communication protocol. This protocol is often referred to as Alternating Bit Protocol (ABP) [Tel94] in the literature. ABP is concerned with the transmission of data through an unreliable channel in such a way that no information will get lost.

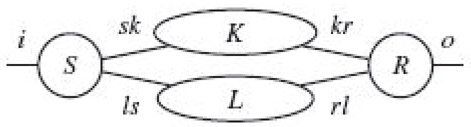


Figure 2: The alternating bit protocol

Figure 2 illustrates the communication network that we will be using in this section. Here S is the sender process that sends elements $d \in \mathit{Data}$ to the receiver process R through the unreliable channel K. Upon receiving a datum, R sends an acknowledgment back to S through another unreliable channel L. (In practice K and L are usually physically the same medium.) Now, the problem is to define processes S and R such that no information get lost.

A solution can be formulated as follows: The sender S reads a datum d at port i, and repeatedly passes this datum with an appended bit 0 (e.g, $d0$) to K until it receives an acknowledgment 0 at port *ls* (from L). Then the next datum is read, and sent on together with bit 1. The acknowledgment is then the reception of a 1 at port *ls*. The process K denotes the data transmission channel. It either passes data of the form $d0, d1, d0, \ldots$, or may corrupt data by passing an *error* message instead. Receiver R gets data of the form $d0, d1, \ldots$ from K, sends on d to port o (if d is not an *error*), and sends acknowledgment 0 resp. 1 to L. The process L is the acknowledgment transmission channel, and passes bits 0 or 1 that it receives from R, on to S. However, L may corrupt data by sending an *error* message instead of 0 or 1.

We use the following simple protocols to specify the behavior of processes and their ports with respect to channels.

protocol *prot1* = { ∗(*d*:*Data*) }
protocol *prot2* = { ∗(*d*:*Data*; *b*:*Integer* | *error*) }
protocol *prot3* = { ∗(*b*:*Integer* | *error*) }

The following illustrates the Erasmus version of the sender process S, where it receives a datum from port i, passes on this datum on port *sk*, and receives the acknowledgment (or *error*) on port *ls*.

```
process $S$ = $i$:+$prot1$; $sk$:−$prot2$; $ls$:+$prot3${
 $n$:$Integer$:=0;
 loop{
   $d$ : $Data$ := $i.d$;
   loop{
    $sk.d$ := $d$;
    $sk.b$ := $n$;
    select{
     || $m$ : $Integer$ := $ls.b$;
         if( $m == n$) then $n$:=($n$+1) mod 2; exit;
     || $ls.error$;
    }
  }
 }
}
```

The following illustrates the code for process K, where it receives a datum followed by a bit from S (on port *sk*), passes on both the datum and bit (or *error* message) on port *kr* (to R). Here the variable *random* is used to make the choice non-deterministic: the decision whether or not the data will be corrupted is internal to the channel K, and can not be influenced by the environment.

```
process $K$ = $sk$:+$prot2$; $kr$:−$prot2${
 loop{
  $random$ : $Integer$ := $Random$(0,1);
  case{
   |$random$ == 0| $kr.d$:=$sk.d$; $kr.b$:=$sk.b$;
   |$random$ == 1| $kr.error$;
  }
 }
}
```

The following illustrates the code for the process R:

```
process $R$ = $o$:−$prot1$; $kr$:+$prot2$; $rl$:−$prot3$;{
 $n$ : $Integer$ := 1;
 loop select{
   || $kr.error$;  $rl.n$ := $n$;
   || $d$:$Data$:=$kr.d$; $m$:$Integer$:=$kr.n$;
        case{
          |$m$ == $n$| $rl.n$ := $n$;
          |$m$ == (1−$n$)| $o.d$:=$d$; $n$:=($n$+1) mod 2; $rl.n$:=$n$;
        }
 }
}
```

Similar to K, process L uses a variable called *random* to make the choice non-deterministic:

```
process L = rl:+prot3; ls:−prot3;{
 loop{
  random : Integer := Random(0,1);
  case{
   |random == 0| ls.n := rl.n;
   |random == 1| ls.error;
  }
 }
}
```

The next component of the program is the *unreliableChannel*, which is a cell that encapsulates processes K and L, and defines appropriate arriving and leaving ports:

```
cell unreliableChannel = {
 sk, kr: prot2; rl, ls: prot3;
 K(sk, kr); L(rl, ls);
}
```

Next, we need two trivial processes. First, the process *Generator* that sends a sequence of data elements (from a stack) to the sender process S. Second, the process *Reporter* that displays the data elements that process R receives.

```
process Generator = i:−prot1{
 while (not stack.empty()){
   i.d = stack.top();
   stack.pop();
 }
}

process Reporter = o:+prot1{
 loop
  scrln(o.d + '\n');
}
```

The main program is a cell called *ABP* that encapsulates processes S, R, *Generator*, *Reporter*, and the cell *unreliableChannel*.

```
cell ABP = {
 i, o: prot1;
 sk, kr: prot2;
 rl, ls: prot3;
 Generator(i);
 S(i, sk, ls);
 unreliableChannel(sk, kr, rl, ls);
 R(o, kr, rl);
 Reporter(o);
}
```

The final line of the program,

ABP();

creates an instance of the cell *ABP*, which in turn instantiates the other cells and processes. Figure 3 shows the high-level structure of the program.

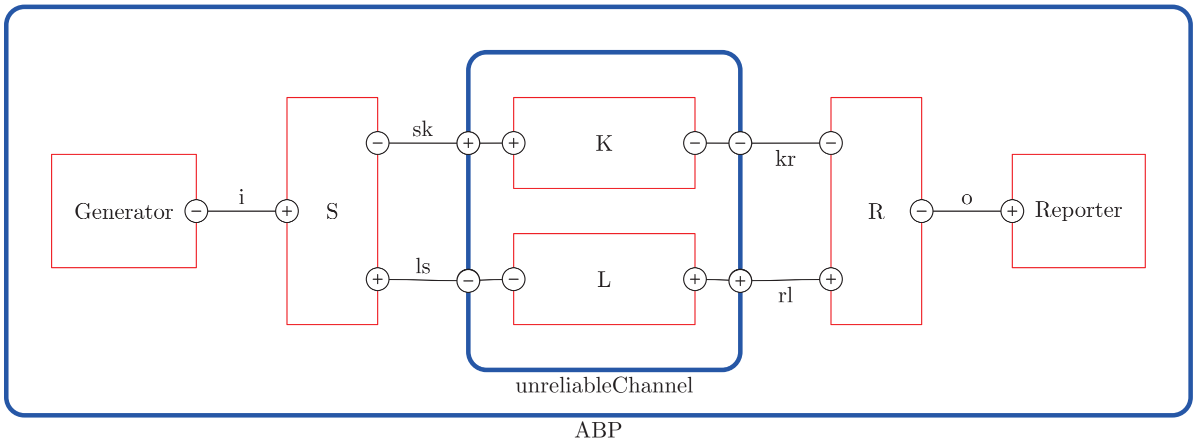


Figure 3: High-level structure of alternating bit protocol

2.3 Transition Systems

The representation that we use to model protocols and processes are *Transition Systems*. Transition systems are basically directed graphs where nodes represents *states* and edges models *transitions*.

States describe information about a system at a certain moment of its behavior. For instance, a state of a traffic light indicates the color of the light (green, yellow, or red) the traffic light displays. Similarly, a state of a sequential program indicates the current values of all program variables together with the current value of the program counter that indicates the next program statement to be executed.

Transitions, on the other hand, specify *how* the system can evolve from one state into another. In the case of the traffic light, a transition typically corresponds to switches that make the light to change from one color to another, whereas for the sequential program a transition typically corresponds to the execution of a statement and may involve the change of some variables and the program counter.

Definition 1. A *transition system* is a tuple $\mathscr{L} = (S, s_0, F, Act, T)$ where,

- S is a set of states,
- s_0 is the initial state,
- F is a set of final states,
- Act is a set of action names,
- $T \subseteq S \times Act \times S$ is a set of transitions.

The *behavior* of a transition system is defined by its initial state as well as its set of the transition relations. The notation $s \xrightarrow{\alpha} s'$ is usually used as an abbreviation of $(s, \alpha, s') \in T$. If s is the current state of a transition system, then a transition relation originating from the state s, $s \xrightarrow{\alpha} s'$, is chosen by performing the action α which causes the transition system to evolve from state s into state s'. This procedure is repeated in state s' and finishes when no transition relation is left.

It is important to realize that in a case where a state has more than one outgoing transition, the next transition is chosen in a purely *non-deterministic* manner, meaning that the outcome of this selection is not known a priori, and no statement can be made about the certain transition that is selected.

In this document, we use circles to depict states of a transition system. In addition, transitions are defined by directed edges (arrows) connecting states, initial state is indicated by the state having an incoming transition without a source state, and the final states are indicated by double cycles.

As an example, consider the transition system of a beverage vending machine depicted in Figure 4. This transition system models a vending machine that upon the insertion of a coin it nondeterministically dispenses either tea or coffee.

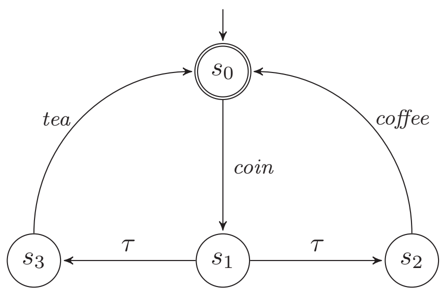


Figure 4: A beverage vending machine modeled by a transition system

The state space of the above example is $S = \{s_1, s_2, s_3, s_4\}$, the initial state is s_0, the set of final states is $F = \{s_0\}$, and the set of action names is Act= $\{$*coin, coffee, tea*$\} \cup \{\tau\}$. The action *coin* indicates the insertion of a coin by a customer, and the actions *coffee* and *tea* indicate the actions of dispensing coffee or tea by the machine respectively. The action τ is a special action that represents an internal activity of the vending machine.

A transition system $\mathscr{L} = (S, s_0, F, Act, T)$ is called *finite* if S, and Act are finite, and *infinite* otherwise. In this document, we assume that all the transition systems are finite.

Definition 2. Let $\mathscr{L} = (S, s_0, F, Act, T)$ be a transition system. For $s \in S$ and $\alpha \in Act$, the set of *direct α-successors* of s is defined as follows:

$$Post(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\}, \text{ and } Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

Similarly, the set of α*-predecessors* of s is defined as follows:

$$Pre(s,\alpha) = \{s' \in S \mid s' \xrightarrow{\alpha} s\}, \text{ and } Pre(s) = \bigcup\nolimits_{\alpha \in Act} Pre(s,\alpha)$$

Definition 3. Let $\mathscr{L} = (S, s_0, F, Act, T)$ be a transition system. $\mathscr{L}$ is called *deterministic* if for any action $\alpha \in Act$ and any state $s \in S$ the following holds:

$$|Post(s,\alpha)| \leq 1$$

Definition 4. A *run* of a transition system is an ordered and possibly infinite set of transition relations:

$$\sigma = \{s_0 \xrightarrow{\alpha_0} s_1, s_1 \xrightarrow{\alpha_1} s_2, s_2 \xrightarrow{\alpha_2} s_3 \ldots\}$$

An accepting run of a transition system is a finite run σ in which the final transition $s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$ has the property $s_n \in F$.

Definition 5. A state s in a transition system is called *reachable* if and only if is there exists a finite run σ such that:

$$\sigma = \{s_0 \xrightarrow{\alpha_1} s1, \ldots, s_n \xrightarrow{\alpha_n} s_n = s\}$$

A set of all reachable states $Reach(\mathscr{L})$ denotes the set of all reachable states in $\mathscr{L}$.

2.4 Process Algebra

This section describes the notion of process algebra, provides a brief history of it, and gives a review of the syntax of the ACP algebra that we will be using to model concurrent systems in this thesis.

The term ‘process algebra’ refers to a field of study that studies the behavior of parallel systems. Consider the word ‘process’. It refers to the *behavior* of a *system*. A system is anything showing behavior, in particular the execution of a software system, the actions of a machine, or even the actions of human being. Behavior is the total of events (or actions) that a system can perform, the order of which they can be executed, and possibly other aspects of this execution such as timing. Usually, the actions are thought to be discrete: occurrences of actions are at some moment in time, and different actions can be distinguished in time. This is why a process is also called a *discrete event system* [BBR10].

On the other hand, the term ‘algebra’, refers to the fact that the approach taken to reason about behavior is algebraic and axiomatic. Indeed, process algebra has its root in universal algebra. A process algebra is a mathematical structure that consists of a single universe of elements (processes), a set of operators defined on this universe, and a set of axioms (laws) that allow calculations on the elements of the universe.

Process algebra is usually considered to be an approach to *concurrency theory.* Concurrency theory is the theory of interacting, parallel or distributed systems. Process algebra started in the 1970s, when the only part of concurrency theory that existed was the theory of Petri nets [Pet62]. At that time, three main style of formal reasoning about computer programs could be distinguished, focusing on giving semantics (meaning) to programming languages.

1. *Operational Semantics*: A computer program is modeled as an execution of an abstract machine. A state of such a machine is a valuation of variables, and a transition between states is an elementary program instruction. The pioneer of this field is McCarthy [Mca63].

2. *Denotational Semantics*: A computer program is modeled as a function transforming input into output. This field was instrumental in the development of the *automata theory.* Pioneers of this field are Scott and Strachey [SS71].

3. *Axiomatic Semantics* An axiomatic semantics emphasizes more on using proof methods to prove the correctness of programs. Central notions are program assertions, proof triples consisting of precondition, program statement, and postcondition, and invariants. Pioneers are Floyd [Flo67] and Hoare [Hoa69].

Then the question was raised on how to give semantics to concurrent programs: programs containing parallel operators. It was found that it is very hard to give semantics to concurrent or distributed programs using only the methods of denotational, operational, or axiomatic semantics. For this reason, process algebra was developed. However, there are two paradigm shifts that needed to be made before a theory of parallel programs in terms of a process algebra can be developed. First of all, the idea of a behavior as an input/output function needed to be abandoned. This is because the interaction a process has between its input and output may influence the outcome, disrupting the functional behavior. Second, the notion of *global* variables needed to be overcome. This is because the independent execution of parallel processes makes it difficult to determine the values of global variables at any given time. It turned out to be simpler to let each process to have its own local variables, and to denote exchange of information explicitly via *message passing.*

CCS

The central person in the history of process algebra is without a doubt Robin Milner. He developed his process theory CCS, the *Calculus of Communicating Systems* [Mil82], over the yeas of 1973 to 1982. In CCS, Milner introduced *synchronization trees* to model parallel systems. Transitions were labeled with ports where a named port synchronizes with the port with its co-name. The operators that he introduced for specifying parallel programs were sequential composition, parallel composition, alternative composition, restriction (to prevent certain actions from happening), and relabeling (for renaming ports). He also introduced some laws for these operators.

CSP

Another very important contributor to the development of process algebra is Tony Hoare who developed the theory of *Communicating Sequential Processes* (CSP) [Hoa78]. The most important step was that he put away the use of global variables, and adopted the message passing paradigm of communication. CSP has synchronous communication and is a guarded-command language (based on Dijkestra [Dij75]). The first model that was introduced in this theory was based on *trace theory* (sequences of actions a process can perform). Later, it was found that deadlock behavior is not preserved in the trace model, so a new model based on *failure pairs* (actions a process cannot perform at each state) was introduced. CSP has an additional operator than those defined in CCS to distinguish between internal and external non-determinism.

π-Calculus

The π-calculus is a process calculus that was developed by Robin Milner, Joachim Parrow and David Walker [MPW92]. The theory can be seen as a continuation of Milner's work on the process calculus CCS (Calculus of Communicating Systems). The π-calculus allows channel names to be defined as a mean for communication between components. Channels are mobile, meaning that channel names can be communicated along the channels themselves. Mobile channels allow describing of concurrent systems whose network configuration may change during their executions.

ACP

The *Algebra of Communicating Processes* (ACP) was initially developed by Jan Bergstra and Jan Willem Klop [BK82, BK85], as a part of an effort to investigate the solutions of unguarded recursive equations. They were the first who used the term 'process algebra', with exactly the two meanings given in the first two paragraphs of this section. They first defined the theory with alternative, sequential, and parallel composition, but without communication. A model was established based on projective sequences, meaning that a process is given by a sequence of approximations by finite terms. Later it was shown that all recursive equations, both guarded and unguarded, have a solution in the model. The algebra was later extended with the communication operator to yield the theory ACP.

2.4.1 Syntax of ACP

Algebra of Communicating Processes (ACP) is a process algebra that provides a way to describe systems in terms of algebraic process expressions. Processes can perform actions and can be composed to form new processes using algebraic operators.

Atomic action ACP uses atomic actions as its primitives. As an example, the process term

$$P := a$$

illustrates a process that offers its environment the action 'a'. Some actions have special meanings. For example, the action δ represents a deadlocked process that can not perform any actions, the action τ represents an internal action of a process that is not observable by the environment, and the action ϵ represents the empty process that allows us to distinguish between successful and unsuccessful termination. Actions can be parameterized with data elements. As an example, a process that uses the atomic action *send* to send the data value $d : D$ can be defined as:

$$P := \mathit{send}(d)$$

Sequential composition Actions can be combined to form processes using a variety of operators. The simplest operator is probably the *prefix operator* (denoted by '.'). As an example, the process term

$$P := a.P'$$

illustrates a process that offers its environment the action 'a', and after performing the action, it behaves as the process term P'. The *sequential composition* operator (denoted by '$\cdot$') is the generalization of the prefix operator. Given the process terms P and Q, the term $P \cdot Q$ denotes the sequential composition of P and Q. The intuition of this operation is that upon the *successful* termination of process P, process Q is started. If process P ends in a deadlock, then the sequential composition $P \cdot Q$ also deadlocks.

Alternative composition Another fundamental algebraic operator is the *alternative* operator (denoted by '+'). The process term

$$P := a.\delta + b.\epsilon$$

illustrates a process that is willing to perform either action 'a' followed by the deadlock action, or action 'b' followed by the successful termination. Note that the choice is solely made by the environment, and that the process doesn't have any control over which choice will be chosen.

Abstraction The *abstraction operator* (denoted by 'τ_I') provides a way to hide certain actions (actions in I), and treats them as events that are internal to the systems being modeled. As an example, the process term $P := \tau_I(Q)$ acts like process Q, except that actions from I are hidden (renamed to τ). That is:

$$P := \tau_I(a.b.c) = a.\tau.b \text{ where } I = \{b\}$$

Encapsulation Unlike the abstraction operator, the *encapsulation operator* (denoted by '∂_H') provides a way to block certain actions (actions in H) from happening. As an example, the process term $P := \partial_H(Q)$ acts like process Q, excepts that actions in H never happen (renamed to δ). That is:

$$P := \partial_H(a.b.c) = a.\delta.b \text{ where } H = \{b\}$$

Communication Interaction between processes is defined by the *communication function* (denoted by 'γ'), and the merge operator (denoted by '$|$'). The communication function takes a pair of communicating actions and returns the result of the communication, which is also an action. As an example, consider the process term $P := \mathit{send}(1)$ that can perform a send action, parameterized with value 1, and assume that P is executing in parallel with the process term $Q := \mathit{receive}(1)$ that can perform a receive action, parameterized with data value 1. Now, suppose that an atomic action $\mathit{comm}(1)$ is the result of the simultaneous executions of these actions. Thus, communication between P and Q can be achieved as follows:

$$(P \mid Q) = \Big(\mathit{send}(1) \mid \mathit{receive}(1)\Big) = \mathit{comm}(1),\ \text{ where } \gamma(\mathit{send}(1), \mathit{receive}(1)) = \mathit{comm}(1)$$

If two actions do not communicate, then their communication function is not defined, and the result of their merge is equal to the deadlock action. For example:

$$\Big(\mathit{send}(5) \mid \mathit{receive}(6)\Big) = \delta,\ \text{ where } \gamma(\mathit{send}(5), \mathit{receive}(6)) = \text{ undefined}$$

Choice quantifier In order to allow choices over range of a data type, the *choice quantifier* (denoted by '$\sum$') is defined. As an example, consider the following

$$\sum_{d:Nat} a(d) = a(1) + a(2) + \ldots + a(n)$$

Having defined the choice quantifier, the following holds:

$$\Big(\mathit{send}(1) \mid \sum_{d:Nat} \mathit{receive}(d)\Big) = \mathit{comm}(1),\ \text{ where } \gamma(\mathit{send}, \mathit{receive}) = \mathit{comm}$$

Parallel composition Another fundamental algebraic operator is the *parallel* operator (denoted by '$||$'). The *parallel operator* illustrates the parallel composition of processes, where the individual actions are interleaved. As an example, the process term (assuming that no communication can occur)

$$P := (a.b) \mid\mid (c.d)$$

illustrates a process that may perform the actions of a, b, c, d in any of the following sequences: $(a.b.c.d)$, $(a.c.b.d)$, $(a.c.d.b)$, $(c.a.b.d)$, $(c.a.d.b)$, and $(c.d.a.b)$.

In order to express all the interleaving options given above, the parallel operator is defined in terms of the *left merge* operator (denoted by $\mathbin{|\!\lfloor}$) and the *merge* operators. That is;

$$P \mid\mid Q = P \mathbin{|\!\lfloor} Q + Q \mathbin{|\!\lfloor} P + P \mid Q$$

Here the $P \mathbin{|\!\lfloor} Q$ denotes the parallel composition of P and Q with the restriction that the first step comes from P, and $P \mid Q$ (as defined earlier) denotes the parallel composition of P and Q starting with a joint activity (communication). For example, the process term (assuming that no communication can occur)

$$P := (a.b) \mathbin{|\!\lfloor} (c.d)$$

only performs the sequences $(a.b.c.d)$, $(a.c.b.d)$, and $(a.c.d.b)$ since the left merge operator ensures that the action 'a' occurs first.

Guarded commands Conditional statements are specified in ACP by the unary operator called *guarded-command* operator. As an example, '**if** (ϕ) **then** a' can be specified by '$(\phi) :\rightarrow a$', and '**if** (ϕ) **then** a **else** b' can be specified either by '$(\phi) :\rightarrow a + (\neg\phi) :\rightarrow b$', or by '$(\phi) :\rightarrow a \diamond b$'.

We finish this section by giving the algebraic axioms for the operators defined in this section. Table 1 illustrates the algebraic axioms for sequential composition, alternative composition, abstraction, encapsulation, left merge, communication merge, parallel composition, guarded commands, and the choice quantifier operators.

The following is a brief explanation of the meaning of these axioms. Axioms A_{1-10} illustrate the properties of the alternative composition and sequential composition operators. Axioms A_1, A_2, and A_3 express the fact that the alternative composition is *commutative*, *associative*, and *idempotent* respectively. Axiom A_4 describes the distribution of sequential composition over alternative composition from the right. Note that the sequential composition does not distribute over the alternative composition from the left; that is: $x \cdot (y + z) \neq x \cdot y + x \cdot z$. Axiom A_5 states that the sequential composition is associative. Axiom A_6 expresses that in the context of a choice deadlock is avoided as long as possible. Axiom A_7 states that after a deadlock has been reached no continuation is

$x + y = y + x$	A_1	$x + \delta = x$	A_6
$(x + y) + z = x + (y + z)$	A_2	$\delta.x = \delta$	A_7
$x + x = x$	A_3	$x \cdot \epsilon = x$	A_8
$(x + y) \cdot z = x \cdot z + y \cdot z$	A_4	$\epsilon.x = x$	A_9
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A_5	$a.x \cdot y = a.(x \cdot y)$	A_{10}
$\partial_H(\epsilon) = \epsilon$	D_1	$\tau_I(\epsilon) = \epsilon$	TI_1
$\partial_H(\delta) = \delta$	D_2	$\tau_I(\delta) = \delta$	TI_2
$\partial_H(a.x) = \delta \quad (\text{if } a \in H)$	D_3	$\tau_I(a.x) = \tau.\tau_I(x) \quad (\text{if } a \in I)$	TI_3
$\partial_H(a.x) = a.\partial_H(x) \quad (\text{if } a \notin H)$	D_4	$\tau_I(a.x) = a.\tau_I(x) \quad (\text{if } a \notin I)$	TI_4
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D_5	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI_5
$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$	M	$x \mid y = y \mid x$	SC_1
$\delta \lfloor\!\lfloor x = \delta$	LM_1	$x \parallel \epsilon = x$	SC_2
$\epsilon \lfloor\!\lfloor x = \delta$	LM_2	$\epsilon \mid x + \epsilon = \epsilon$	SC_3
$a.x \lfloor\!\lfloor y = a.(x \parallel y)$	LM_3	$(x \parallel y) \parallel z = x \parallel (y \parallel z)$	SC_4
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	LM_4	$(x \mid y) \mid z = x \mid (y \mid z)$	SC_5
$\delta \mid x = \delta$	CM_1	$(x \lfloor\!\lfloor y) \lfloor\!\lfloor z = x \lfloor\!\lfloor (y \parallel z)$	SC_6
$(x + y) \mid z = x \mid z + y \mid z$	CM_2	$(x \mid y) \lfloor\!\lfloor z = x \mid (y \lfloor\!\lfloor z)$	SC_7
$\epsilon \mid \epsilon = \epsilon$	CM_3	$x \lfloor\!\lfloor \delta = x \cdot \delta$	SC_8
$a.x \mid \epsilon = \delta$	CM_4	$x \lfloor\!\lfloor \tau.y = x \lfloor\!\lfloor y$	SC_9
$a.x \mid b.y = c.(x \parallel y) \quad (\text{if } \gamma(a,b) = c)$	CM_5	$x \mid \tau.y = x \lfloor\!\lfloor y$	SC_{10}
$a.x \mid b.y = \delta \quad (\text{if } \gamma(a,b) \text{ is not defined})$	CM_6		
$\mathit{true} :\to x = x$	GC_1	$\sum_n x = x \quad (\text{if } n \text{ not free in } x)$	CQ_1
$\mathit{false} :\to x = \delta$	GC_2	$\sum_n x = x + \sum_n x$	CQ_2
$\phi :\to \delta = \delta$	GC_3	$\sum_n (x + y) = \sum_n x + \sum_n y$	CQ_3
$\phi :\to (x + y) = (\phi :\to x) + (\phi :\to y)$	GC_4	$\sum_n \phi :\to x = \phi :\to \sum_n x \quad (\text{if } n \text{ not free in } \phi)$	CQ_4
$(\phi \vee \psi) :\to x = (\phi :\to x) + (\psi :\to x)$	GC_5	$\sum_n \phi :\to x = \exists n \phi :\to x \quad (\text{if } n \text{ not free in } x)$	CQ_5
$\phi :\to (\psi :\to x) = (\phi \wedge \psi) :\to x$	GC_6	$\sum_n (n = v) :\to x = \sum_n (n = v) :\to x[v/n]$	CQ_6

Table 1: Axioms for the ACP operators defined in this section

possible. Axioms A_8 and A_9 express that the empty process is an identity element with respect to sequential composition.

Axioms D_{1-5} (resp. TI_{1-5}) illustrate the properties of the encapsulation (resp. abstraction) operator, that blocks (resp. skip) the execution of actions from H (resp. I) , Axiom D_3 (resp. I_3), and leaves the other actions unchanged, Axiom D_4 (resp. I_4). Axioms D_1 (resp. I_1) and D_2 (resp. I_2) express the fact that the actions ϵ and δ can not be blocked (resp. skipped).

Axioms M, and LM_{1-4} illustrate the properties of the parallel composition, and the left merge operators respectively. Axiom M expresses the fact that the parallel composition can be broken up into three alternatives, namely the part where the first step comes from x, the part where the first step comes from y, and the part where x and y execute together. Axioms LM_1 and LM_2 express that a (successfully or unsuccessfully) terminated process cannot perform a step, which implies that these constants as the left operand of a left merge lead to the deadlock action. Axiom LM_3 expresses the fact that in the parallel composition of processes $a.x$ and y where the first step is from $a.x$, this first step must be an 'a'. What remains is the parallel composition of x and y without any restrictions. Axiom LM_4 expresses that the moment of choice on both process terms $(x+y) \mathbin{\|\!\!\lfloor} z$ and $x \mathbin{\|\!\!\lfloor} z + y \mathbin{\|\!\!\lfloor} z$ is the same, because the choice is made by the execution of the first action.

Axioms CM_{1-6} illustrate the properties of the communication merge operator. Axioms CM_1 expresses that the action δ on one side of a communication merge allows no joint activity. Axioms CM_2 expresses that the merge operator distributes over choice, because the communication merge operator involves activity from both sides. Axiom CM_4 expresses that communication-merge expressions combining an action prefix and an empty process (ϵ) lead to deadlock, because both a joint action and synchronized successful termination are impossible. Axioms CM_5 and CM_6 illustrates communication-merge expressions where both sides are action prefixes. In this case, the result is based on the communication function γ. If γ is defined for the involved actions, then the communicating processes can perform the defined communication action, and then proceed as parallel composition of the remaining behaviors of both operands of the communication. However, if γ is undefined, the communicating processes cannot perform any action at all.

Besides the axioms M, LM_{1-4}, and CM_{1-6}, ACP contains seven additional axioms called *Axioms of Standard Concurrency* that denote properties of parallel composition operator. The commutativity and associativity of the merge operator, and the fact that ϵ is an identity element are captured by Axioms SC_1, SC_4, and SC_2 respectively. Axiom SC_3 captures the fact that a communication with the empty process either results in a successful termination or a deadlock. Axioms SC_{4-8} are basic axioms of he theory of parallel processes, and they can not be derived by the other axioms. Axioms SC_9 and SC_{10} expresses the fact that communications with the silent step τ are always assumed to be undefined.

Axioms GC_{1-6} express the properties of the guarded-commands, and are mostly self-explanatory. Finally, Axioms CQ_{1-6} express the properties of the choice quantification. CQ_1 is a generalization of the Axiom A_3. It considers the case where the variable n does not occur free in the term x. In that case, all summands are equal and by idempotency, the sum is equal to one term. CQ_2 deals with separating out one summand. Axiom CQ_3 states that the choice quantification distributes over alternative construct. Axiom CQ_4 gives distribution over guarded commands. Axiom CQ_5 states that a choice quantification over a conditional reduces to a boolean existential quantification in the conditonal if the bound variable doesn't occur in the guard term. Finally, Axiom CQ_6 allows to substitute a variable for any expression it is equal to.

2.5 The Modal μ-Calculus

In the previous section, we explained how to model parallel systems using ACP algebra. In this section, we discuss how to specify properties of such systems. A property usually describes some aspect of the behavior of a system. Freedom of deadlock, livelock, and the fact that every message that is sent will eventually be received are typical examples of properties of a system. There are two main reasons to formulate properties of a system:

1. In the early design stage, it is unclear what the behavior of a system will be. Therefore, writing down basic properties can help us to establish some of the essential aspects of the system. For example, *use cases* in UML are used for this purpose. They are examples of the potential run of a system. In this section, we will describe a property language which not only allows us to denote use cases, but also allows to denote properties which hold for all runs of a system.

2. It is very common that a system is designed or implemented incorrectly. Checking that the behavior of a system satisfies its desirable properties, guarantees the correctness of its design and implementation.

Two standard types of correctness properties that can be verified are the *safety* and *liveness* properties. A safety property of a system establishes that "*something bad never happens*". Examples are freedom from deadlock and system invariance (*x is always less than $y + 5$*). A liveness property, on the other hand, establishes that "*something good eventually happens*". An example of liveness property is the responsiveness of a system (*every request is eventually followed by an acknowledgment*). Formulating safety and liveness properties, and verifying these properties provide a convenient and effective way to guarantee the correctness of a system. In the rest of this section, we will introduce three ways of specifying the safety and liveness properties that we will be using in this thesis.

2.5.1 Hennessy-Milner Logic

Hennessy-Milner logic [HM85] was introduced by Matthew Hennessy and Robin Milner in 1980 as an approach to formulate *logical correctness* of a system. Logical correctness of a system determines which design requirements could possibly be violated, not in how probable such violation might be. The syntax of Hennessy-Milner logic is given by the following BNF grammar:

$$\phi \ ::= \ \mathit{true} \ | \ \mathit{false} \ | \ \neg\phi \ | \ \phi_1 \vee \phi_2 \ | \ \phi_1 \wedge \phi_2 \ | \ \langle a \rangle \phi \ | \ [a]\phi$$

The modal formula *true* is true in each state of a process, and the modal formula *false* is always false in each state of a process. The connectives $\vee$ (or), $\wedge$ (and), and $\neg$ (negation) have their usual meanings. The diamond modality $\langle a\rangle\phi$ is valid whenever an a action can be performed such that ϕ becomes valid. An example of diamond modality is the formula:

$$\langle \mathit{send}\rangle\langle \mathit{ack}\rangle\langle \mathit{signal}\rangle \mathit{true}$$

which expresses that it is possible for a process to do a *send* followed by an *acknowledgment* followed by issuing a *signal*. The box modality $[a]\phi$ is valid whenever for every action a that can be done, ϕ holds after doing that a. So, the formula

$$[\mathit{send}][\mathit{ack}]\mathit{true}$$

expresses that every *send* action is followed by an acknowledgment.

In order to clarify the differences between the diamond modality and the box modality, consider the four transition systems given in Figure 5. The transition system at the left illustrates a situation where $\langle a\rangle\phi$ is valid and $[a]\phi$ is not valid. This is because, from the initial state, there is an a action to a state where ϕ holds, and one to a state where ϕ doesn't hold. In the second transition system, there is no a action at all, so certainly not one to a state where ϕ holds. Thus, $\langle a\rangle\phi$ is not valid. However, all a-transitions (which are none) go to a state where ϕ is valid. Thus $[a]\phi$ is valid. The third transition system illustrates a situation where both modal formulas are valid. Finally, the last transition system illustrates a situation where neither $\langle a\rangle\phi$ nor $[a]\phi$ is valid.

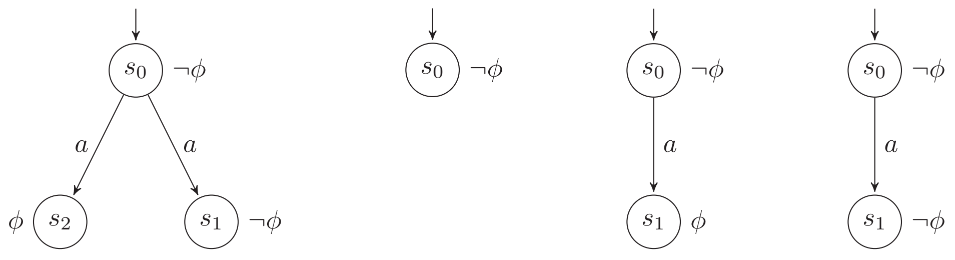


Figure 5: Differences between diamond and box modalities

2.5.2 Regular Formulas

Regular Formulas [J.F08, BK08] are useful to allow more than just a single action in modalities. For example, we are interested in saying that after two arbitrary actions, a specific action must happen. Regular formulas are based on *action formulas*. Action formulas are defined by the following BNF grammar:

$$\alpha \ ::= \ a_1 \mid \ldots \mid a_n \ \Big| \ \mathit{true} \ \Big| \ \mathit{false} \ \Big| \ \overline{\alpha} \ \Big| \ \alpha_1 \cap \alpha_2 \ \Big| \ \alpha_1 \cup \alpha_2$$

Here the formula $a_1 \mid \ldots \mid a_n$ defines a set with only the set of multi-actions $a_1 \mid \ldots \mid a_n$ in it. The formula *true* represents the set of all actions, and the formula *false* represents the empty set. The connectives $\cap$ and $\cup$ represents the intersection and the union of sets of actions respectively. For example, the formula $\langle true\rangle\langle a\rangle true$ expresses that an arbitrary action followed by the action 'a' can occur. Similarly, the formula $[true]false$ expresses that no action can be done.

Regular formulas extend the action formulas to allow use of sequences of actions in modalities. Regular formulas are defined by the following BNF grammar (α is an action formula):

$$R \ ::= \ \epsilon \ \mid \ \alpha \ \mid \ R_1 \cdot R_2 \ \mid \ R_1 + R_2 \ \mid \ R^* \ \mid \ R^+$$

In the above, ϵ is the empty sequence of actions. The formula $R_1 \cdot R_2$ represents the concatenation of actions in R_1 and R_2. For example, $\langle send \cdot receive\rangle true$ is the same as $\langle send\rangle\langle receive\rangle true$. Both express that a sequence of a *send* followed by a *receive* can be performed. The formula $R_1 + R_2$ represents the union of actions in R_1 and R_2. For example, the formula $[send \cdot send + receive \cdot receive]false$ expresses that neither the sequence $send \cdot send$ nor $receive \cdot receive$ can be performed. The regular formula R^* denotes zero or more repetitions of the sequences in R, and the formula R^+ denotes one or more repetitions of the sequences in R. For example, $\langle send^*\rangle true$ expresses that any sequence of the *send* action is possible, and $[send^+]true$ expresses that the *send* action must be done at least once.

Using regular formulas, we can formulate two commonly used modalities, namely *always* and *eventually*. The always modality, denoted by $\Box\phi$, expresses that ϕ holds in all reachable states. The eventually modality, denoted by $\Diamond\phi$, expresses that there is a sequences of actions that leads to a state in which ϕ is valid. These two modalities can be written as follows:

$$\Box\phi = [true^*]\phi \quad \Diamond\phi = \langle true^*\rangle\phi$$

Note that the *always* modality can be used to formulate safety properties of a system. For example, the property "*there exists no deadlock in any reachable states*" can be formulated as:

$$[true^*]\langle true\rangle true$$

Similarly, the safety property "*it is impossible to do two send actions without a receive*" can be formulated as:

$$[true^* \cdot send \cdot \overline{receive}^* \cdot send]false$$

Liveness properties can be formulated using *eventually* modality. For example, the property "*after every send, the message can be eventually received*" can be expressed as:

$$[true^* \cdot send]\langle true^* \cdot receive\rangle true$$

2.5.3 Fixed Point Modalities

In the previous section, we have described regular expressions and showed how expressive and suitable they are for expressing most behavioral properties. This section describes a much more expressive language that is called *modal μ-calculus* [And94]. The modal language is obtained by adding the *minimal* and *maximal* fixed point operators to Hennessy-Milner logic. The syntax of this language is obtained by the following BNF grammar:

$$\phi \ ::= \ \mathit{true} \ | \ \mathit{false} \ | \ \neg\phi \ | \ \phi_1 \wedge \phi_2 \ | \ \phi_1 \vee \phi_2 \ | \ \langle a \rangle \phi \ | \ [a]\phi \ | \ \phi_1 \rightarrow \phi_2 \ | \ \mu X.\phi \ | \ \nu X.\phi \ | \ X$$

Here the first seven modalities are the Hennessy-Milner logic. The formula $\phi_1 \rightarrow \phi_2$ is the same as the formula $\neg\phi_1 \vee \phi_2$. The formula $\mu X.\phi$ is the minimal fixed point, and the formula $\nu X.\phi$ is the maximal fixed point, where the variable X is the fixed point variable.

To explain the minimal and maximal modalities, let's consider the fixed point variable X as a set of states. We say that the formula $\mu X.\phi$ is valid, if for all those states in the smallest set X, the equation $X = \phi$ is satisfied (X occurs in ϕ). For example, the formula $\mu X.X$ denotes the smallest set of states X that satisfies $X = X$, which is obviously the empty set. This means that $\mu X.X$ is not valid for any state. This is equivalent to saying that $\mu X.X = \mathit{false}$.

Unlike the minimal fixed point operator, the formula $\nu X.\phi$ is valid for all those states in the largest set of states X that satisfies $X = \phi$. As an example, the formula $\nu X.X$ denotes the largest set of states X that satisfies $X = X$, which is obviously the set X itself. This means that $\nu X.X$ is valid for all states. This is equivalent to saying that $\nu X.X = \mathit{true}$.

Another way of understanding minimal and maximal fixed points is by considering the formula as a graph to be traversed, where the fixed point variables are states and modalities are seen as transitions. A formula is true if it can be made true by passing a finite number of times through the minimal fixed point variables, whereas it is allowed to traverse an infinite number of times through the maximal fixed point variables.

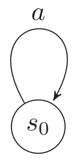


Figure 6: Differences of the minimal and maximal fixed point operators

As an example consider the transition system given in Figure 6. Here the formula $\mu X.\langle a \rangle X$ is invalid, because the a transition can not be traversed finite number of times. Since the a transition can be traversed infinite number of times, therefore, $\nu X.\langle a \rangle X$ is valid.

It is possible to translate regular formulas to modal μ-calculus. The following illustrates the translation of regular formulas containing '$*$' and '$+$':

$$\langle R^*\rangle\phi = \mu X.(\langle R\rangle X \ \vee \ \phi) \qquad [R^*]\phi = \nu X.([R]X \ \wedge \ \phi)$$
$$\langle R^+\rangle\phi = \langle R\rangle\mu X.(\langle R\rangle X \ \vee \ \phi) \qquad [R^+]\phi = [R]\nu X.([R]X \ \wedge \ \phi)$$

For example, the regular formula representing the eventually modality $\Diamond\phi = \langle true^*\rangle\phi$ can be formulated as follows:

$$\mu X.(\langle true\rangle X \ \vee \ \phi)$$

Sometimes a stronger property is required, namely that ϕ will eventually become valid along *every* path. This property can be formulated using the minimal fixed point operator as follows:

$$\mu X.([true]X \vee \phi)$$

Strictly speaking, this formula will also become true for paths ending in a deadlock, because in such a state $[true]X$ is also valid. To avoid this anomaly, the absence of deadlock must be explicitly mentioned. That is:

$$\mu X.(([true]X \wedge \langle true\rangle true) \vee \phi)$$

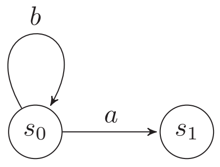
A variation of this is the action 'a' must be unavoidably be done, provided that there is no deadlock before the action a. That is:

$$\mu X.[\overline{a}]X$$

In order to express that the action 'a' must be done anyhow, the possibility of deadlock must be explicitly excluded. That is:

$$\mu X.([\overline{a}]X \wedge \langle true\rangle true)$$

The last two formula is not valid for the transition system given in Figure 7. This is because, the action 'b' can occur infinite often which can avoid the action 'a' to occur. The formula $\mu X.[\overline{a} \ \vee \ \langle a\rangle true]$ is valid in this transition system. Therefore, Figure 7 distinguishes between the last formula and the two before that.



$$\begin{tikzcd} s_0 \arrow[loop above, "b"] \arrow[r, "a"] & s_1 \end{tikzcd}$$

Figure 7: Differences between the last three formulas

The minimal and maximal fixed point operators can be combined to express *fairness* properties. Fairness properties can express that some action must happen, provided it is unboundedly often enabled, or because some other action happens only a bounded number of times. For instance, the formula:

$$\mu X \cdot \nu Y([b]X \ \vee \ [a]Y)$$

expresses that each sequence consisting of a and b actions ends in an infinite sequences of a's. This is because the X variable can only be traversed finitely, while the variable Y can be traversed infinitely often.

Note that we can also use data in modal formulas. For example, the formula:

$$[true^* \cdot \exists n : N \cdot error(n)]\mu X([\overline{shutdown}]X \wedge < true > true)$$

expresses that whenever an error with some number n is observed, a shutdown is inevitable.

2.6 mCRL2 Toolset

Micro Common Representation Language 2 [J.F08, GMR$^+$07] (mCRL2) is the name of a specification language that is used to specify and model check distributed and parallel systems. mCRL2 is based on ACP algebra that is extended to include abstract data types and time. It uses algebraic operators to construct systems containing very complex processes with lots of parallelism. The constructed system can then be model checked by exploring all possible system states in a brute force manner. This guarantees the system's correctness. Correctness of a system involves showing that the modeled system doesn't exhibit undesirable properties (*safety* properties) or it does exhibit desirable ones (*liveness* properties).

Around 1980 many specification languages were developed to model and verify the behavior of reactive systems. The most well-known of all are LOTOS, FDR refinement checker, SPIN, and μCRL. Language of Temporal Ordering Specification [BB87] (LOTOS) was initially developed by Brinskma for the formal description of the OSI (Open Systems Interconnection), but later extended to model concurrent systems. It is based on both CSP and CCS process algebras. Failure-Divergence Refinement (FDR) [BR00] is a refinement checker based on CSP process algebra that was originally developed by Formal Systems (Europe) Ltd. Instead of a model checker, the FDR toolset is called a refinement checker, because it translates CSP process expressions into labeled transition systems, and then determines whether one of the transition systems is a refinement of the other within the specified semantic model (traces, failure, failures divergence, and ...). Simple Promela Interpreter [Hol03] (SPIN) is a general model checker tool that was originally developed by Gerald Holzmann at Bell Labs. Systems to be verified are written in Promela language which supports asynchronous distributed systems and non-deterministic automata. Linear Temporal Logics (LTL) [MP92, Pnu77] are used to formulate system properties. The specification language μCRL [GP90] is a predecessor of the mCRL2 language which was originally developed by Groote and Ponse. It is a specification language based on ACP processes algebra, but without any support for abstract data types.

Unfortunately, the use of abstract data types made these languages hard to use when it came to the specification of complex systems. To this reason μCRL was extended with those data types that one would expect when writing specifications. Examples are boolean, numeric numbers, lists, sets, bags, functions and functional data types. The central notion of mCRL2 is the linear process. A linear process is a process expression from which all parallelism has been removed. Systems containing hundreds of thousands process expressions can be translated into a single linear process, which can then be used for analysis and verification. Model checking in mCRL2 is provided using Parameterized Boolean Equation System (PBES). Given a modal μ-calculus formula that specifies a desired behavior of a system, and a linear process, a PBES can be generated. The solution to this PBES indicates whether a formula holds on the processes or not.

As an example of a mCRL2 program, consider the following program that specifies a vending machine which dispenses a cup of tea for one inserted coin, or a cup of coffee for two inserted coins.

```
act
   insCoin, accCoin, coin;
   insTea, accTea, tea;
   insCoffee, accCoffee, coffee;

proc
   CS = ( insCoin . accTea + insCoin . insCoin . accCoffee ) . CS;
   VM = accCoin . (insTea + accCoin . insCoffee) . VM;

init
   allow( {coin, tea, coffee},
   comm( {insCoin|accCoin-> coin, insTea|accTea-> tea, insCoffee|accCoffee-> coffee},
     CS || VM ) );
```

The above specification defines nine atomic ACP actions representing payments and dispensing tea and coffee, two process expressions (CS and VM) representing a customer and the vending machine, and a parallel composition of these processes ($CS \ || \ VM$) representing the whole system. The keyword *allow* specifies those actions that are not going to be blocked (actions not in H for the ACP operator ∂_H), and the keyword *comm* defines the communication function (γ).

A visualization of the specified system can also be obtained in mCRL2 by converting the linear process into a labeled transition system. Figure 8 illustrates the labeled transition system of the vending machine program given above.

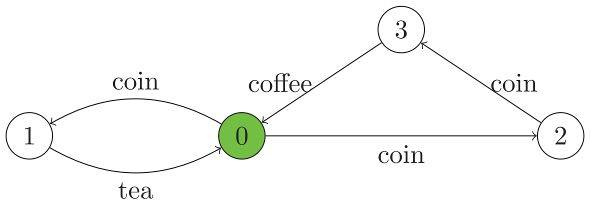


Figure 8: Labeled transition system generated by mCRL2 toolset for the vending machine program

Chapter 3

Implementation of the Generalized Alternative Construct

3.1 Overview

This chapter describes the design and implementation of the *generalized alternative construct* for Erasmus programming language. Erasmus is a CSP-like programming language that is based on processes and their interactions using synchronous channels. Synchronous channels ensure that the execution of a write action by a sender process is synchronized with the execution of the corresponding read by a receiver process. As an example, consider the following code:

```
protocol prot = {x:Integer; y:Integer; ↑sum:Int}

process adder = p:+prot{
  x:Integer := p.x;
  y:Integer := p.y;
  p.sum := x+y;
}
process user = q:−prot{
  q.x := 54;
  q.y := 66;
  scrln(q.sum);
}
cell main = {C:prot; adder(C); user(C);}
main();
```

The first line defines a protocol which illustrates the structure of messages that can be transmitted between a client and a server process (see Chapter 4). The protocol *prot* defines a message with three fields: x, y, and *sum*. The above code also defines two processes (*user* and *adder*) that communicate with one another through their ports (channel ends). A process is either a *client* or a *server* with respect to a channel. A process having the port with sign "−" is called a client process (e.g., *user*), and with sign "+" is called a server process (e.g., *adder*). Direction of messages with no symbols (e.g., x and y) are from clients to servers, and with the symbol "↑" (e.g., ↑*sum*) are from servers to clients. The code also defines a *cell* where the synchronous channel C is constructed, the ports of the two processes are linked, and the two processes are executed in parallel.

Erasmus includes a construct, called **select**-construct, to provide a choice between communicating on different channels. The **select** construct – inspired by CSP-alternative construct[Bro84] – is a generalization of the familiar if-then-else statement which provides a process to non-deterministically choose between several different communicating actions (send or receive). As an example, the process:

```
process P = q1:+prot; q2:+prot{
 x : Integer;
 select{
   || x := q1.x;  ...
   || x := q2.x;  ...
 }
}
```

tests whether the environment is willing to send to this process a value on port q_1, or a value on port q_2. Each **select** construct may have several branches (separated by ||), and a branch is chosen according to which communication takes place. In case where the environment offers more than one communication, the **select** construct ensures that only one is chosen.

Each branch of the **select** construct may have a boolean guard. As an example, in the following code:

```
select{
  |n≥ 0| x := q1.x;
  |n≤ 0| x := q2.x;
}
```

communication on port q_1 is enabled if the value of n is greater than or equal to zero. Similarly, communication on port q_2 is enabled if the value of n is less than or equal to zero.

In addition, a **select**-construct may also have an *orelse* branch which is executed when all the guards of other branches are disabled, or all the channels for which the process is selecting are terminated. For example, a process performing:

```
select{
  |n>0| x := q1.x;
  |n<0| x := q2.x;
  | orelse |  scrln("Error: the value of n is zero")
}
```

selects the *orelse* branch if and only if either the value of n is equal to zero, or the channels connected to port q_1 and q_2 are both terminated. In the case where all of the guards are disabled, and there is no *orelse* branch, the **select**-construct throws an exception.

In the original CSP, each alternative could only perform receive operations on channels, and only one end of a channel could participate in alternation. These restrictions make the implementation of the **select** construct considerably easy. However, it can be useful in a number of situations to combine both inputs and outputs at the same time, and to allow both ends of a channel to participate in an alternation. As an example, Figure 9 shows a system of processes, intended to implement the *Chain of Responsibility* pattern[GHJV95]. At the left, process G is a generator that generates problems and sends them to a sequence of solvers, $S_1; S_2; \ldots, S_n$. A solver S_i receives a problem on its query port (upper left). If it solves the problem, it sends the answer back to the generator on its answer port (lower left). If it cannot solve the problem, it forwards it to the next solver using its upper right port. At the end of the chain, there is a terminator process, T, which receives a message only when all of the solvers have failed; it sends a "failed" message back to the generator.

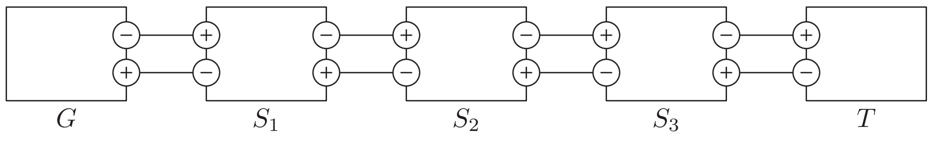


Figure 9: The chain of responsibility pattern with processes

The following code shows an Erasmus version of the chain of responsibility. The generator maintains a collection of problems (Integer IDs of the problems) in a stack to be passed to solver processes. Similarly, the terminator maintains a collection of failed problems in a stack to be passed to the generator.

```
protocol prot={pId:Integer; solved:Integer;  failed : Integer}

process generator = upR:−prot; lowR:+prot{
 counter: Integer:=0;
 loop select{
   |not stackA.empty()| upR.pId:=stackA.pop(); counter+=1;
   |counter>0| scrln(lowR.solved+"solved"); counter−=1;
   |counter>0| scrln(lowR.failed+"failed"); counter−=1;
   }
}
process solver = upL:+prot; lowL:−prot; upR:−prot; lowR:+prot; id:Int{
 loop select{
  ||  p: Integer := upL.pId;
          if  p % 2=0
          then lowL.solved := p;
          else upR.pId := p;
  ||  lowL.solved := lowR.solved;
  ||  lowL.failed := lowR.failed;
 }
}
process terminator = upL:+prot; lowL:−prot{
 loop select{
  ||  stackB.push(upL.pId);
  |not stackB.empty()| lowL.failed:=stackB.pop();
 }
}
cell  controller  = {
 q0,q1,q2,q3,a0,a1,a2,a3 :  prot;
 generator(q0,a0);
 solver(q0,q1,a0,a1,2);
 solver(q1,q2,a1,a2,3);
 solver(q2,q3,a2,a3,5);
 terminator(q3,a3);
}
```

Unfortunately, it is difficult to find an efficient implementation for the **select** construct that overcomes the above restrictions. Inefficient implementations include those that use global information (a central coordinator), require an unbounded amount of time, or use an unbounded amount of communication [BS83].

The goal of this chapter is, therefore, to present a design and an implementation for the **select** construct that overcomes the above restrictions. Particularly, we are aiming for a design with no central controller, and that doesn't employ additional channels internally. In order to come up with a correct design, we use process algebra to model and validate our design. In particular, we use ACP algebra (Algebra of Communicating Processes) to design our models, and we use mCRL2 model checker to validate the correctness of our design. Using ACP and mCRL2 for modeling and validating is important for two main reasons. First, ACP models are more abstract and we believe that they are easier to understand than codes. Second, mCRL2 performs exhaustive state space exploration, and so detects all errors in the design. In addition, when errors are found, mCRL2 generates counter examples of minimal length showing how the correctness of the model is violated.

The rest of this chapter is structured as follows. Section 3.2 describes the related work. In Section 3.3 we give a quick overview of our initial design. Section 3.4 describes the final design of the **select** construct. Particularly, in Section 3.4.1 we describe the design for the bi-directional synchronous channels, and in Section 3.4.2 we describe the design of processes performing selection. Section 3.5 extends our model to include closing of synchronous channels. Section 3.6 describes the model of non-**select** processes (processes performing regular send or receive operations). Section 3.7 discusses the priorities that may be imposed on the **select**-construct. Section 3.8 explains the implementation of the **select**-construct, and provides test cases. Finally, Section 3.9 summarizes.

3.2 Related Work

Buckley and Silberschatz [BS83] were the first to propose a protocol that avoids all three of the inefficiencies above, but their protocols are prone to deadlock for cyclic communication patterns [Kna92].

Knabe was the first to discover a deadlock-free protocol [Kna92]. His solution was based on a two-phase algorithm which uses asynchronous (buffered) messages to implement the (synchronous) generalized alternative construct for many-to-many channels. His protocol requires six or seven control messages in the general case, and creates an extra process for each channel.

Demaine proposed a deadlock-free protocol for implementing the generalized alternative construct that can achieve optimal number of message cycle per user-level communication [Dem98]. He proposed *fan channels* as a mean for one-to-many (or many-to-one) communications. Fan channels allow much higher efficiency than general many-to-many communications. Many-to-many channels, however, can be implemented using two fan channels, but they require an extra process in between.

Welch et al. [Wel10] implemented the **select** construct, within the JCSP library (CSP for Java) [Wel00]. Their implementation makes use of a single (system-wide) Oracle server process that includes a barrier branch to allow multi-way synchronization. Alts that use only input branches can be implemented without the Oracle. This is a pragmatic solution, but has the disadvantage of the Oracle potentially being a bottleneck.

Brown et al. [Bro07] developed a library for C++, called C++CSP, that provides easy mechanisms for concurrent C++ programming using CSP primitives. C++CSP follows the model captured by Occam and JCSP, with an API similar to the latter [OB04]. C++CSP was the first that introduced *poisonable channels* for the simple and safe shutdown of networks or sub-networks. These channels were later implemented also in JCSP.

Lowe [Low11] implemented the generalized **select** construct within the Scala programming language [Suf08]. His solution was based on passive uni-directional channels where active processes use to synchronize. His algorithm, however, is subject to the following two restrictions: 1. If a shared channel is involved in selection, it must not simultaneously be read or written by a non-select process. 2. A selection may not have two simultaneously enabled branches using the same channel.

3.3 Initial Design

Our initial design [GJ10] for the synchronous channels and the **select**-construct was based on Knabe's protocol in the sense that we used asynchronous (buffered) messages to implement synchronous channels and the **select** construct. In the later stage of our research, it turned out that our design is not an appropriate model for the following reasons:

1. We considered channels as active-entities that can take active roles in performing communications. For this reason, at least three processes were required for a single communication. We believe that considering channels as passive-entities can have a significant effect on the efficiency of the **select** construct.

2. For particular cases, e.g., cyclic communication patterns, our algorithm required extensive handshaking. This has a negative impact on the efficiency of our protocol in general.

3. Our algorithm was designed specifically for communication between distributed processes through *network channels*. However, we believed that for processes located on the same machine a more efficient algorithm can be obtained.

4. We didn't consider the closing of channels within this model. Considering the closing of channels is essential in the implementation of the **select** construct.

5. For configurations consisting of several processes, an existence of one slow process, treated fairly, can slow down the entire system.

For the above reasons, we developed another model for both the synchronous channels and the **select** construct that we will be presenting in this chapter. For readability reason, we present our initial model in the Appendix A, which is given at the end of this thesis.

3.4 The design of the select construct

Let us begin clarifying our assumptions. In most of the communication models cited previously, channels are usually viewed as active objects which can meditate between input and output requests. Our aim in the Erasmus implementation is, however, to implement channels as passive objects which serve as rendezvous points and can be transmitted from one process to another. In addition, since a process doesn't leave a selection until at least one communication is possible (or all channels are closed), it is possible for processes to deadlock or starve depending on how the programmer has structured use of the **select**.

We now present the structure of the design. We show first the design of synchronous bi-directional channels, and then turn into the role of processes.

3.4.1 The Channel Side

In this section we present the design for synchronous channels. For simplicity, we do not consider the closing of channels within this model. We begin by describing the model informally, before presenting the ACP model and the analysis.

Channels are *passive-objects* that accept procedure calls from processes and proceed through various states. Channels are connected to two or more processes, and their task is to recognize when they have received two complementary matches. A process that is willing to communicate sends a *request* to one or more of its channels. The channel maintains a list of requests it receives from processes. Upon receiving a *request* from a process, P_1 say, the channel, C say, tries to find a match. A match for a request of a process is a request of another process that is willing to perform a complementary action over the same field of the channel. Depending on the match, the channel C performs the following actions:

- If C doesn't find any matches, it responds to P_1 with a 'No', but saves P_1's request before going back to its initial state.

- If C finds a match that is sent by a process performing regular send or receive operation (a *committed* match), C responds to P_1 with a 'Yes', and removes the matching request from its list before going back to its initial state.

- If there exists a match that is sent by a process, say P_2, performing selection (a *half-committed* match), C sends a *commit* message to P_2, requesting the process to permanently commit itself to this communication and cancel others. The channel then waits for a reply. If P_2 replies with a 'Yes', C removes P_2's request from its list, and responds with a 'Yes' to P_1. If P_2 replies with a 'No', C removes P_2's request, saves P_1's request, and replies to P_1 with a 'No'. Finally, if P_2 replies with a 'TryAgain', C responds to P_1 with a 'TryAgain' and goes back to its initial state.

Processes performing selection may also release their half-commitments from channels. Releasing a half-commitment illustrates that the half-committed process has successfully synchronized with another channel. This is done by receiving the *release* signal. When a channel receives a *release* signal from a process, it removes all the requests that it has received by that process, before going to its initial state.

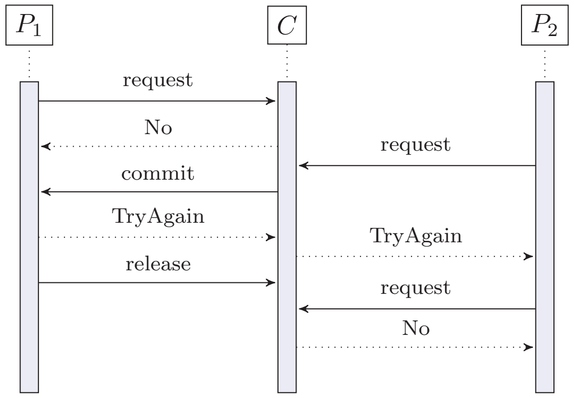


Figure 10: Sequence diagram of the channel's behavior

Figure 10 demonstrates a scenario where channel C connects process P_1 (performing selection) to process P_2 (performing regular send or receive). Since process P_1 is first at rendezvous, it receives a 'No' from channel C to its request. When C receives the request of P_2 (second at rendezvous), it sends a commit signal to P_1 to check whether or not P_1 can commit itself to this channel. When P_1 releases its half-commitment, channel C responds with a 'No' to the request of P_2, because there exists no match for P_2's request anymore.

ACP Model

We now present the ACP model (using mCRL2 syntax) that captures the behavior of channels informally described above. We define the following data types to indicate the identities of processes, channels, branches, and field numbers:

```
sort −− defining mCRL2 datatypes
 pID = p(1)|···|p(n) −−IDs of processes
 fID = f(1)|···|f(n) −−IDs of fields
 chID = ch(1)|···|ch(n) −−IDs of channels
 brID = br(1)|···|br(n) −−IDs of branches
```

We can further define the following data types to indicate the status of channels, responses, and the type of operations (regular send or receive, or **select** construct) processes may perform:

```
sort
 chStatus = ACTIVE | TERMINATED
 Resp = YES | NO | TRYAGAIN | CLOSED
 Opr = SEND | RECEIVE | SELECT_SEND | SELECT_RECEIVE
```

A *request* message is a fixed-size block of data containing five fields:

1. The unique ID of the process sending the request
2. The ID of the branch within the **select** statement
3. The ID of the channel the request is sent to
4. The field number for which the process is willing to communicate
5. The type of operation the process is willing to perform.

For this reason, we define the following data type to illustrate the structure of request messages:

```
sort MSG = struct msg( pId:pID, brId:brID, chId:chID, fId:fID, opr:Opr )
```

As an example, the following:

$r = \texttt{msg}(\ p(1),\ br(1),\ ch(1),\ f(1),\ \texttt{SELECT\_SEND}\ )$

indicates a request message that is sent from process P_1 which is willing to send a value (by performing selection) through field f_1 of channel C_1. Having r defined above, the following holds:

$\texttt{pId}(r) = p(1)$, $\texttt{brId}(r) = br(1)$, $\texttt{chId}(r) = ch(1)$, $\texttt{fId}(r) = f(1)$,
$\texttt{opr}(r) = \texttt{SELECT\_SEND}$

The following defines the ACP channels that models communication between components (Erasmus channels and processes). Every ACP channel is defined by three parameterized actions with the same action name: a send action (ending with !) that represents a send to the channel, a receive action (ending with ?) that represents a receive from the channel, and an atomic action that represents the synchronized execution of the shriek and the query actions.

```
act
 request!, request?, request: chID×MSG
 reqResp!, reqResp?, reqResp: chID× Resp
 commit!, commit?, commit: pID×chID×MSG
 commitResp!, commitResp?, commitResp: pID×Resp
 release!, release?, release: chID×MSG
 notify!, notify?, notify: pID×chID
```

We now consider the definition of a channel. Listing 3.1 illustrates the ACP process *Channel*(*me, reqList, status*) that represents a channel with identity *me*, where *reqList* and *status* are a list for storing requests and the channel's status respectively. In its initial state, a channel either receives a *request*, or a *release* signal. In case of a *request* signal, the channel goes through different states depending on its status and the match it may find. The channel responds with a 'No' if it can't find any matches, and responds with a 'Yes' if it finds a committed match. When a channel finds a *half-committed* match, it proceeds as the ACP process *ChannelCommit* to decide whether or not the matching process can commit itself to this communication. When a channel receives a *release* signal, it removes the requests of the querying process before returning to its initial state.

Listing 3.1: ACP specification of the Erasmus channel

```
Channel(me:chID, reqList:List(MSG), status:chStatus) =
  $\sum_{req:\mathbf{MSG}}$ · request?(me, req)·
  (status == ACTIVE) → (
     Let match = findMatch(reqList, req)·
     (match == null)→
        reqResp!(pId(req), NO)·
        Channel(me, add(reqList,req), status)
     ◇( opr(match) == SEND || opr(match) == RECEIVE) →
         reqResp!( pId(req), YES )·
         notify!(pId(match), me)·
         Channel(me, remove(reqList,match), status)
     ◇( opr(match) == SELECT_SEND || opr(match) == SELECT_RECEIVE) →
         ChannelCommit(me, reqList, status, req, match) )
  ◇( status == TERMINATED ) → ( ··· To be implemented ···)
+
  $\sum_{req:\mathbf{MSG}}$ · release?(me, req)·
  Channel(me, remove(reqList,req), status)
```

Listing 3.2 captures the behavior of the process *ChannelCommit*. When this process is executed, it sends the matching process a commit signal, asking whether the process can commit itself to this channel and ignore the rest.

Listing 3.2: ACP specification of Channel Commit

```
ChannelCommit(me:chID, reqList:List(MSG), status:chStatus, req:MSG,
    match:MSG) =
  commit!(pId(match), match)·
  $\sum_{resp:\mathbf{Resp}}$· commitResp?(me, resp)·
  (resp == YES) →
       reqResp!(pId(req), YES)·
       Channel(me, remove(reqList, match), status)
  ◇(resp == NO) →
       reqResp!(pId(req), NO)·
       Channel(me, add(reqList, req)), status)
  ◇(resp == TRYAGAIN) →
       reqResp!(pId(req), TRYAGAIN)·
       Channel(me, reqList, status)
  ◇ error· $\delta$;
```

Note that in the above listings *findMatch, add*, and *remove* are helper functions to find a match, to add an element to a list, and to remove an element from a list. The action *error* followed by deadlock action is defined and used only for specification purposes.

3.4.2 The process side

The process protocol is somewhat different in structure from the channel side. Each process is composed of two components: the *main process* and the *handler*. The main process is an *active-object* that will be executing the user code (including the code of the **select** statement), and its role is to send requests to the channels of its branches and execute the selected branch. The handler is a *passive-object* that helps the main process to select a particular branch, and its role is to handle *commit* signals.

Main process A main process P controls the execution of the program, and follows a procedure when it enters a selection. Its first step is to select a branch, say b_i, and send a send or receive request (*half-committed* request) to the channel involved in b_i; then it waits for the channel's response. The channel may reply with a 'No', 'TryAgain', or a 'Yes', depending on its status and the match it may find. If P receives a 'No' or a 'TryAgain', it continues sending request to the remaining channels involved in selection, but keeps the request it has just sent. If P receives a 'Yes' from the channel, it sends *release* to all of the channels it has received 'No' (releasing its half-commitment), and executes the branch b_i.

If P receives 'No' or 'TryAgain' from all of the channels involved in selection, it enters the second phase of its protocol. In the second phase, P pauses for a short while before resending requests to those channels it has received 'TryAgain'. P repeats this procedure until it either receives a 'Yes' from a channel, or 'No' from all channels. In case of the former, P releases its half-commitments from channels it has received ‘No’, and performs the actual data transfer. In case of the later, P waits until it is awoken by its handler.

Handler The task of a handler is to help the main process to perform a selection. A handler keeps track of the status of the main process, and accepts *commit* signals from channels. When a handler receives a *commit* signal from a channel, say C, it enters a procedure to appropriately respond to it. If the main process is sending requests, the handler responds to C with a ‘TryAgain’. A ‘TryAgain’ message indicates that the process is busy right now, and the channel can try again later. If the main process is waiting or pausing, the handler responds to C with a ‘Yes’, and awakes the main process, passing the identity of channel C (along with the identity of the branch associated with C). At this time, the main process releases its half-commitment from other channels, and performs actual data transfer through channel C.

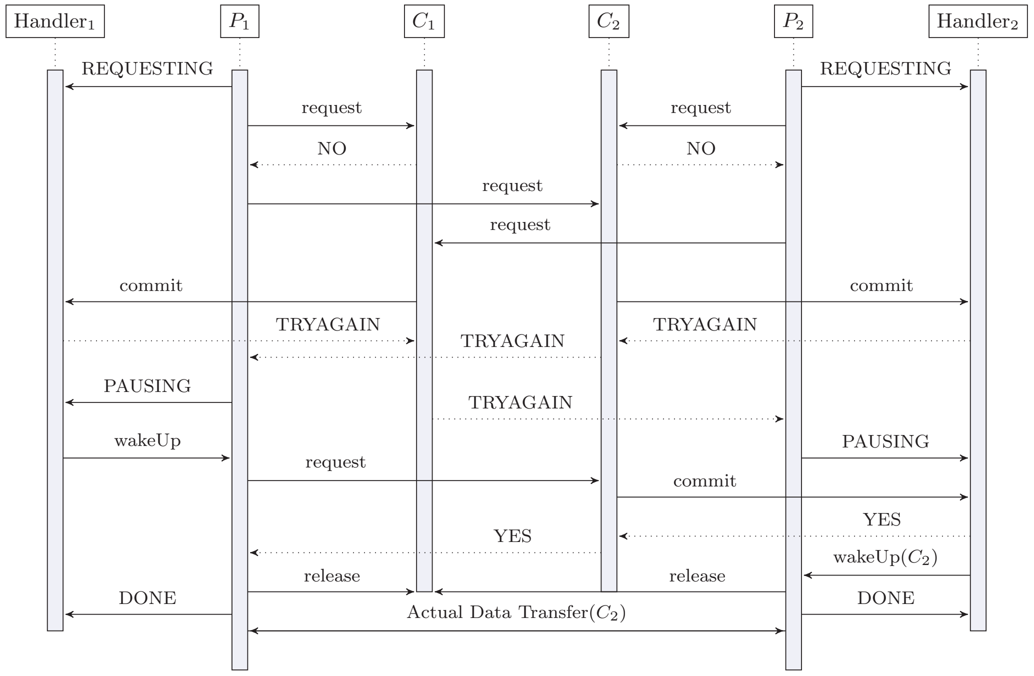


Figure 11: Cooperation of processes P_1 and P_2 with their handlers

Figure 11 illustrates an example of how processes cooperate with their handlers in order to select a particular branch. In this example, both processes P_1 and P_2 are willing to perform selection on both channels C_1 and C_2. Both processes receive negative responses from the first channel to which they send request. Since they both receive ‘TryAgain’ responses from the second channel to which they send request, they both pause for a short while before trying again. Assuming that the pausing time for P_1 is shorter than P_2, P_1 receives a positive response from channel C_2 (after trying again), and hence, both process will agree upon communicating on channel C_2.

ACP Model

We now present the ACP model that captures the behavior of the main process and the handler informally explained above. We start by defining the following data types representing the status of processes and the type of messages for which the handler wakes up the main process.

```
sort
 pStatus = REQUESTING | WAITING | PAUSING | DONE
 wakeUpFlag = PAUSEOVER | SELECTED
```

The following defines the ACP channels that processes and their handlers use to change the status of the process and to wake up the main process.

```
act
 setStatus!, setStatus?, setStatus: pID×pStatus
 wakeUp!, wakeUp?, wakeUp: pID×MSG×wakeUpFlag
```

Model of the main process We now consider the definition of the main process. Listing 3.3 illustrates the ACP process *select*(*me, branches*) that represents a process with identity *me*, where *branches* is a list of request messages[1] for which the process is performing selection. In its initial state, the select process informs its handler by changing its status to 'Requesting', and then behaves as the process *SelectRequest* to send requests to the channels involved in selection.

Here, *toRequest* is a list of branches for which the process hasn't sent requests yet, *toTryAgain* is a list of branches for which the process has received 'TryAgain', and *toRelease* is a list of branches for which the process has received 'No'. The process *select_request* starts its execution by sending request messages (half-commitment) to each of the channels of its branches, and depending on the response it may receive it goes through different states. When a response to a request is a 'Yes', the process behaves as the ACP process *SelectDone* (after changing its status to 'Done') to release its half-commitment from other channels and perform the actual data transfer. When a response to a request is a 'No', the process keeps the request in the *toRelease* list and continues sending requests to other channels. When a response to a request is a a 'tryAgain', the process keeps the request in the *toTryAgain* list and continues with other branches. In a case, where the process is not matched with any other processes (*to_request* is empty and *to_tryAgain* is not empty), it behaves as the process *selectTryAgain* to pause for a short time before resending the request messages. Finally, if all the channels respond with NO, the process then acts as the process *selectWait* to perform the wait command until it hears back from its handler.

[1]Recall that each message contains the following five fields: process ID, branch ID, channel ID, field ID, and the type of operation the process is willing to perform.

Listing 3.3: ACP specification of **select** construct

```
Select(me: pID, branches: List(MSG)) =
   setStatus!(me, REQUESTING)· SelectRequest(me, branches, branches, [], []);

SelectRequest(me:pId, branches:List(MSG), toRequest:List(MSG),
     toTryAgain:List(MSG), toRelease:List(MSG)) =
  (toRequest·size() != 0) :→ (
     Let req := toRequest[0] ·
     request!(chId(req), req) ·
     $\sum_{resp:\mathbf{Resp}}$· reqResp?(chId(req), resp)·
     (resp == YES) :→
        setStatus!(me, DONE)·
        SelectDone(me, req, toRelease)
     ◇ (resp == NO) :→
        SelectRequest(me, branches, remove(toRequest,req), toTryAgain,
             add(toRelease,req))
     ◇ (resp == TRYAGAIN) :→
        SelectRequest(me, branches, remove(toRequest,req),
             add(toTryAgain,req), toRelease)
     ◇ error· $\delta$ )
  ◇ (toTryAgain·size() != 0) :→
        SelectTryAgain(me, branches, toTryAgain, toRelease)
  ◇  SelectWait(me, branches, toRelease);
```

Listing 3.4 illustrates the definition of ACP processes *SelectDone* and *Execute*. The main process releases its half-commitments from other channels, before performing the actual data transfer. When a process is about to perform the actual data transfer, it signals the environment that it is executing a particular branch. This is illustrated by the ACP actions *send_signal* and *receive_signal*. Like the *error* action, ACP actions *send_signal* and *receive_signal* are used for specification purposes.

Listing 3.4: ACP specification of the *SelectDone* and *Execute*

```
SelectDone(me:pID, toRelease:List(MSG), selectedBranch:MSG) =
 (toRelease·size()!=0) :→
    Let req:=toRelease[0] ·
    release!(chID(req), req) ·
    SelectDone(me, remove(toRelease, req))
 ◇  Execute(me, selectedBranch);

Execute(me:pID, selectedBranch:MSG)=
 (opr(selectedBranch)==SEND || opr(selectedBranch)==SELECT_SEND) :→
   send_signal(me, chId(selectedBranch), fId(selectedBranch))·
   transfer!(chID(selectedBranch))
 ◇ transfer?(chId(selectedBranch))·
    receive_signal(me, chId(selectedBranch), fId(selectedBranch));
```

Listing 3.5 illustrates the behavior of the main process when it is willing to resend requests to those channels it has received ‘TryAgain’. The parameterized action *pause* is defined to illustrate the pausing act of a process. When a process pauses, it receives from its handler either a signal indicating that the pausing time is over, or a signal indicating that a particular branch has been selected.

Listing 3.5: ACP specification of the *SelectTryAgain*

```
SelectTryAgain($me$:pID, $branches$:List(MSG), $toTryAgain$:List(MSG),
     $toRelease$:List(MSG)) =
  setStatus!($me$, PAUSING)$\cdot$
  pause($me$)$\cdot$(
     wakeUp?($me$, $null$, PAUSEOVER)$\cdot$
      $selectRequest(me, branches, toTryAgain, [], toRelease)$
    +
     $\sum_{selectedBranch:\mathbf{MSG}}\cdot$ wakeUp?($me$, $selectedBranch$, SELECTED)$\cdot$
     $selectDone(me, selectedBranch, toRelease)$
);
```

Finally, when the main process receives negative responses from all of the channels, it waits until it hears back from its handler. Listing 3.6 illustrates this idea:

Listing 3.6: ACP specification of **select** construct

```
SelectWait($me$:pID, $branches$:List(MSG), $toRelease$:List(MSG)) =
  setStatus!($me$, WAITING)$\cdot$
  $\sum_{selectedBranch:\mathbf{MSG}}\cdot$ wakeUp?($me$, $selectedBranch$, SELECTED)$\cdot$
   SelectDone($me$, $selReq$);
```

Model of the handler We now consider the definition of the handler. Listing 3.7 illustrates the *processHandler* with identity *me*, where *status* is the status of the main process. In its initial state, a handler can receive either a *setStatus* from the main process, or a *commit* signal from a channel. In case of the former, the handler goes through different states, depending on the status of the process. In case of the later, the handler responds appropriately as follows: If the main process is busy sending requests, the handler responds with a 'TryAgain' to the commit signals it may receive. If the main process is pausing or waiting, the handler responds with a 'Yes' to the first commit signal, wakes up the the main process, and sends 'No' to subsequent commit signals. Finally, if the main process is done selecting a branch, the handler then responds with a 'No' to every commit signals it may receive.

Listing 3.7: ACP specification of handler

```
processHandler(me: pID) = Handler(me, null);

Handler(me:pID, status:pStatus) =
 ∑_{st:pStatus} · setStatus?(me, st)·
  (st == REQUESTING || st == DONE) :→ Handler(me, st)
  ◇(st == PAUSING) :→ handlerPause(me)
  ◇(st == WAITING) :→ handlerWait(me)
  ◇ error· δ
+
  ∑_{req:MSG,ch:chID} · commit?(me, ch, req)·
  (status == REQUESTING) :→
       commitResp!(chId(req), TRYAGAIN)·
       Handler(me, st)
  ◇(status == DONE ) :→
       commitResp!(chId(req), NO)·
       Handler(me, status)
  ◇ error· δ;
```

If the main process is pausing, the handler sends to the main process either a signal indicating that the pausing time is over, or a signal indicating that a branch is selected due to a commit signal. Listing 3.8 captures this idea:

Listing 3.8: ACP specification of handlerPause

```
handlerPause(me: pID) =
   wakeUp!(me, null, PAUSEOVER)·
   Handler(me, REQUESTING)
+
   ∑_{selectedBranch:MSG} ·  commit?(me, selectedBranch)·
   commitResp!(chId(selectedBranch), YES)·
   wakeUp!(me, selectedBranch, SELECTED)·
   Handler(me, DONE);
```

Finally, when the main process is waiting, the handler sends to it either a signal indicating that a branch is selected, or a signal indicating that all the channels for which the process is waiting are closed. Listing 3.9 captures this idea:

Listing 3.9: ACP specification of handlerWait

```
handlerWait(me: pID) =
 ∑_{selectedBranch:MSG} · commit?(me, selectedBranch)·
 commitResp!(chId(selectedBranch), YES)·
 wakeUp!(me, selectedBranch, SELECTED)·
 Handler(me, DONE);
```

3.4.3 Validating the Model

This section considers a configuration of processes and channels, and analyzes them using mCRL2 to make sure that they behave as the way we expect.

Figure 12 illustrates a configuration of processes P_1 and P_2 that are connected to each other through channels C_1 and C_2. Both processes are performing selection (each with four branches) on the two channels; that is: P_1 is willing to either send through fields f_1 of C_1 (or field f_2 of C_2), or to receive from fields f_2 of C_1 (or field f_1 of C_2). Similarly, P_2 is also willing to either send through fields f_2 of C_1 (or field f_1 of C_2), or to receive from fields f_1 of C_1 (or field f_2 of C_2). The arrows indicates the direction of data flows.

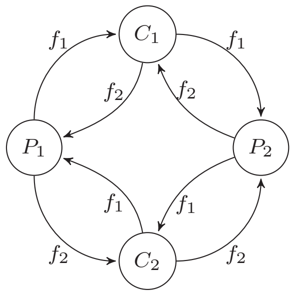


Figure 12: A simple configuration

Listing 3.10 illustrates the ACP specification of the whole system. In this specification, set H (actions to be blocked) contains all the send and receive actions (actions ending in "!" and "?"), and set I (actions to be hidden) contains all actions except *send_signal, receive_signal, pause* and *error*. In addition, the communication function for all shriek and query actions are defined as: $\gamma(action\_name!,\ action\_name?) = action$

Listing 3.10: ACP specification of the configuration given in Figure 12

```
Process(me:pID, ch1:chID, ch2:chID) =
 Select(me,
         [msg(me, SELECT_SEND, f(1), br(1), ch1),
          msg(me, SELECT_RECEIVE, f(2), br(2), ch1),
          msg(me, SELECT_SEND, f(2), br(3), ch2),
          msg(me, SELECT_RECEIVE, f(1), br(4), ch2)])·
 Process(me, ch1, ch2);

System = τ_I( ∂_H(
  Channel(ch(1), [], ACTIVE) ||
  Channel(ch(2), [], ACTIVE) ||
  Process(p(1), ch(1), ch(2)) || processHandler(p(1)) ||
  Process(p(2), ch(2), ch(1)) || processHandler(p(2)) ));
```

Processes should repeatedly agree upon which field of a channel to communicate, and no *error* action should occur. Figure 13 illustrates the safety and liveness properties in modal formulas written in mCRL2. The first two properties illustrate that the *System* shouldn't deadlock and no *error* actions should occur. The third property illustrates that every send message by a process, should eventually be received by another process. Finally, the last property illustrates that a process should not pause for ever.

---

1. *Absence of deadlock*:

   `[true*]<true> true`

2. *Absence of* `error`:

   `[true*.error] false`

3. *Every send can eventually be received*:

```
nu X.[true] X &&
(forall p:pID, c:chID, f:fID. [send_signal(p, c, f)] mu Y. <true>true &&
    exists q:pID. val(q!=p) ⇒!receive_signal(q, c, f)] Y)
```

4. *After every* `pause`, *the process eventually performs a communication*:

```
nu X.[true]X &&
(forall p:pID.[pause(p)] mu Y. <true>true && <!pause(p)>Y)
```

Figure 13: Safety and liveness properties

When we use mCRL2 to test if *System* satisfies all the properties given in Figure 13, the test succeeds for all properties except for the last one. Failure of the last property indicates that *System* can *diverge*. Divergence can happen, because process P_1 and P_2 can perform actions at about the same time. Figure 14 illustrates a scenario where the system diverges. In this scenario both P_1 and P_2 send their half-commitment *requests*, receive *tryAgain* messages, and *pause* at the same time (for the same amount of time), and this pattern is repeated. In the implementation, the pause will be of a random amount of time, to ensure that the symmetry is eventually broken (with probability 1).

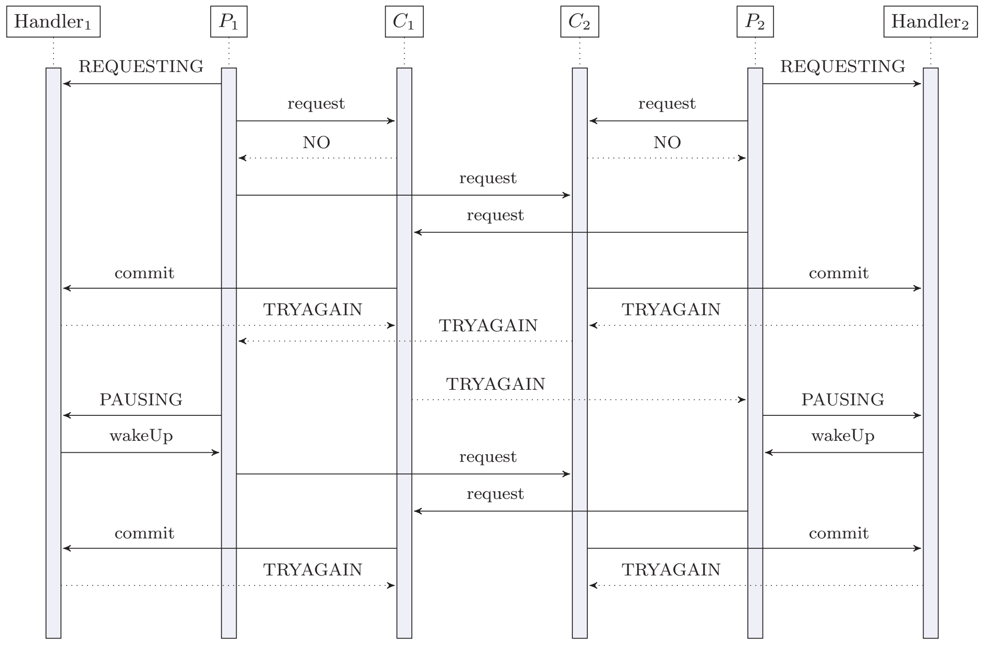


Figure 14: Sequence diagram of an execution where divergence occurs

3.5 Closing of Synchronous Channels

The previous sections explained the design and presented the model for the **select** construct by explaining the main functionality of processes and synchronous channels. This section extends our model to capture the feature of closing of channels. We start by describing the model informally, and then we present the ACP model and the analysis.

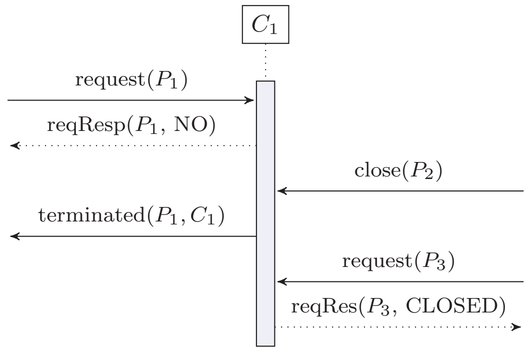


Figure 15: Sequence diagram of an execution where channel C_1 is closed

Each channel has a status which is set to either *active* or *terminated*. A channel can be terminated by sending to it a *close* signal. When a channel receives a *close* signal, it notifies all processes (their handlers to be exact) that are waiting for it, and responds with a 'Closed' signal to the subsequent requests it may receive. Figure 15 illustrates this idea.

Each handler also keeps track of the number of closed channels. When a main process is about to do a wait, it sends its handler a list of channels for which the process is about to wait. The process then waits for a response. The handler responds with a boolean that indicates whether or not all the channels have been terminated. If not, the process waits, otherwise the process either executes the *orelse* branch (if there is one), or throws an exception. Note that channels can also be closed while the main process is waiting. In this case, the handler wakes up the main process, and informs it about the termination of all channels. Figure 16 clarifies this idea by illustrating an example where process P is willing to perform selection on channels C_1 and C_2. Since both channels become terminated, the process should executed its *orelse* branch or throws an exception.

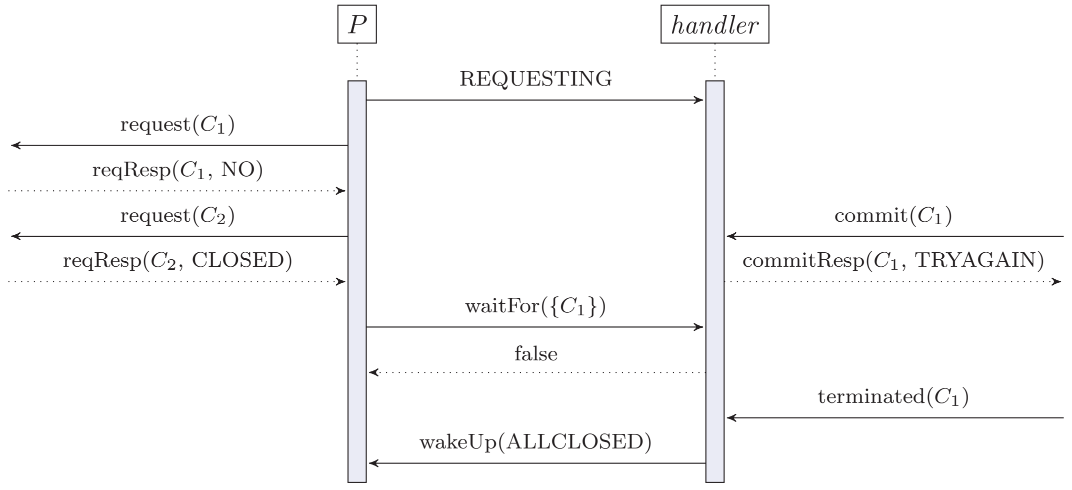


Figure 16: Handler wakes up the main process when all channels are closed

3.5.1 ACP Model

We now present the ACP model that captures the feature of closing of channels that was explained informally above. We first extend the data types to capture the status of channels, responses, and the type of messages for which the handler wakes up the main process.

```
sort
 chStatus = ACTIVE | TERMINATED
 Resp = YES | NO | TRYAGAIN | CLOSED
 wakeUpFlag = PAUSEOVER | SELECTED | ALLCLOSED
```

Further we define the ACP channels (communicating actions) that the main process uses to close a channel, the channel uses to inform processes that it is terminated, and the main process and the handler use to interact with one another.

```
act
 close!, close?, close : chID ×pID
 terminated!, terminated?, terminated: pID ×MSG
 waitFor!, waitFor?, waitFor: pID ×Set(chID)
 waitForResp!, waitForResp?, waitForResp: pID ×Bool
```

Listing 3.11 illustrates the ACP specification of the channel. The definition of the *Channel* is mostly similar as before, so we just describe the main differences here. In its initial state, a channel can receive a *request* signal, a *release* signal, or a *close* signal. The channel responds with a 'CLOSED' to every request it receives, if and only if its status is set to 'TERMINATED'.

Listing 3.11: ACP specification of Channels

```
Channel(me:chID, reqList:List(MSG), status:chStatus) =
  $\sum_{req:\mathbf{MSG}}$ · request?(me, req)·
  (status == ACTIVE) :→  ( ··· Same As Before ··· )
  ◇(status == TERMINATED) :→
        reqResp!(pId(req), CLOSED)·
        Channel(me, reqList, status)
+
  $\sum_{req:\mathbf{MSG}}$ · release?(me, req)·
  Channel(me, remove(reqList, req), status)
+
  $\sum_{p:\mathbf{pID}}$ · close?(me, p)·
  ChannelClose(me, reqList);
```

When a channel receives a 'close' signal, it behaves as the process *ChannelClose* to notify all the processes waiting for this channel before setting its status to 'TERMINATED'. Listing 3.12 illustrates the ACP specification of the process *ChannelClose*.

Listing 3.12: ACP specification of ChannelClose

```
ChannelClose(me:chID, reqList:List(MSG)) =
  (reqList·size() != 0) :→
        terminated!(pId(reqList·0), me)·
        ChannelClose(me, remove(reqList, reqList·0))
  ◇ Channel(me, [], TERMINATED);
```

Listing 3.13 illustrates the ACP specification of the processes *Select* and *SelectRequest*. The definition of these processes are same as before. The only difference is that after sending a request signal, the main process may receive a 'CLOSED' signal from a channel. At this point, the process ignores this channel (because it is terminated), and continues sending requests to other channels.

Listing 3.13: ACP specification of **select** construct

```
Select(me: pID, branches: List(MSG)) = ⋯

SelectRequest(me:pId, branches:List(MSG), toRequest:List(MSG),
     toTryAgain:List(MSG), toRelease:List(MSG)) =
  (toRequest·size() != 0) :→ (
     Let req := toRequest[0] ·
     request!(chId(req), req) ·
     $\sum_{resp:\mathbf{Resp}}$· reqResp?(chId(req), resp)·
     (resp == YES) :→ ⋯
     ◇ (resp == NO) :→ ⋯
     ◇ (resp == TRYAGAIN) :→ ⋯
     ◇ (resp == CLOSED) :→
          SelectRequest(me, branches, remove(toRequest, req), toTryAgain,
               toRelease)
     ◇ error· $\delta$ )
  ◇ (toTryAgain·size() != 0) :→ ⋯
  ◇  SelectWait(me, branches, toRelease);
```

Listing 3.14 illustrates the ACP specification of the process *SelectWait* that captures the behavior of the main process before performing a wait. When a main process is about to do a wait, it sends its handler a list of channels for which the process is about to wait. This list contains only the address of those channels from which the process has received a negative response. The process then waits for a response. The *true* response means that all the channels for which the process is willing to communicate have been closed, so the process signals the environment by performing the atomic action *allclosed_signal* and terminates its execution. However, if the process receives a *false*, it then waits until it is awoken by the handler. During waiting, the process may receive either a *wakeUp* signal with the 'SELECTED' flag, or a *wakeUp* signal with the 'ALLCLOSED' flag. In case of the former, the process has been successfully synchronized with another process, so it behaves as the ACP process *SelectDone* to finish its execution and perform the actual data transfer. In case of the later, the process signals the environment about the termination of all the channels. Note that like *error* action, the atomic action *allclosed_signal* is defined and used only for specification purposes.

Listing 3.14: ACP specification of **select** construct

```
SelectWait(me:pID, branches:List(MSG), toRelease:List(MSG)) =
  waitFor!(me, toRelease)·
  $\sum_{b:\mathbf{Bool}}$· waitForResp?(me, b)·
  (b == false):→
    setStatus!(me, WAITING)·
    $\sum_{selectedBranch:\mathbf{MSG}}$·(
      wakeUp?(me, selectedBranch, SELECTED)·
      SelectDone(me, selectedBranch)
    +
      wakeUp?(me, null, ALLCLOSED)·
      allclosed_signal(me) )
  ◇ allclosed_signal(me);
```

Figure 3.15 illustrates the definition of the handler. Here, *waitForCHS* is a list of channels the main process waits for, and *closedCHS* is a list of all closed channels. In its initial state, a handler can now receive four messages: a *setStatus* and a *waitFor* from the main process, or a *commit* and a *terminated* message from a channel. We have already explained the behavior of the handler when it receives a *setStatus* or a *commit* message. When it receives a *waitFor* message, the handler returns a boolean indicating whether all the channels for which the main process is going to wait have been already terminated or not. When a handler receives a *terminated* signal from a channel, it saves the address of this channel (in *closedCHS*), and either wakes up the main process (if all channels are closed and the main process is waiting), or goes back to its initial state until it receives another signal.

The behavior of the *handlerPause* is exactly the same as in Figure 3.8, so we won't explain it here anymore. The definition of the *handlerWait* is slightly different. When the handler is behaving as the ACP process *handlerWait*, it can receive either a *commit* signal, or a *terminated* signal from a channel. In case of the former, the handler replies with a 'Yes' and wakes up the main process passing the address of the selected channel. In case of the later, the handler saves the address of the terminated channel, and either wakes up the main process if all the channels have been closed, or waits until it receives another *commit* or *terminated* signal.

Listing 3.15: ACP specification of handler

```
processHandler(me:pID) = Handler(me, null, {}, {});

Handler(me:pID, status:pStatus, waitForCHS:Set(chID),
      closedCHS:Set(chID)) =
 $\sum_{st:\mathbf{pStatus}}$· setStatus?(me,st)·
  (st==REQUESTING || st==DONE) :→ Handler(me,st, waitForCHS, closedCHS)
  ◇(st == PAUSING) :→ handlerPause(me, waitForCHS, closedCHS)
  ◇(st == WAITING) :→ handlerWait(me, waitForCHS, closedCHS)
  ◇ error· δ;
+
  $\sum_{S:\mathbf{Set(chID)}}$·  waitFor?(me, S)·
  waitForResp!(me, (S == closedCHS))·
  Handler(me, status, S, closedCHS)
+
  $\sum_{req:\mathbf{MSG},ch:\mathbf{chID}}$· commit?(me, ch, req)·
  (status == REQUESTING) :→
        commitResp!(chId(req), TRYAGAIN)·
        Handler(me, status, waitForCHS, closedCHS)
  ◇(status == DONE ) :→
        commitResp!(chId(req), NO)·
        Handler(me, status, {}, {})
  ◇ error·δ;
+
  $\sum_{ch:\mathbf{chID}}$· terminated?(me, ch)·
  (waitForCHS == (closedCHS ∪{ch}) && (status==PAUSING ||
       status==WAITING)) :→
        wakeUp!(me, null, ALLCLOSED)·
        Handler(me, status, waitForCHS, add(closedCHS,ch))
  ◇ Handler(me, status, waitForCHS, add(closedCHS,ch))

handlerPause(me:pID, waitForCHS:Set(chID), closedCHS:Set(chID)) =
   wakeUp!(me, null, PAUSEOVER)·
   Handler(me,REQUESTING,waitForCHS, closedCHS)
+
   $\sum_{req:\mathbf{MSG}}$· commit?(me, req)·
   commitResp?(chId(req), YES)·
   wakeUp!(me, req, SELECTED)·
   Handler(me, DONE, reqCHS, closedCHS);

handlerWait(me:pID, waitForCHS:Set(chlID), closedCHS:Set(chID)) =
 (waitForCHS == closedCHS) :→
        wakeUp!(me,null,ALLCLOSED)·
        Handler(me,DONE,waitForCHS,closedCHS)
 ◇($\sum_{req:\mathbf{MSG}}$· commit?(me, req)·
     commitResp!(chId(req), YES)·
     wakeUp!(me, req, SELECTED)·
     Handler(me, DONE, waitForCHS, closedCHS)
   +
     $\sum_{ch:\mathbf{chID}}$· terminated?(me, ch)·
     handlerWait(me,waitForCHS,add(closedCHS,ch)) );
```

3.5.2 Validating the Model

We now analyze the model of closing channels. We use the same configuration as in Figure 12 with only one difference. The process *Terminator* is added to the *System*, which terminates channels C_1 and C_2 after a short period (1 second). The whole *System* is defined in Listing 3.16.

Listing 3.16: ACP specification of a system with a *Terminator* process

```
Process(me:pID, ch1:chID, ch2:chID) =
 Select(me,
        [msg(me, SELECT_SEND, f(1), br(1), ch1),
         msg(me, SELECT_RECEIVE, f(2), br(2), ch1),
         msg(me, SELECT_SEND, f(2), br(3), ch2),
         msg(me, SELECT_RECEIVE, f(1), br(4), ch2)])·
 Process(me, ch1, ch2);

Terminator(me:pID, ch1:chID, ch2:chID) =
  pause(me)@1· close!(ch1, me)· close!(ch2, me);

System = τ_I( ∂_H(
  Channel(ch(1), [], ACTIVE) || Channel(ch(2), [], ACTIVE)
  || Process(p(1), ch(1), ch(2)) || processHandler(p(1))
  || Process(p(2), ch(2), ch(1)) || processHandler(p(2))
  || Terminator(p(3), ch(1), ch(2))
)) ;
```

Here, set I also contains the action *allclosed_signal*. Therefore, processes should repeatedly agree upon which field of a channel to communicate, and eventually they should signal the environment that all the channels are closed. This is captured by the following modal formula:

```
mu Y. nu X.([true]Y ||
(<allclosed_signal(p(1))>X && <allclosed_signal(p(2))>)X))
```

Figure 17: Liveness property of the system in the μ-calculus

Using mCRL2 shows that the *System* satisfies the above property, and thus, it behaves as we expect.

3.6 Non-select Processes

The previous sections explained the design of the **select** construct, where processes chooses between different available communications. This section explains the behavior of processes performing regular send or receive operations.

A process P performing regular send or receive operation, starts its execution by sending a *fully-committed* request to the channel, say C, it is willing to communicate. P then waits for a response. If C replies with a 'Yes', then P performs the actual data transfer. If C replies with a 'No', then P waits until the communication takes place by another process. If C replies with a 'TryAgain', then P pauses for a short while before trying again. Finally, if C replies with a 'Closed', then P throws an exception. Note that processes performing non-select communication never receive *commit* signals, and therefore do not have handlers.

3.6.1 ACP Model

We now present the ACP model that captures the behavior of non-select processes that was explained informally in the previous section.

Listing 3.17 illustrates the ACP process *nonSelect* that captures the behavior of non-**select** processes. The process starts its execution by sending a request to the channel it is willing to communicate. The process then waits for the reply. If the channel replies with a 'Yes', then the process behaves as the ACP process *Execute* to perform the actual data transfer. If the channel replies with a 'No', then the process behaves as the ACP process *processWait* to wait until it is awoken by the channel. When the process waits, it either receives a *notify* signal, or a *wakeUp* signal from the channel. In case of the former, the process is successfully synchronized with another process, so it continues to perform the actual data transfer. In case of the later, the process is notified because the channel has been closed. Therefore, it signals the environment by performing the atomic action *allclosed_signal*, and terminates its execution.

Listing 3.17: ACP specification of non-**select** processes

```
nonSelect(me:pID, req:MSG) =
 request!(chId(req), req)·
 ∑_{resp:Resp}: reqResp?(me, resp)·
 (resp == YES) :→ Execute(me, req)
 ◇(resp == NO) :→ processWait(me, req)
 ◇(resp == TRYAGAIN) :→ pause(me)· nonSelect(me, req)
 ◇(resp == CLOSED) :→ allclosed_signal(me)
 ◇error· δ;

processWait(me:pID, req:MSG) =
 notify?(me, chId(req))· Execute(me, req)
 +
 wakeUp?(me, null, ALLCLOSED)· allclosed_signal(me);
```

Note that in order to capture the notion of closing of channels properly, we need to modify the definition of synchronous channels. Listing 3.18 illustrates the modified version of the process *channelClose*. A closed channel always sends the closed notification to the main process of a non-**select** process, not to its handler.

Listing 3.18: ACP specification of ChannelClose

```
ChannelClose(me:chID, reqList:List(MSG)) =
  ( reqList·size() != 0) :→
        (opr(reqList·0) == SELECT_SEND || opt(reqList·0) ==
             SELECT_RECEIVE) →
                 terminated!(pId(reqList·0), me)·
                 ChannelClose(me, remove(reqList, reqList·0))
        ◇
                 wakeUp!(pId(reqList·0), null, ALLCLOSED)·
                 ChannelClose(me, remove(reqList, reqList·0))
  ◇ Channel(me, [], TERMINATED);
```

3.6.2 Validating the Model

In this section we validate the model of non-**select** processes. Figure 18 illustrates a simple configuration of processes, where the client processes P_2 and P_3 are willing to repeatedly communicate with the server process P_1 through channels C_1 and C_2 respectively. The server process is performing selection on the two channels. There is also a terminator process that terminates both channels after a short period (1 millisecond).

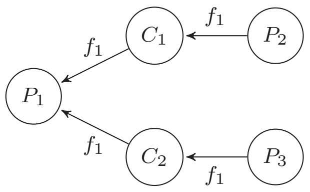


Figure 18: A simple configuration

Listing 3.19 illustrates the ACP specification of the whole system. Using mCRL2 shows that the the three processes communicate with one another, and all will issue the *allclose_signal*. The *System* indeed satisfies the properties given in Figure 13 and Figure 17. Therefore, the whole system behaves as expected.

Listing 3.19: ACP specification of a simple configuration

```
Client(me:pID, ch:chID) =
 nonSelect(me, msg(me, br(1), ch, f(1), SEND))· Client(me, ch);

Server(me:pID, ch1:chID, ch2:chID) =
 Select(me, [
        msg(me, br(1), ch1, f(1), SELECT_RECEIVE),
        msg(me, br(1), ch2, f(1), SELECT_RECEIVE)
        ])· Server(me, ch1, ch2);

Terminator(me:pID, ch1:chID, ch2:chID) =
  pause(me)@1· close!(ch1, me)· close!(ch2, me);

System = Channel(ch(1), [], ACTIVE) || Channel(ch(2), [], ACTIVE)
         || Client(p(2), ch(1)) || Client(p(3), ch(2))
         || Server(p(1), ch(1), ch(2)) || ProcessHandler(p(1))
         || Terminator(p(4), ch(1), ch(2));
```

3.7 Priority

In this section we investigate the priorities that may be imposed on the **select** construct. Erasmus allows each **select** construct to select its branches according to three different policies: *fair*, *random*, and *ordered*. The *fair* policy ensures that the branches of a **loop select** construct to be chosen fairly. The *random* policy doesn't put any restrictions on the selection, and the *ordered* policy ensures that the branches of a **loop select** construct to be chosen according to the order written by the programmer. As an example consider the following Erasmus code where the server process performs selection with the *ordered* policy:

```
protocol prot = {signal}
process client = p: −prot{
 loop{
   p. signal;
 }
}
process server = p:+prot, q:+prot{
 loop select ordered {
   || p. signal;
   || q. signal;
 }
}
cell main = {
 c1: prot; c2: prot;
 client(c1); client(c2);
 server(c1, c2);
 }
main()
```

Now, consider the implementation of the **select** construct presented in this chapter. Suppose that the two client processes run first: they both send their requests to channels C_1 and C_2 before the server process runs. When the server process runs, it first sends a request to C_1 (on port p), and receives a 'Yes'. Thus, the server chooses to communicate on port p, and therefore, server's priority is followed. However, if the server process runs first, then the choice is chosen depending on the client which sends its request message first.

The example above shows that the server process tries to follow its priority as much as possible by sending its first request to the branch that has the highest priority. To this reason, we provide each **select** construct with a *sorting mechanism* that sorts the branches (before sending requests) with respect to the given policy. That is: each branch is associated with an integer value representing the priority of the branch (lowest value represents highest priority). Before sending requests, the **select** construct associates and sorts the priorities as follows:

- Ordered policy: the priority of the first branch is set to 0, and the priority of the n'th branch is set to n. The process then sorts the branches only once in increasing order.
- Random policy: the priorities of all branches are set to 0. No sorting is required.
- Fair policy: the priorities of all the branches are set to 0. When a branch is selected, its priority will be increased by one. This allows the sort mechanism to put this branch at the end of the list, causing the **select** construct to send a request to the channel of this branch after all the other branches.

3.8 Implementation and Testing

We have implemented the behavior of the **select** construct, processes, and synchronous channels, explained in this chapter using the Java programming language. In our implementation, processes are threads that will be executing the code of the **select** construct, although at some point these threads will be within procedure calls to other components. Unlike processes, channels and handlers are implemented as passive objects (with one monitor each) that are willing to receive procedure calls from active processes.

The message passing between processes, handlers, and channels is implemented as procedure calls and their returns. For example, the *request* signal is implemented by a procedure called *request* and its response (*reqResp*) is implemented by the value returned from that procedure. Similarly, the pausing phase of processes is implemented as a procedure called *pause* with no return value, which causes the invoking process to sleep for a random amount of time (between 1 to 3 milliseconds in our implementation).

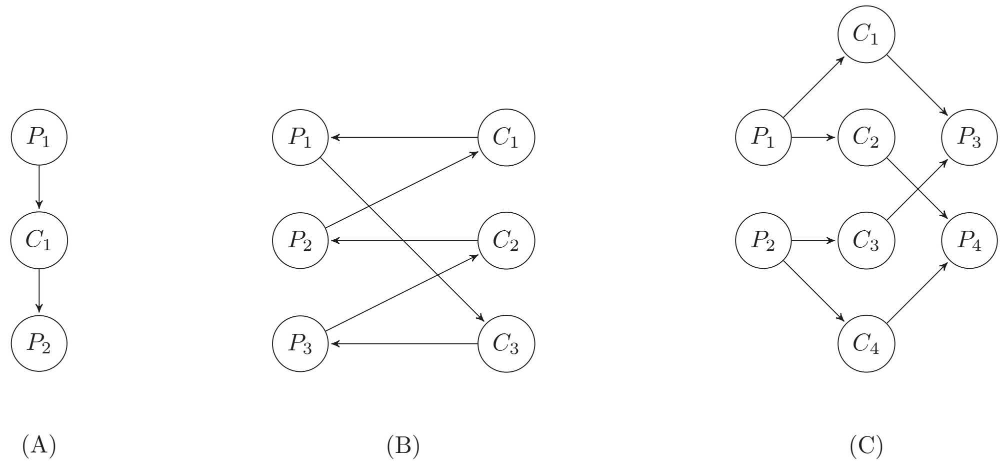


Figure 19: Test configurations

We have tested the implementation of the **select** construct on different configurations. The implementation seems robust and efficient. For example, the configuration given in Figure 12 with fair policy achieves more than 168,000 communications per second on a standard quad-core PC. In this configuration, processes are more likely to send their requests at the same time, leading to more pausing than in most other configurations. In this test, about 15% of the total time was spent in pausing phase.

We have built other configurations, including those in Figure 19. For each, we have used mCRL2 model checker to check that the system behaves as the way we expected. We have also tested the implementation for each of these configurations. Table 2 illustrates the results of such tests.

Configuration	Total messages (1 sec)	P_1	P_2	P_3	P_4	C_1	C_2	C_3	C_4	avg. pause
Fig 19.(A)	235,872	235,872	235,872	–	–	235,872	–	–	–	0%
Fig 19.(B)	299,158	190,439	204,989	202,887	–	92,270	108,718	94,169	–	13.3%
Fig 19.(C)	203,660	29,009	174,652	28,423	175,237	11,735	17,274	16,689	157,963	17.8%
Fig 12	167,846	167,846	167,846	–	–	167,450	396	–	–	14.8%
Fig 18	265,281	265,281	122,756	142,525	–	122,756	142,525	–	–	1%

Table 2: Test results

3.9 Summary

In this chapter we have described the implementation of the generalized alternative construct for the Erasmus programming language, using ACP models and mCRL2 model checker to develop a design that has all of the desired properties. The use of ACP and mCRL2 is invaluable in this work, and we believe that we would not have ended up with a correct design without them.

Our models remove previous restrictions on the use of alternative construct by allowing branches of an alternation to be guarded by both send and receive operations. Our models also remove the restriction that prevented both ends of a synchronous channel to participate in an alternation. In addition, we have also considered closing of channels, the orelse branch, and priorities that can be imposed on the select construct. We also performed several tests on different configurations. The test results gave very acceptance performance to our implementation, and greatly increased our confidence in our design.

We also have plans for developing the implementation of the alternative construct further. We would like to change the semantics of the alternative construct to also cover network channels, as is done in JCSP [Wel10] and CTJ [SHW00] (*Communicating Threads in Java*). Network channels allow processes to communicate with one another across the network. We believe that the extension to network channels would be very straight forward: the same design can be used with messages being sent across the network through a *broker* process. A broker will be a process that records the location of channel ends, the socket number of ports, ... In addition, we would like to extend the model to include *timeouts*, as is done in Scala programming language [Low11]. A Timeout branch is very similar to the *orelse* branch with the difference that this branch is selected if and only if the select construct won't choose a branch the in the specified time. Moreover, we would like to extend our design to also cover *barrier synchronizations*, as is done in JCSP [Wel10]. A barrier synchronization allows n processes to synchronize together, for arbitrary n.

Chapter 4

Safety of Client Server Communications

4.1 Overview

Similar to other process oriented languages, Erasmus follows the *client-server* relationships between processes: server processes usually offer some services to their clients, and may themselves act as clients to other servers. Similarly, client processes require some services and will obtain these services by sending requests to server processes. Figure 20 illustrates this idea by indicating a configuration where server processes P_2 and P_3 offer services to client processes P_1 and P_2 through synchronous channels C_1 and C_2 respectively.

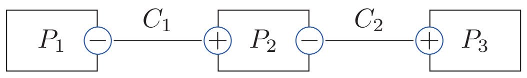


Figure 20: Client-server relationships between Erasmus processes

The client-server pattern has proved extremely useful when building complex process-oriented systems [MW97]. Server processes actually play the same role as objects in object oriented languages. Indeed, OO languages such as Smalltalk use message passing terminology to describe method calls between objects. However, process oriented servers avoid many of the concurrency problems endemic in OO languages, and their interfaces are more powerful: a client-server communication is a *conversation* consisting of several messages in both direction, not just a single request-response pair.

A client-server communication is said to be *safe*, if every message sent by the client is eventually received by the server process, and every response sent by the server is eventually received by the client. Safety property allows the construction of client-server systems of processes that are guaranteed to be free from deadlock and livelock properties. The problem is to check programs for safety. As much safety checking as possible should be done at compile-time (static-checking). Safety checking can also be done at run-time (dynamic-checking), but this is less desirable.

In the Erasmus programming language, the client-server communication pattern is implemented using a bidirectional synchronous channel. To ensure safety, channels and processes' ports (channel ends) are augmented with protocols. Protocols define the structure of messages and allow the patterns of communication between processes to be specified. Figure 21 illustrates a configuration where processes P_1, P_2, and the channel C_1 are augmented with protocols π_1, π_2, and π_c respectively.

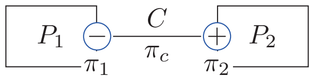


Figure 21: Client-server communications with protocols

Figure 21 identifies four positions at which safety checking is required. We say that safe communication is ensured if message sequences, or their descriptions in terms of protocols, are *compatible*; that is the program in Figure 21 is safe with respect to communications on channel C if:

1. The code implementing P_1 is compatible with protocol π_1,
2. The code implementing P_2 is compatible with protocol π_2,
3. π_1 is compatible with π_c.
4. π_2 is compatible with π_c.

The aim of this chapter is, therefore, to explore some general mechanisms and structures which can be used for specifying the client-server communications in Erasmus language. Particularly, we would like to define and implement protocol compatibility that can be served as a basis for static safety checking of client-server communications.

The rest of this chapter is structured as follows. Section 4.2 discusses the related work. Section 4.3 briefly describes the protocol specification facilities, and how they can be used to specify client-server interfaces. Section 4.4 gives operational semantics for Erasmus syntaxes and protocols in terms of labeled transition systems. Section 4.5 introduces the notion of *protocol satisfaction* as a mean to ensure protocol compatibility, and explains how safety of programs can be obtained using the satisfaction relation. Section 4.6 illustrates an algorithm that implements the satisfaction relation. Section 4.7 discusses the problems and solutions that we have encountered in designing Erasmus language. Finally, Section 4.8 summarizes.

4.2 Related work

Facilities for specifying client-server communications are presented in some other process oriented languages.

Client-server communications in occam-π [Pet05, Sam08] are implemented using a pair of unbuffered and unidirectional channels: one carries *requests* from the client to the server, and the other carries *responses* from the server to the client. The two channels are packaged inside a *channel bundle*. The order of messages permitted over each channel is specified using a protocol, but there exists no facilities for specifying the relationship between the two protocols in a client-server interface.

The draft occam 3 language specification [Geo98] described a *call channels* mechanism built on top of channel bundles. The declaration of call channels implicitly define protocols to carry the parameters and results of a procedure. Call channels make clients look like procedure calls, and servers look like procedure declarations. They are a useful abstraction for programmers switching from object oriented world, because they make calls to servers look like method calls upon objects. However, they only allow a single request and response, and they do not allow richer conversations between components.

The Honeysuckle language design provides facilities for easily compromising client-server systems, with interfaces being defined as *services* [Eas05]. Services provide a convenient and flexible way of specifying client-server interfaces. Of particular interests are *compound services*, which allow a server's behavior to be specified using a subset of Honeysuckle communication including sequence, choice, and repetition. However, it is possible to specify a protocol that cannot be statically verified by using repetition. Such protocols may require run-time checks to be inserted by the compiler.

There are some programming languages such as Haskell and Scala that use *session types* [Sim03, ADZ$^+$12] to specify client-server interfaces. Session types provide a formal approach to the problem of specifying the interactions between multiple processes. They allow communication to be defined as types. The type of a communication channel, therefore, describes the sequence of messages that may be sent across it. Neubauer and Thiemann [Mat04, SE08] describe an encoding of session types in Haskell's type system, representing communication operations using a continuation-passing approach. The specifications are applied to sequence of IO operations, such as communications on a network socket. However, there is no discussion of their application to local communication.

4.3 Protocols

Client-server communications are implemented in Erasmus using channels that are augmented with protocols. Channels are synchronous and bidirectional. Processes use ports to connect to channels. The signs + and − in the ports indicate whether the process is a client or a server with respect to a channel. A process having the port with sign $-$ is called a client and a process having the port with sign $+$ is called a server.

Data is sent through channels in the form of *messages*. A message has a *field* name and a *type*. In programs, messages are defined as $f : T$ where f is the field name and T is the type. The type of fields may be omitted, in which case the message is called a *signal*.

$$\mathbf{P} ::= \quad s \;\Big|\; \hat{r} \;\Big|\; q : T \;\Big|\; \hat{q} : T \;\Big|\; P_1 ; P_2 \;\Big|\; P_1 | P_2 \;\Big|\; P_1 + P_2 \;\Big|\; * P \;\Big|\; \# P \;\Big|\; (P)$$

Figure 22: Syntax of Erasmus protocols

The allowable sequence of messages that can be transmitted over a channel is specified by using *protocols*. Figure 22 illustrates the syntax of Erasmus protocols. Protocols are defined recursively. The base cases are *signals* and *messages*. The protocol s defines a query signal that is sent from a client to a server, and the protocol $\hat{r}$ defines a reply signal that is sent from a server to a client. Similarly, $q : T$ and $\hat{q} : T$ specify a query message and a reply message respectively, both of type T. The protocol $P_1; P_2$ specifies sequential composition, which illustrates a communication pattern that is defined by P_1 followed by P_2. The protocol $P_1|P_2$ specifies a *deterministic* choice, a visible choice that is made by the environment. The protocol $P_1 + P_2$ denotes a *non-deterministic* choice, an invisible choice that is made internally by the process. The protocol $*P$ denotes the "Kleene star" operator, and specifies a communication defined by P that can be performed zero or more times. Similar to the Kleene star, the protocol $\#P$ specifies a communication defined by P that can be performed any number of time, but at least once. Finally, (P) denotes grouping and shows that parentheses can be used in the usual way to override precedences. Without parentheses, the precedences go from "|" and "+" (lowest), ";", to "$*$" and "#" (highest).

As an example of protocols, consider the following code that specifies and implements a client-server interface to a random-number generator, which will attempt to roll an N-sided die for you, and either succeed or drop it on the floor.

Listing 4.1: The code of a random generator number

```
protocol DIE = {#(roll:Integer; (↑rolled: Integer | ↑dropped) )}

process Person = p:−DIE{
 p. roll := 6;
 n: Integer;
 select{
   || n := p. rolled ; scrln(" rolled an 'n'");
   || p.dropped; scrln("dropped the die");
   }
}
process Generator = p:+DIE{
 loop{
   n:Integer := p. roll ;
   select{
   || p. rolled := random(1,n);
   || p.dropped; exit;
   }
 }
}
cell main = c:DIE; Person(c); Generator(c);
```

Listing 4.1 illustrates an example where both processes' ports and the channel are augmented with the same protocol. The *DIE* protocol allows any client process (e.g., *Person*) to send the integer message *roll*, followed by a receive of either an integer message *rolled* or the signal *dropped*. Thus, in the perspective of a client process, one valid conversation would be *!roll;?rolled* and another one would be *!roll; ?dropped*, where the exclamation mark before a field name denotes a send operation, and the question mark denotes a receive operation. The same protocol, however, allows any server process (e.g., *Generator*) to receive the message *roll*, followed by a send of either the message *rolled* or the signal *dropped*. Therefore, in the perspective of a server, one valid conversation would be *?roll;!rolled*, and another would be *?roll; !rolled; ?roll; !dropped.*

Channels and processes' ports may also have different protocols. This approach is useful for two reasons. First, standard engineering practice is to separate specification and implementation: programmers are likely to be writing codes of clients (or servers) to somebody else's servers (or clients). Since each protocol specifies how a process interacts with respect to a channel, programmers should be able to write their codes without any knowledge of how the other programmers' code is written. Second, it enables us to decide when two processes can safely communicate using only their specifications.

As an example, consider the code of Listing 4.2 that specifies a client-server interface to a vending machine that provides its customers with a cup of tea for one inserted coin, or with a cup of coffee for two inserted coins.

Listing 4.2: The code of a vending machine with several protocols

```
protocol protCS1 = {coin; ↑tea}
protocol protCS2 = {coin; coin; ↑coffee}
protocol protVM = {#(coin; (↑tea | coin; ↑coffee))}

process CS1 = p:−protCS1{
 p.coin; p.tea;
}
process CS2 = p:−protCS2{
  p.coin; p.coin; p.coffee;
}
process VM = p:+protVM{
 loop{
  p.coin;
  select{
   || p.tea;
   || p.coin; p.coffee;
   }
  }
 }
cell main = c:protVM; VM(c); CS1(c); CS2(c);
```

In the above example, the behavior of processes *CS1*, *CS2*, and *VM* are specified by the protocols *protCS1*, *protCS2*, and *protVM* respectively. In addition, the behavior of the synchronous channel is specified by the protocol *protVM*. Although, the customer processes *CS1* or *CS2* do not use all of the fields specified by the channel's protocol (*protVM*), but the communications between the two clients and the vending machine are *safe*. Safety is obtained because every request that is sent by each customer can be received by the vending machine, and *vice-versa*.

Safety of client-server communications, however, can be violated in various situations. As an example, consider a case where the vending machine in Listing 4.2 is replaced by the process given in Listing 4.3.

Listing 4.3: The code of a vending machine with several protocols

```
process VM2 = p:+protVM{
 loop select{
  ||  p.coin; p.tea;
  ||  p.coin; p.coin; p.coffee;
 }
}
```

Replacing the process *VM* with *VM2* results in an *unsafe* communication, because the process *VM2* may deadlock: after the insertion of a coin by the customer *CS1*, the vending machine may choose its second branch. At this point the customer waits for a cup of tea while the vending machine waits for another coin to provide the customer with a cup of coffee.

4.4 Semantics of Programs

The previous section introduced the syntax of Erasmus programs and protocols, and explained how they can be used to specify client-server communications. To proceed further, we need some formal representation of processes and protocols. We assume that the behavior of processes and protocols are specified by *labeled transition systems*, which can fairly be described as the workhorses of concurrency theory.

Recall that a labeled transition system (Section 2.3) on a set of labels L is a structure $\mathscr{L} = \langle S, s_0, F, L, T\rangle$ where S is a set of states, s_0 is the initial state, F is the set of final states, and $T \subseteq S \times L \times S$ is the transition relation. We write $s_i \xrightarrow{\alpha} s_j$ for $(s_i, \alpha, s_j) \in T$. We assume, in standard fashion, that L contains the *silent* or *internal* action τ; transition $s_i \xrightarrow{\tau} s_j$ represents internal computational steps of a system, without reference to its environment.

4.4.1 Processes and Transition Systems

There is a procedure, described below, for constructing the transition system corresponding to the code of a process. The code of a process consists of simple and structured statements. States of the transition system corresponding to the code of a process are defined by Erasmus statements and sub-statements. Transition relations are defined by *inference rules*, showing how the program evolves from one state into another.

The main ingredient of statements that formalizes the behavior of processes are the atomic action *exit*, *variable assignments*, *send* and *receive* communication actions, *conditional* commands, *loops*, and *non-deterministic* choices. Figure 23 illustrates the formal syntax of the statements that specify the behavior of Erasmus processes.

$$
\begin{array}{lll}
\textbf{stmt} & ::= & \textbf{exit} \;\Big|\; x := \mathit{expr} \;\Big|\; p.f := \mathit{expr} \;\Big|\; x := p.f \;\Big|\; \mathit{stmt}_1 ; \mathit{stmt}_2 \;\Big| \\
 & & \textbf{case}\ |b_1| \rightarrow \mathit{stmt}_1 \ldots |b_n| \rightarrow \mathit{stmt}_n\ \textbf{end} \;\Big| \\
 & & \textbf{select}\ |b_1| \rightarrow p_1.f_1 := x_1 \ldots |b_n| \rightarrow x_n := p_n.f_n \ldots |\textbf{orelse}| \rightarrow \mathit{stmt}\ \textbf{end} \;\Big| \\
 & & \textbf{loop}\ \mathit{stmt}\ \textbf{end}
\end{array}
$$

Figure 23: Syntax of Erasmus statements

Before presenting the formal semantics, let us give some informal explanations on the meaning of the commands. The intuitive meanings of an assignment $x := expr$ is conventional: variable x is assigned the value of the expression *expr*. The meaning of *send* and *receive* operations is as follows: action $p.f := expr$ represents sending the value of the expression *expr* over the field f of port p. Similarly, action $x := p.f$ represents receiving a value (and assigning it to x) over the field f of port p. The statement $stmt_1; stmt_2$ denotes sequential composition. That is, $stmt_1$ is executed first and after its termination $stmt_2$ is executed.

The statement:

$$\textbf{case } |b_1| \rightarrow stmt_1 \ldots |b_n| \rightarrow stmt_n \textbf{ end}$$

stands for a *non-deterministic* choice between statements $stmt_1, \ldots, stmt_n$. That is, $stmt_1$ is executed if b_1 holds, or else $stmt_2$ is executed if b_2 holds, ..., or else $stmt_n$ is executed if b_n holds. This kind of choice is called non-deterministic, because the decision is made internally by the process and that the environment can neither control nor detect which branch is chosen by the process.[1]

Unlike the case statement, the statement:

$$\textbf{select } |b_1| \rightarrow p_1.f_1 := x_1 \ldots |b_n| \rightarrow x_n := p_n.f_n \ldots |\textbf{orelse}| \rightarrow stmt \textbf{ end}$$

stands for a *deterministic* choice between communication actions $p_i.f_i$ for which the guard b_i is satisfied in the current state. It is called a deterministic choice, because the environment can control the behavior of the process. That is; if more than one guard is satisfied in the current state, then the **select** construct chooses a branch ($p_i.f_i := x_i$ or $x_j := p_j.f_j$ where b_i or b_j is satisfied) according to which communication takes place. Note that if none of the guards $b_1, \ldots, b_n$ is satisfied in the current state, then the **select** construct chooses its **orelse** branch, if there is one, or throws an exception otherwise.

The statement:

$$\textbf{loop } stmt \textbf{ end}$$

models repetitive command which terminates upon executing the **exit** command. Note that a loop without the **exit** command doesn't terminate.

As an example of the statements introduced above, consider the vending machine given in Listing 4.4. The vending machine counts the number of cups available, and provides its customers with tea or coffee if and only if the number of cups are greater than zero. The machine returns the inserted coins and terminates its execution if no cups is left.

[1]We use deterministic in the sense of CSP, to mean a choice that is made by the environment on behalf of the process. Similarly, a choice is non-deterministic if it is made internally by the process, independently of the environment.

Listing 4.4: A vending machine that counts the number of available cups

```
protocol prot = { ∗(coin; ( ↑tea | coin; ↑coffee | ↑coin) ) }

process VM = p:+prot; max: Integer {
  ncup : Integer := max;
  loop{
   p.coin;
   case{
    |ncup>0|
      select{
        || p.tea;
        || p.coffee;
      }
      ncup := ncup − 1;
     |ncup=0|
      p.coin; exit;
} } }
```

Figure 24 illustrates the transition system of the vending machine given in Listing 4.4. In the starting location, the variable *ncup* is assigned the value associated with the variable *max*. The assignment is illustrated by the edge labeled with action τ that connects the state *VM* into the state *loop*. In the state *loop*, only one option is available and that is the insertion of a coin. This is illustrated by the edge labeled with the receive action *p?coin* that connects the state *label* to state *case*. In the state *case*, two options are available, depending on the value of *ncup*. If *ncup* is equal to zero, the transition system evolves into the state *select*. Otherwise, it evolves to state *p!coin;exit*. In the location *select*, two options are available which are illustrated by two edges labeled with send actions *p!tea* and *p!coffee*. From these two locations, controls goes back to the location *loop*. The location *p!coin;exit* explains the behavior of sequential composition. Thus, only one option is available, and that is the execution of *p!coin*. Upon the execution of the *exit* command the loop terminates. This is illustrated by the edge labeled with *exit* that connects the location *exit* to the location *terminate* where no other options is available.

The goal is, therefore, to formalize the idea sketched above. We start with a formal definition of sub-statements that are the potential locations of intermediate states during the execution of a program. For each set of sub-statements, we also define the *inference rules* that explains how a transition system evolves from one state into another.

Atomic Actions The set of sub-statements of a Erasmus-statement **stmt** is defined recursively. For statements $stmt \in \{exit, x := expr, p.f := x, x := p.f\}$, the set of sub-statements is:

$$sub(stmt) = \{stmt,\ terminate\}$$

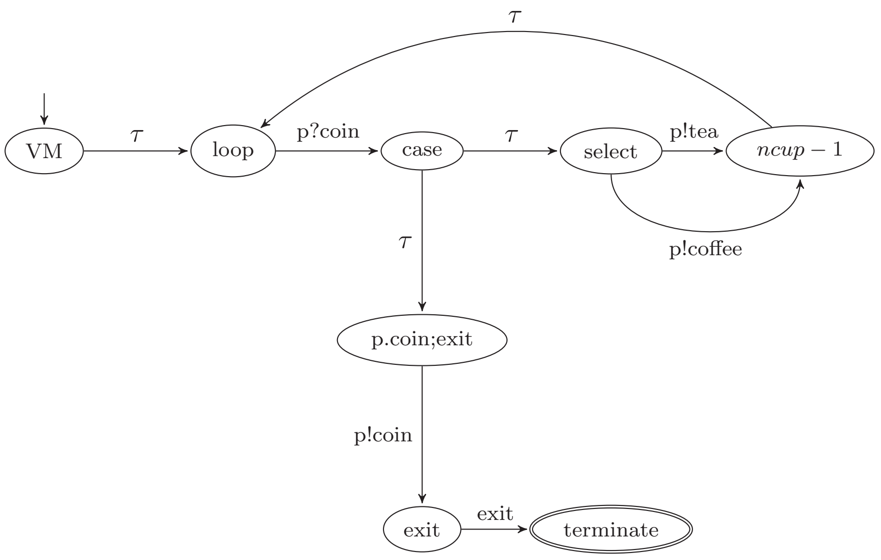


Figure 24: The beverage vending machine of Listing 4.4 modeled by a transition system

The operational semantics of the above statements is defined by the inference rules given in Table 3 below. These inference rules indicate that the execution of the exit command, variable assignment, and communication actions terminates in one step.

$\dfrac{}{exit \xrightarrow{exit} terminate}$	(I_1)	$\dfrac{}{x := expr \xrightarrow{\tau} terminate}$	(I_2)
$\dfrac{}{x := p.f \xrightarrow{p?f} terminate}$	(I_3)	$\dfrac{}{p.f := expr \xrightarrow{p!f} terminate}$	(I_4)

Table 3: Inference rules for statements $stmt \in \{exit, x := expr, p.f := x, x := p.f\}$

Sequential composition For the sequential composition, the set of sub-statements is defined as follows:

$$sub(stmt_1; stmt_2) = \Big\{stmt_1; stmt_2' \mid stmt_2' \in sub(stmt_2)\backslash\{terminate\}\Big\} \bigcup sub(stmt_2)$$

The operational semantics of the sequential composition is defined by the inference rules given in Table 4 below.

Operational semantics of sequential composition $stmt_1; stmt_2$ is defined by two inference rules to distinguish whether or not $stmt_1$ terminates in one step. If the first step of $stmt_1$ leads to a statement different from *terminate*, then the inference rule on the left applies (I_5). However, if the computation of $stmt_1$ terminates in one step by executing the action α, then $stmt_1;stmt_2$ moves to state $stmt_2$ after executing α. This is explained by the inference rule on the right (I_6).

$$\frac{stmt_1 \xrightarrow{\alpha} stmt_1' \neq terminate}{stmt_1; stmt_2 \xrightarrow{\alpha} stmt_1'; stmt2} \quad (I_5) \qquad \frac{stmt_1 \xrightarrow{\alpha} terminate}{stmt_1; stmt_2 \xrightarrow{\alpha} stmt2} \quad (I_6)$$

Table 4: Inference rules for sequential composition

Conditional statement For conditional statements, non-deterministic choice, the set of sub-statements is defined as the set consisting of the conditional statement itself, *cond_stmt* and sub-statements of its guarded statements. That is, for:

$$cond\_stmt = \textbf{case } |b_1| \to stmt_1 \ldots |b_n| \to stmt_n \textbf{ end}$$

the set of sub-statements is:

$$sub(cond\_stmt) = \{case\_stmt\} \ \cup \bigcup_{1 \leq i \leq n} sub(stmt_i)$$

The operational semantics of the conditional statement is defined by only one rule (see Table 5). This inference rules explain that the conditional statement has a τ-transition to each branch of the statement. The significance of the τ-transition is that the environment can neither control nor detect which branch is taken. The choice is made internally by the process and is hidden from the environment.

$$\frac{cond\_stmt = \textbf{case } |b_1| \to stmt_1 \ldots |b_n| \to stmt_n \textbf{ end}}{cond\_stmt \xrightarrow{\tau} stmt_i} \quad (I_7)$$

Table 5: Inference rule for conditional statement

Select construct For the deterministic choice statement in the form of:

$$select = \textbf{select } |b_1| \to stmt_1 \ldots |b_n| \to stmt_n \textbf{ end}$$

the set of sub-statements is defined as follows:

$$sub(select) = \{select, terminate\} \cup \bigcup_{1 \leq i \leq n} sub(stmt_i),$$

The transition system for the **select** statement has the same general appearance as the transition system of the **case** statement but its behavior is very different. Table 6 illustrates the only inference rule for the select construct. This rule explains that the select construct is able to deterministically choose one of its branches for which the environment offers communication.

$$\frac{stmt_i \xrightarrow{\alpha} stmt'_i}{select \xrightarrow{\alpha} stmt'_i} \qquad (I_8)$$

Table 6: Inference rule for **select** construct

Repetitive statement For the loop statement in the form of *loop* = **loop** *stmt* **end**, the set of sub-statements are defined as follows:

$$sub(loop) = \{loop, terminate\} \bigcup \Big\{stmt'; loop \mid stmt' \in sub(stmt) \backslash \{terminate\}\Big\},$$

$$\frac{stmt \xrightarrow{\alpha} stmt' \;\wedge\; stmt' \neq terminate \;\wedge\; \alpha \neq exit}{loop \xrightarrow{\alpha} stmt'; loop} \qquad (I_9)$$

$$\frac{stmt \xrightarrow{\alpha} stmt' \;\wedge\; stmt'_1 = terminate \;\wedge\; \alpha \neq exit}{loop \xrightarrow{\alpha} loop} \qquad (I_{10})$$

$$\frac{stmt \xrightarrow{exit} terminate}{loop \xrightarrow{exit} terminate} \qquad (I_{11})$$

Table 7: Inference rules for loop statement

For loops, we deal with three inference rules (see Table 7). The first rule explains that if the statement *stmt* doesn't terminate in one step, then the loop executes the statement *stmt*, and after the execution has been terminated the control moves back to the loop. The second rule explains the case where the statement *stmt* terminates in one step. In this case, the control moves back to the loop after the execution of *stmt* has been completed. The third rule, explains the execution of the explicit **exit** command inside the loop construct which causes the loop to terminate.

Figure 25 summarizes this section by illustrating some examples of Erasmus syntax on the left and their corresponding transition systems on the right. Note that for readability reason, the state names of transition systems are labeled with s_i instead of location names.

4.4.2 Protocols and Transition Systems

The previous section introduced the syntax of Erasmus programs, and explained how they can be modeled by means of transition systems. This section introduces the semantics of protocols, and explains how they can be modeled by transition systems.

Similar to processes, operational semantics of each protocol is defined as a transition system where states are protocol expressions and sub-expressions, and transition relations are defined by

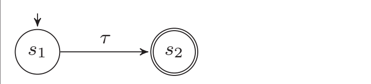
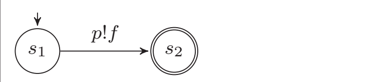
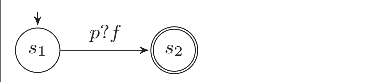
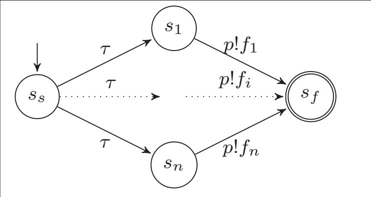
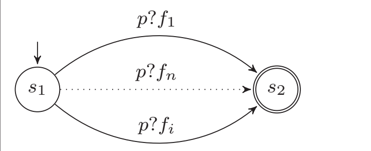
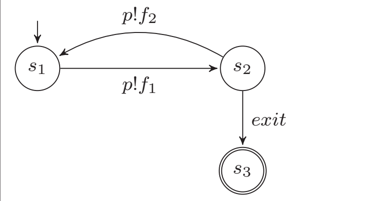
x:=*expr*	
p.*f*:=*expr*	
x:=*p*.*f*	
case{ | *b1*|*p*.*f1*:=*x1*; ... | *bn*|*p*.*fn*:=*xn*; }	 s_1 s_s s_f s_n τ τ τ $p!f_1$ $p!f_i$ $p!f_n$
select{ || *x1*:=*p*.*f1*; ... || *xn*:=*p*.*fn*; }	
loop{ *p*.*f1*:=*x*; ... **exit**; *x*:=*p*.*f2*; }	

Figure 25: Examples of Erasmus programs and their corresponding transition systems

inference rules. For each protocol expression, the set of sub-expressions are defined recursively (see Table 8).

$$\begin{array}{ll}
P \in \{s, \hat{r}, q:T, \hat{q}:T\} & sub(P) = \{P, terminate\} \\
P = P_1; P_2 & sub(P) = \Big\{P_1; P_2' \mid P_2' \in sub(P_2) \backslash \{terminate\}\Big\} \bigcup sub(P_2) \\
P = P_1 + P_2 & sub(P) = \{P_1 + P_2\} \cup sub(P_1) \cup sub(P_2) \\
P = P_1 \mid P_2 & sub(P) = \{P_1 \mid P_2\} \cup sub(P_1) \cup sub(P_2) \\
P = *P_1 & sub(P) = \{*P_1, terminate\} \bigcup \Big\{P_1'; *P_1 \mid P_1' \in sub(P_1) \backslash \{terminate\}\Big\} \\
P = \#P_1 & sub(P) = \{\#P_1\} \cup sub(*P_1)
\end{array}$$

Table 8: Protocol expressions on the left and the set of sub-expressions on the right

Table 9 provides inference rules for the Erasmus protocols. The inference rules for communication messages, sequential composition, deterministic and nondeterministic choice, and repetitive protocols give rise to the edges of a large transition systems, where the set of states agree with the set of protocol statements and sub-statements that are defined in Table 8. Thus the edges have the form

$P \xrightarrow{\alpha} P'$, where P is a protocol expression, P' a protocol subexpression, and α an action. Actions can be communication actions (representing send or receive), τ action (representing internal action), or the exit action (representing termination of loops).

$$\frac{}{s \xrightarrow{p!s} terminate} \quad (IP_1) \qquad \frac{}{\hat{r} \xrightarrow{p?r} terminate} \quad (IP_2)$$

$$\frac{}{q : T \xrightarrow{p!q} terminate} \quad (IP_3) \qquad \frac{}{\hat{q} : T \xrightarrow{p?q} terminate} \quad (IP_4)$$

$$\frac{P_1 \xrightarrow{\alpha} P_1' \ \wedge \ P_1' \neq terminate}{P_1; P_2 \xrightarrow{\alpha} P_1'; P_2} \quad (IP_5) \qquad \frac{P_1 \xrightarrow{\alpha} P_1' \ \wedge \ P_1' = terminate}{P_1; P_2 \xrightarrow{\alpha} P_2} \quad (IP_6)$$

$$\frac{P_1 \xrightarrow{\alpha_1} P_1'}{P_1 | P_2 \xrightarrow{\alpha_1} P_1'} \quad (IP_7) \qquad \frac{P_2 \xrightarrow{\alpha_2} P_2'}{P_1 | P_2 \xrightarrow{\alpha_2} P_2'} \quad (IP_8)$$

$$\frac{}{P_1 + P_2 \xrightarrow{\tau} P_1} \quad (IP_9) \qquad \frac{}{P_1 + P_2 \xrightarrow{\tau} P_2} \quad (IP_{10})$$

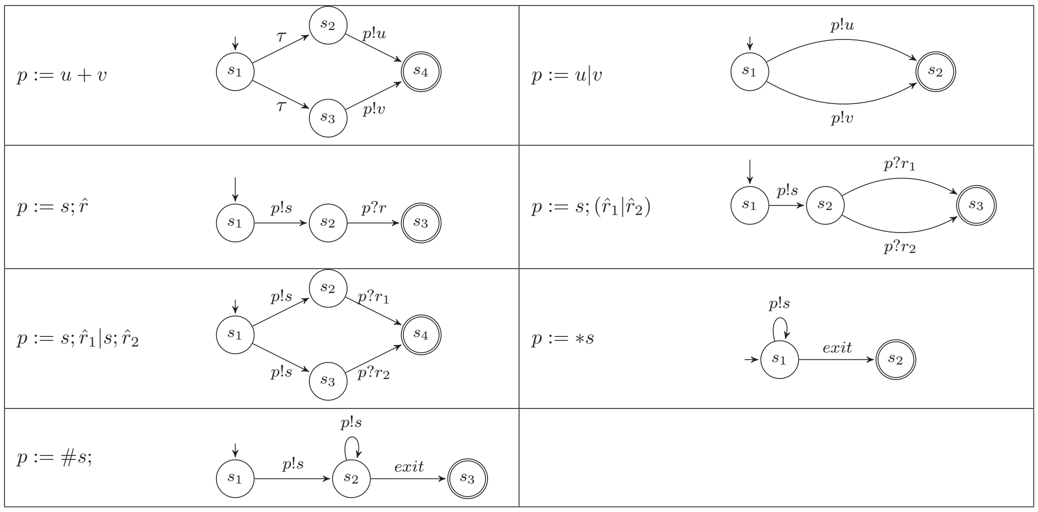
$$\frac{P \xrightarrow{\alpha} P' \ \wedge \ P' \neq terminate}{*P \xrightarrow{\alpha} P'; *P} \quad (IP_{11}) \qquad \frac{P \xrightarrow{\alpha} P' \ \wedge \ P' = terminate}{*P \xrightarrow{\alpha} *P} \quad (IP_{12})$$

$$\frac{}{*P \xrightarrow{exit} terminate} \quad (IP_{13})$$

$$\frac{P \xrightarrow{\alpha} P' \ \wedge \ P' \neq terminate}{\#P \xrightarrow{\alpha} P'; *P} \quad (IP_{14}) \qquad \frac{P \xrightarrow{\alpha} P' \ \wedge \ P' = terminate}{\#P \xrightarrow{\alpha} *P} \quad (IP_{15})$$

Table 9: Inference rules for protocols

Figure 26 summarizes this section by illustrating some examples of protocol terms (on the left) and their corresponding transition systems (on the right).



$p := u + v$
s_1 s_2 s_3 s_4 τ τ $p!u$ $p!v$
$p := u|v$
s_1 s_2 $p!u$ $p!v$
$p := s; \hat{r}$
s_1 s_2 s_3 $p!s$ $p?r$
$p := s; (\hat{r}_1|\hat{r}_2)$
s_1 s_2 s_3 $p!s$ $p?r_1$ $p?r_2$
$p := s; \hat{r}_1 | s; \hat{r}_2$
s_1 s_2 s_3 s_4 $p!s$ $p!s$ $p?r_1$ $p?r_2$
$p := *s$
s_1 s_2 $p!s$ $exit$
$p := \#s;$
s_1 s_2 s_3 $p!s$ $p!s$ $exit$

Figure 26: Examples of Erasmus protocols and their corresponding transition systems

4.5 Deciding Safety of Programs

The previous sections introduced the syntax of Erasmus programs and protocols, and explained how they can be used to specify client-server communications. This section introduces the *protocol satisfaction* as a solution for compatibility of components that can be used for safety checking of programs at compile time.

Checking that a process code is compatible with its protocol, or a protocol is compatible with another one is straightforward: each input or output operation must always perform the complete sequence of communications that the protocol describes. Since, protocols and processes can be represented as labeled transition systems, we define the following relation to check the compatibilities between labeled transition systems.

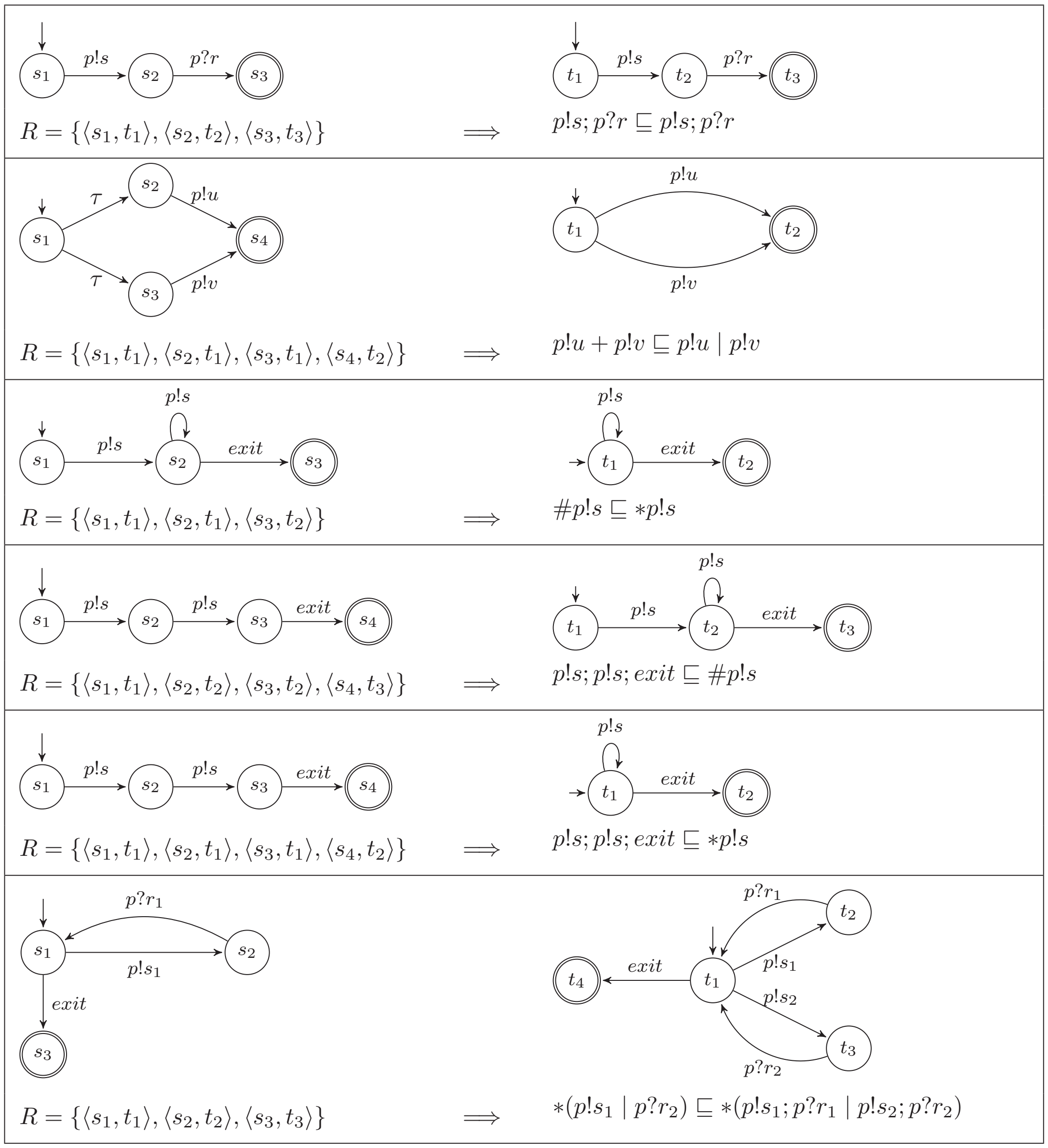
Definition 6. Let $\mathscr{L}_1 = (S_1, s_{1,0}, F_1, L_1, T_1)$ and $\mathscr{L}_2 = (S_2, s_{2,0}, F_2, L_2, T_2)$ be two transition systems. A binary relation R over the set of states S_1 and S_2 is called a *satisfaction relation* if and only if $\langle s_{1,0}, s_{2,0} \rangle \in R$, and whenever $\langle s, t \rangle \in R$ then:

1. For all actions $\alpha \in (L_1 \cup L_2) \backslash \{\tau\}$ and all states s' and t' such that $s' \in Post(s, \alpha)$ and $t' \in Post(t, \alpha)$, then $\langle s', t' \rangle \in R$,
2. For all states s' such that $s' \in Post(s, \tau)$, then $\langle s', t \rangle \in R$,
3. For all states t' such that $t' \in Post(t, \tau)$, then $\langle s, t' \rangle \in R$.

We say that $\mathscr{L}_2$ satisfies $\mathscr{L}_1$, or $\mathscr{L}_1$ is satisfied by $\mathscr{L}_2$, written $\mathscr{L}_1 \sqsubseteq \mathscr{L}_2$, if and only if there exists a *satisfaction relation* R over $S_1 \times S_2$.

One can think of the notion of satisfaction relation in terms of a two-person game. Suppose that there are two players which have their own behavior, modeled by means of transition systems. The game is played as follows: The first player makes a visible move (probably after some invisible moves) from its initial state. The role of the other player is to match this move precisely (probably after performing some invisible moves), also starting from its initial state. Next, again the first player makes another visible move and the other player must match this move, and so on. If the second player can play in such a way that at each point in the game it can match all the visible moves of the first player, then the second player satisfies the behavior of the first player. Otherwise, it doesn't.

Figure 27 illustrates examples of protocols (on the left) that are satisfied by protocols (on the right). The satisfaction relation, R, is also shown at the bottom of each pair of protocols.



$R = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \langle s_3, t_3\rangle\} \quad \Longrightarrow \quad p!s; p?r \sqsubseteq p!s; p?r$

$R = \{\langle s_1, t_1\rangle, \langle s_2, t_1\rangle, \langle s_3, t_1\rangle, \langle s_4, t_2\rangle\} \quad \Longrightarrow \quad p!u + p!v \sqsubseteq p!u \mid p!v$

$R = \{\langle s_1, t_1\rangle, \langle s_2, t_1\rangle, \langle s_3, t_2\rangle\} \quad \Longrightarrow \quad \#p!s \sqsubseteq *p!s$

$R = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \langle s_3, t_2\rangle, \langle s_4, t_3\rangle\} \quad \Longrightarrow \quad p!s; p!s; exit \sqsubseteq \#p!s$

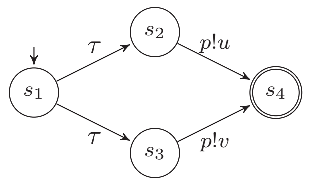
$R = \{\langle s_1, t_1\rangle, \langle s_2, t_1\rangle, \langle s_3, t_1\rangle, \langle s_4, t_2\rangle\} \quad \Longrightarrow \quad p!s; p!s; exit \sqsubseteq *p!s$

$R = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \langle s_3, t_3\rangle\} \quad \Longrightarrow \quad *(p!s_1 \mid p?r_2) \sqsubseteq *(p!s_1; p?r_1 \mid p!s_2; p?r_2)$

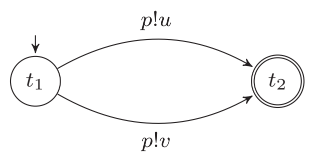
Figure 27: Examples of Erasmus protocols one the left that are satisfied by protocols on the right

The following examples illustrate some cases where satisfaction relation doesn't hold.

Example 1. This example shows that $p!u \mid p!v \not\sqsubseteq p!u + p!v$. Consider the transition systems of $p!u + p!v$ and $p!u \mid p!v$ that are given as follows:



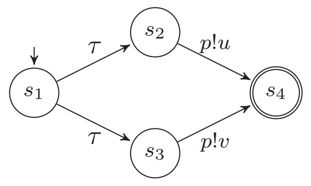
$p!u + p!v$

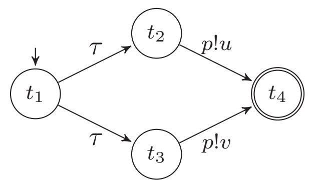


$p!u \mid p!v$

We start by constructing the satisfaction relation R over the states of the transition systems of $p!u + p!v$ and $p!u \mid p!v$. First, we add the pair of initial states; that is: $R = \{\langle t_1, s_1\rangle\}$. Having $\langle t_1, s_1\rangle \in R$ implies that the pairs $\langle t_1, s_2\rangle$ and $\langle t_1, s_3\rangle$ should also be added to R (third rule of definition 6). Having both $\langle t_1, s_2\rangle \in R$ and $\langle t_1, s_3\rangle \in R$ implies that (first rule of definition 6) whatever transitions t_1 has, s_2 and s_3 should have. This is, however, not true: t_1 has the transition $t_1 \xrightarrow{p!u} t_2$, but s_3 doesn't have this transition. Similarly, t_1 has the transition $t_1 \xrightarrow{p!v} t_2$, but s_2 doesn't have this transition. Thus, $p!u \mid p!v \not\sqsubseteq p!u + p!v$.

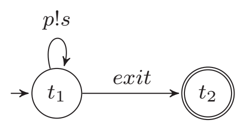
Example 2. This example illustrates that $p!u + p!v \not\sqsubseteq p!u + p!v$. We construct the satisfaction relation R over the states of the following transition systems:



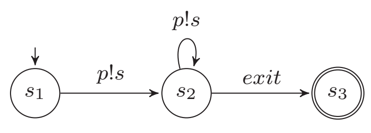


First, we add the pair of initial states, that is: $R = \{\langle s_1, t_1\rangle\}$. Since $\langle s_1, t_1\rangle \in R$, it should be clear that $\langle s_2, t_3\rangle$ is also in R (rules 2 and 3 of definition 6). This implies that whatever transition relations s_2 has, t_3 should also have. This is, however, not true, because $s_2 \xrightarrow{p!u} s_4$ and $|Post(t_3, p!u)| = \phi$. Therefore, $p!u + p!v \not\sqsubseteq p!u + p!v$.

Example 3. This example shows that $*p!s \not\sqsubseteq \#p!s$. We construct the satisfaction relation R on the following transition systems:



$*p!s$



$\#p!s$

First, we add the pair of initial states; that is: $R = \{\langle t_1, s_1 \rangle\}$. Having $\langle t_1, s_1 \rangle \in R$ implies that whatever transition relations t_1 has, s_1 should also have. This is, however, not true, because $t_1 \xrightarrow{exit} t_2$, while $|Post(s_1, exit)| = \phi$. Therefore, $*p!s \not\sqsubseteq \#p!s$.

Theorem 4.5.1. The satisfaction relation has the following properties:

i) It is not *reflexive*: $P \not\sqsubseteq P$,

ii) It is not *symmetric*: $P \sqsubseteq Q$ does not imply that $Q \sqsubseteq P$,

iii) It is *transitive*: $P \sqsubseteq Q$ and $Q \sqsubseteq R$ implies that $P \sqsubseteq R$.

Proof. For reflexivity, it is enough to show a counter example. Example 2 illustrates such an example, that is: $a + b \not\sqsubseteq a + b$. For symmetry, similar to reflexivity, it is enough to show a counter example. Let $P = a + b$ and $Q = a \mid b$ be two protocol expressions. It is clear from Figure 27 and Example 1 that $P \sqsubseteq Q$, and $Q \not\sqsubseteq P$.

For transitivity, we must show that if S_1 and S_2 are satisfaction relations, then so is their relational composition

$$S_1 S_2 = \{\langle p, r \rangle \mid \exists q.\ \langle p, q \rangle \in S_1 \text{ and } \langle q, r \rangle \in S_2\}$$

It is enough to show that this is a satisfaction relation. Let $\langle p, r \rangle \in S_1 S_2$, and α be an action such that $p \xrightarrow{\alpha} p'$. Since there exists q such that $\langle p, q \rangle \in S_1$ and $\langle q, r \rangle \in S_2$, there exists also q' such that $q \xrightarrow{\alpha} q'$, and $\langle p', q' \rangle \in S_1$, and therefore r' such that $r \xrightarrow{\alpha} r'$ and $\langle q', r' \rangle \in S_2$. Thus, $\langle p', r' \rangle \in S_1 S_2$, and we have established the satisfaction condition for $S_1 S_2$. □

Note that the syntax of protocols do not illustrate the direction of messages for both client and server processes. Indeed they define the direction of messages from the client perspective only. As an example, consider the following protocol:

$$prot = s_1; \uparrow r_1 \mid s_2; \uparrow r_2$$

in the perspective of the client valid conversations are $s_1; \uparrow r_1$ and $s_2; \uparrow r_2$. However, for a server process valid conversations are $\uparrow s_1; r_1$ and $\uparrow s_2; r_2$. When comparing a protocol with a server process using protocol satisfaction relation, we should also take the direction of messages into our consideration. For this reason the following notation is used.

Notation 1. Let P be a process expression, and let $\mathscr{L}(P)$ be its corresponding transition system. The transition system $\mathscr{L}(\overline{P})$ is called the dual of $\mathscr{L}(P)$ and is obtained by replacing all send operations in $\mathscr{L}(P)$ by receive operations and *vice-versa.* That is: all $q!s$ actions are replace by $q?s$, and all $q?r$ actions are replaced by $q!r$ (for all ports q in P).

Having defined the satisfaction relation, we are still left with the problem of deciding safety of programs with respect to communication. We define safety of programs as follows:

Definition 7. Let P_{cl} and P_s be the client and the server processes that are connected through channel C, and let π_{cl}, π_s, and π_c be the protocols associated with the client process, server process, and the channel respectively.

We say that the client and the server processes can be connected safely through channel C if and only if the following satisfaction relations hold:

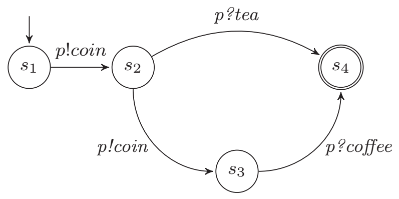
$$\mathscr{L}(P_{cl}) \sqsubseteq \mathscr{L}(\pi_{cl}) \sqsubseteq \mathscr{L}(\pi_c) \sqsubseteq \mathscr{L}(\pi_s) \sqsubseteq \mathscr{L}(\overline{P_s})$$

A program is called *safe* with respect to communication if every pair of communicating processes in the program can be safely connected to one another.

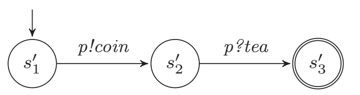
In the rest of this section, we illustrate some examples of safe and unsafe programs. These examples demonstrate how the static analysis can be used to determine whether or not a program is safe with respect to communication.

Example 4. This example is a variation of an earlier example of the vending machine. Figure 28 shows the complete program which consists of a protocol, a vending machine process, and a customer process. Transition systems of processes and the protocol are shown beside their code. Note that the transition system beside the vending machine corresponds to $\mathscr{L}(\overline{VM})$.

```
protocol prot={coin;(↑tea|coin;↑coffee)}
```



```
process CS = p: −prot{
  p!coin; p?tea;
}
```



```
process VM = p: +prot{
  p?coin;
  select{
    || p?coin; p! coffee ;
    || p! tea;
  }
}
```

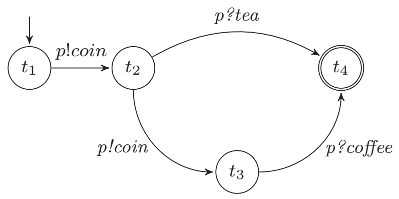


Figure 28: Examples of a reliable vending machine

We conclude from the following satisfaction relations that the program is safe with respect to communications:

1. $\mathscr{L}(CS) \sqsubseteq \mathscr{L}(prot)$: Let $R_1 = \{\langle s'_1, s_1\rangle, \langle s'_2, s_2\rangle, \langle s'_3, s_4\rangle\}$
2. $\mathscr{L}(prot) \sqsubseteq \mathscr{L}(prot)$: Let $R_2 = \{\langle s_1, s_1\rangle, \langle s_2, s_2\rangle, \langle s_3, s_3\rangle, \langle s_4, s_4\rangle\}$
3. $\mathscr{L}(prot) \sqsubseteq \mathscr{L}(\overline{VM})$: Let $R_3 = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, \langle s_3, t_3\rangle, \langle s_4, t_4\rangle\}$

Relations R_1, R_2, and R_3 imply that: $\mathscr{L}(CS) \sqsubseteq \mathscr{L}(prot) \sqsubseteq \mathscr{L}(\overline{VM})$. Therefore, the customer process can safely be connected to the vending machine, and the program is safe.

Example 5. This example shows an unsafe program with respect to communication. Suppose that the vending machine given in the previous example is replaced by the vending machine given in Figure 29. Figure 29 illustrates the code of *VM2* (on the left) and the transition system of $\overline{VM2}$ (on the right).

```
process VM2 = p: +prot{
 select{
   || p?coin; p?coin; p! coffee ;
   || p?coin; p! tea;
 }
}
```

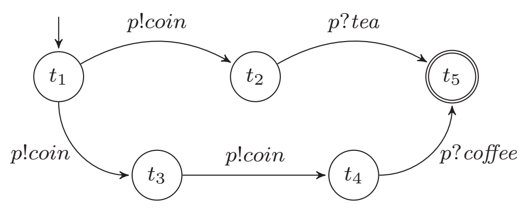


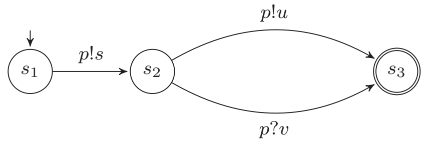
Figure 29: Examples of an unreliable vending machine

This program is considered to be unsafe, because the communication between the vending machine and the channel is unsafe: After the insertion of a coin, the vending machine may choose the first branch of its **select** construct, which forces the customer to insert another coin for a cup of coffee.

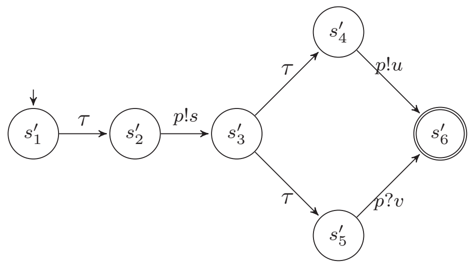
We show this by constructing the satisfaction relation R over the states of $\mathscr{L}(prot)$ and $\mathscr{L}(\overline{VM2})$. At the first step, we add the $\langle s_1, t_1\rangle$ (pair of initial states) into R. Having $s_1 \xrightarrow{p!coin} s_2$, $t_1 \xrightarrow{p!coin} t_2$, and $t_1 \xrightarrow{p!coin} t_3$ imply that we should also add $\langle s_2, t_2\rangle$ and $\langle s_2, t_3\rangle$ into R. Having both $\langle s_2, t_2\rangle \in R$ and $\langle s_2, t_3\rangle \in R$ implies that whatever transitions s_2 takes, both states t_2 and t_3 should match it. This is, however, not true: state s_2 has the transition $s_2 \xrightarrow{p?tea} s_4$ but $|Post(t_3, p?tea)| = \phi$. Thus, $\mathscr{L}(prot) \not\sqsubseteq \mathscr{L}(\overline{VM2})$, and the communication between the vending machine and the channel is not safe.

Example 6. Figure 30 demonstrates an example in which the program is carefully designed to be safe, but static analysis doesn't demonstrate this. In this program, the process *Chooser* (client) picks a random value (*true* or *false*) for the boolean variable *start*, and sends the value to the server process *User*. Both the client and server processes, therefore, use the same value of *start* to choose the branch of **case** statement, and the communication between them should be safe. However, it is clear that this is a fragile kind of safety, which would be destroyed by a small change in the program logic.

```
protocol prot={s:Boolean;(u:Integer|↑v:Int)}
```



```
process Chooser = p: −prot{
 w : Integer;
 start : Boolean := random()>0.5;
 p.s := start;
 case{
  | start |  p.u := 42;
  ||  w := p.v;
 }
}
```



```
process User = p: +prot{
 w : Integer;
 start : Boolean := p.s;
 case{
  | start |  w := p.u;
  ||  p.v := 55;
 }
}
```

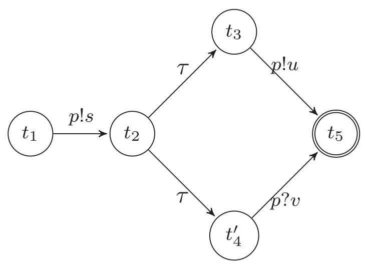


Figure 30: Example of a fragile kind of safety

Checking the program for safety, we get:

$$\mathscr{L}(Chooser) \sqsubseteq \mathscr{L}(prot), \text{ and } \mathscr{L}(prot) \not\sqsubseteq \mathscr{L}(\overline{User})$$

This example demonstrates that there is nothing intrinsically wrong with the process *Chooser* and the protocol *prot*. Its is linking of the process *User* and the channel that creates the problem. To make this program safe, we replace the process *User* with the process *User2*, given in Figure 31. The **select** statement in *User2* makes the choice dependent on the environment (on the messages received) rather than the process's own logic. Now, it can be easily shown that $\mathscr{L}(prot) \sqsubseteq \mathscr{L}(\overline{User2})$, and that the program is safe.

```
process User2 = p: +prot{
 w:Integer;
 start : Boolean := p.s;
 select{
   || w := p.u;
   || p.v := 55;
 }
}
```

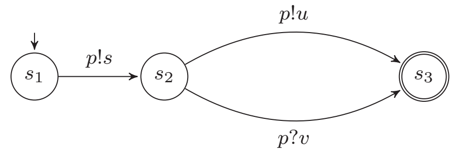


Figure 31: Safety of program is obtained by replacing the process *User* by *User2*

4.6 Algorithms

The Erasmus compiler constructs transition systems from the code of processes and protocols associated with processes' ports and channels. Construction of these transition systems are briefly explained in sections 4.4.1 and 4.4.2. The compiler then uses the algorithm given in Figure 32 to construct the satisfaction relation on each connected pair of transition systems to determine whether or not they can be safely connected. That is:

1. $\mathscr{L}(client) \sqsubseteq \mathscr{L}(\pi_{client})$ (client code is satisfied by its protocol)
2. $\mathscr{L}(\pi_{client}) \sqsubseteq \mathscr{L}(\pi_{channel})$ (client protocol is satisfied by channel protocol)
3. $\mathscr{L}(\pi_{channel}) \sqsubseteq \mathscr{L}(\pi_{server})$ (channel protocol is satisfied by server protocol)
4. $\mathscr{L}(\pi_{server}) \sqsubseteq \mathscr{L}(\overline{server})$ (server protocol is satisfied by server code)

The algorithm can be realized by performing depth-first-search on the given labelled transition systems ($\mathscr{L}_1 = \langle S, s_0, F, T\rangle$ and $\mathscr{L}_2 = \langle S', s'_0, F', T'\rangle$). It starts with the initial states, and returns *true* if it can construct a satisfaction relation over the states of $S \times S'$, or *false* otherwise. Two data structures are required:

- A set R that stores all the visited pair of states.
- A stack ST that stores the states being analyzed in the current execution sequence. Each element of ST is a tuple $\langle s, \alpha, t\rangle$, where s and t are states of $\mathscr{L}_1$ and $\mathscr{L}_2$ respectively, and α is an action that indicates how the previous element of the stack is evolved into the current states.

The set R becomes the satisfaction relation if and only if the algorithm returns true; that is: $\mathscr{L}_1 \sqsubseteq \mathscr{L}_2$. In case of a *false*, counter examples are obtained by retrieving the elements of the stack, from bottom to top. These examples demonstrate those *runs* of the program (see Definition 4), that cause the violation of the safety property.

```
Require: $\mathscr{L}_1 = \langle S, s_0, T\rangle$ and $\mathscr{L}_2 = \langle S', s'_0, T'\rangle$
Ensure: True if $\mathscr{L}_1 \sqsubseteq \mathscr{L}_2$, or false + counter example(s).
procedure START()
    Stack $ST \leftarrow \phi$;
    $R \leftarrow \phi$;
    $satisfy \leftarrow true$;
    PUSHSTACK($ST$, $\langle s_0, \tau, s'_0\rangle$);
    ADD($R$, $\langle s, s'\rangle$);
    COMPARE();
    return satisfy;

procedure COMPARE()
    $\langle s, \alpha_1, t\rangle \leftarrow$ TOPSTACK($ST$);
    for all $\langle s, \alpha_2, s'\rangle \in T$ do
        if $\alpha_2 = \tau$ then                                    ▷ Rule 2 of Definition 6
            INSERT($\langle s', \tau, t\rangle$);
        else
            $match\_flag \leftarrow false$;
            for all $\langle t, \alpha_3, t'\rangle \in T'$ do
                if $\alpha_3 = \alpha_2$ then                        ▷ Rule 1 of Definition 6
                    $match\_flag \leftarrow true$;
                    INSERT($\langle s', \alpha_3, t'\rangle$);
                else if $\alpha_3 = \tau$ then                       ▷ Rule 3 of Definition 6
                    $match\_flag \leftarrow true$;
                    INSERT($\langle s, \alpha_3, t'\rangle$);
            if $match\_flag = false$ then
                        ▷ Safety Violation! Counter example is elements of the stack (from bottom to top)
                PRINTSTACK(ST);
                $satisfy \leftarrow false$;
    POPSTACK(ST);

procedure INSERT($\langle s, \alpha, t\rangle$)
    if $\langle s, t\rangle \notin R$ then
        PUSHSTACK(ST, $\langle s, \alpha, t\rangle$);
        ADD($R$, $\langle s, t\rangle$);
        COMPARE();
```

Figure 32: Construction of satisfaction relation between two labelled transition systems

Proposition 1. Algorithm given in Figure 32 terminates, and it returns *true* if and only if $\mathscr{L}_1 \sqsubseteq \mathscr{L}_2$.

Proof. We use the following notations: let $COMPARE_i$ representing the i^{th} execution of the procedure *COMPARE*, and let R_i (resp. ST_i) representing the set R (resp. stack ST) at the end of $COMPARE_i$. In addition, let $\langle s_i, \alpha_i, t_i\rangle$ representing the top element of the stack during the execution of $COMPARE_i$, and n_i (resp. m_i) representing the number of states (resp. number of transitions) of $\mathscr{L}_i$.

Termination To show that the algorithm will eventually terminate, it is sufficient to find an upper bound for the number of times the *COMPARE* function is called. The upper bound is $n \leq n_1 \times n_2$-times, because there exists $n_1 \times n_2$ distinct pairs over $S_1 \times S_2$. Thus, the *COMPARE* function is called maximum n times by the INSERT function, which guarantees the termination of the algorithm.

Correctness To show the correctness of the algorithm, it is sufficient to show that the set R is the satisfaction relation over $S_1 \times S_2$ if and only if $\mathscr{L}_1 \sqsubseteq \mathscr{L}_2$. When $COMPARE_i$ terminates, exactly one of the following properties hold:

i) $\mathit{satisfy} = \mathit{false} \Leftrightarrow \exists \alpha \in ACT.\ s_i \xrightarrow{\alpha} s'_i \ \wedge \ \nexists t'_i.(\ t_i \xrightarrow{\alpha} t'_i \ \vee \ t_i \xrightarrow{\tau} t'_i)$

ii) $\mathit{satisfy} = \mathit{true} \Leftrightarrow \forall \alpha \in ACT \cup \{\tau\}.\ s_i \xrightarrow{\alpha} s'_i \ \Rightarrow \begin{cases} \alpha = \tau \\ \vee \\ \exists t'_i.\ (t_i \xrightarrow{\alpha} t'_i \ \vee \ t_i \xrightarrow{\tau} t'_i) \end{cases}$

From (i) we impliy that, there exists a pair of states $\langle s_i, t_i \rangle \in R_i$ such that s_i has a transition ($\alpha \neq \tau$), but t_i can not match it. Therefore, $R = \bigcup_{i=1}^{n} R_i$ is not a satisfaction relation and $\mathscr{L}_1 \not\sqsubseteq \mathscr{L}_2$. However, if (ii) holds for all executions of $COMPARE_i$, we imply that for $R = \bigcup_{i=1}^{n} R_i$ whenever $\langle s, t \rangle \in R$, then:

$$\forall \alpha \in ACT \cup \{\tau\}.\ s \xrightarrow{\alpha} s' \Rightarrow \begin{cases} \alpha = \tau \ \wedge \langle s', t \rangle \in R \\ \vee \\ t \xrightarrow{\tau} t' \xrightarrow{\alpha} t'' \ \wedge \ \langle s', t'' \rangle \in R \ \wedge \ \langle s, t' \rangle \in R \end{cases}$$

Having $\langle s_0, s'_0 \rangle \in R$, implies that R is indeed the satisfaction relation, and that $\mathscr{L}_1 \sqsubseteq \mathscr{L}_2$. □

The time requirement for the function $COMPARE_i$ is $O(m_1 \times m_2)$ in the worst case. Since we have maximum n recursive calls, the worst theoretical time requirement for the algorithm is $O(n \times m_1 \times m_2)$. The memory requirement for the algorithm is $O(n)$.

4.7 Problems and Solutions

Our description of the analysis of safety of Erasmus programs in the previous sections glossed over some difficult issues which we will address in this section.

Successful Termination Figure 33 demonstrates an example where the communication between processes is unsafe, but the static analysis cannot demonstrate this. Static analysis of these transition systems implies that:

$$\mathscr{L}(\mathit{client}) \sqsubseteq \mathscr{L}(\mathit{prot}) \sqsubseteq \mathscr{L}(\overline{\mathit{server}})$$

protocol *prot* = {*s*;↑*r1*;↑*r2*}

process *client* = *p*: −*prot*{
 p!*s*; *p*?*r1*;
}

process *server* = *p*: +*prot*{
 p?*s*; *p*!*r1*; *p*!*r2*;
}

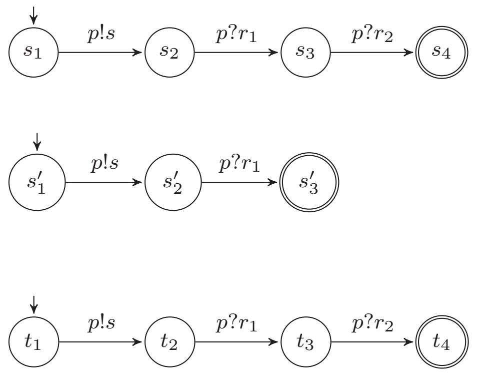


Figure 33: An example of an unsafe program

However, the problem is that, after receiving r_1, the client process terminates while the server process (on the other side of the channel) hangs sending the signal r_2.

The solution that we have adopted is to include an *exit* action for processes and protocols that do not reach their final states with an explicit *exit* action. On the basis of examples of this kind, we have considered an automatic transformation, performed by the compiler, that would, for example generate code for code in Figure 33 as if it was written like the code in Figure 34. Now, static analysis of the three transition systems imply that:

$$\mathscr{L}(\mathit{client}) \not\sqsubseteq \mathscr{L}(\mathit{prot}) \sqsubseteq \mathscr{L}(\overline{\mathit{server}})$$

Unreachable Final States Figure 35 illustrates a problem that we have encountered in designing Erasmus programs. The code contains two processes *Looper* and *Catcher* that are linked together. Process *Looper* performs 10 iterations and then terminates. Unfortunately, process *Catcher* has no way of knowing that *Looper* has terminated, and "hangs", waiting for another message.

```
protocol prot = {s;↑r1;↑r2; exit}

process client = p: −prot{
 p!s;  p?r1;  exit;
}

process server = p: +prot{
 p?s;  p!r1;  p!r2;  exit;
}
```

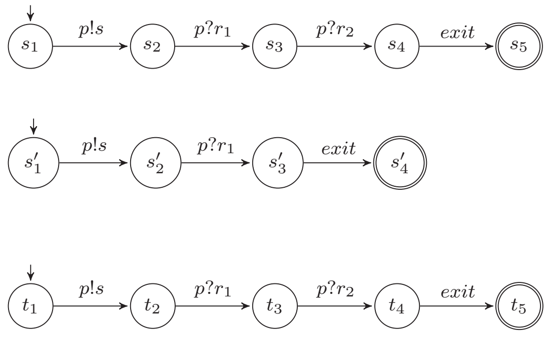

s_1 $\xrightarrow{p!s}$ s_2 $\xrightarrow{p?r_1}$ s_3 $\xrightarrow{p?r_2}$ s_4 $\xrightarrow{exit}$ s_5
s'_1 $\xrightarrow{p!s}$ s'_2 $\xrightarrow{p?r_1}$ s'_3 $\xrightarrow{exit}$ s'_4
t_1 $\xrightarrow{p!s}$ t_2 $\xrightarrow{p?r_1}$ t_3 $\xrightarrow{p?r_2}$ t_4 $\xrightarrow{exit}$ t_5

Figure 34: Automatic conversion of the program given in Figure 33

```
protocol prot = {∗s}

process Looper = p: −prot{
 count: Int := 10;
 loop{
   |count = 0| exit;
   ||  p!s;  count −=1;
 }
}

process Catcher = p: +prot{
 loop{
   p?s;
 }
}
```

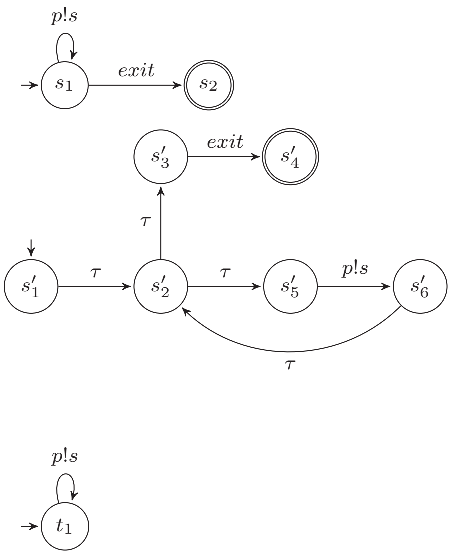


Figure 35: Loop with implicit exit

The program should be safe, because all the messages sent by *Looper* can be received by the *Catcher*, and *vice-versa*. However, when we check safety of the program, we get:

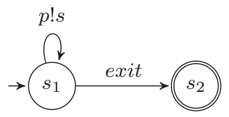
$$\mathscr{L}(\mathit{Looper}) \sqsubseteq \mathscr{L}(\mathit{prot}) \not\sqsubseteq \mathscr{L}(\overline{\mathit{Catcher}})$$

This is because the final state of *Catcher* is unreachable; there is no *exit* statement in the loop. The solution that we adopted is to include a *close* (Recall that the *close* signals is used by processes to terminate channels. Any attempt from a process to perform a communication on a closed channel fails.) signal in the *Looper* process, and to use the **select** construct in the *Catcher* process (see Figure 36).

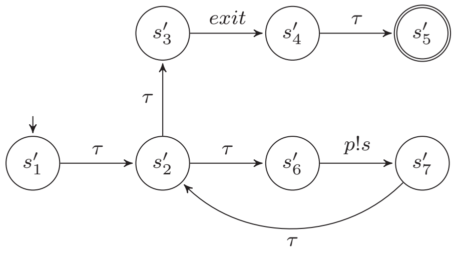
In this way, the *Looper* process closes the channel before it terminates, and forces the *Catcher* to choose the *orelse* branch of the **select** construct (Recall that a **select** construct performs the *orelse* branch if all of the channels are closed.). The communication is now safe, and the following satisfaction relations hold:

$$\mathscr{L}(Looper) \sqsubseteq \mathscr{L}(prot) \sqsubseteq \mathscr{L}(\overline{Catcher})$$

```
protocol prot = {*s}
```



```
process Looper = p: −prot{
 count: Int := 10;
 loop{
   |count = 0| exit;
   || p!s; count −=1;
 }
 p.close;
}
```



```
process Catcher = p: +prot{
 loop select{
   || p?s;
   |orelse| exit;
 }
}
```

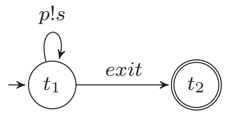


Figure 36: Loop with an explicit exit action

Similar to the previous example, the compiler performs automatic transformation for programs based on this kind of example.

Processes with multiple ports Figure 37 demonstrates an example of a process that has two ports. In this example, process A receives signal s from its server port q, and communicates with its client port p before replying back to its server port q. The problem here is that when we check safety of the program with respect to communication on port p, we get:

$$\mathscr{L}(A) \not\sqsubseteq \mathscr{L}(prot)$$

Similarly, when we check safety of the program with respect to communication on port q, we get :

$$\mathscr{L}(prot) \not\sqsubseteq \mathscr{L}(\overline{A})$$

This is because the transition system of process A contains communication actions of both p and q. However, when we check the safety of the program with respect to communication on a specific

protocol *prot* = {*s*;↑*r*}

$$\begin{tikzcd} \arrow[d] & & \\ s_1 \arrow[r, "p!s"] & s_2 \arrow[r, "p?r"] & s_3 \end{tikzcd}$$

process *A* = *p*:−*prot*; *q*:+*prot*{
 q?*s*; *p*!*s*; *q*!*r*; *p*?*r*;
}

$$\begin{tikzcd} \arrow[d] & & & & \\ s'_1 \arrow[r, "q?s"] & s'_2 \arrow[r, "p!s"] & s'_3 \arrow[r, "q!r"] & s'_4 \arrow[r, "p?r"] & s'_4 \end{tikzcd}$$

Figure 37: Safety checking of processes with more than one ports

port, say p, we would like to perform safety checking solely on port p and skip those on port q. Therefore, the solution that we adopted is to rename communication actions on port q with the τ action when we check the safety of the program with respect to port p. The same approach is performed when we check the safety of communication on port q (see Figure 38).

process *A* = *p*:−*prot*;
 q:+*prot*{
 q?*s*; *p*!*s*; *q*!*r*; *p*?*r*;
}

$$\begin{tikzcd} \arrow[d] & & & & \\ s'_1 \arrow[r, "\tau"] & s'_2 \arrow[r, "p!s"] & s'_3 \arrow[r, "\tau"] & s'_4 \arrow[r, "p?r"] & s'_4 \end{tikzcd}$$

Figure 38: Renaming of communication actions on port q with the τ action

Cyclic Communication Pattern Safety of programs with respect to communications doesn't always imply freedom of deadlock. Programs can be easily written in such a way that results in deadlock states. As an example, consider the following code:

```
protocol prot = {x: Integer}

process A = p:−prot, q:+prot{
 p.x := q.x;
}
process B = p:+prot, q:−prot{
 q.x := p.x;
}
cell main = c1:prot; c2:prot; A(c1,c2); B(c1,c2);
```

The above code illustrates an example where processes A and B are linked together in a cyclic fashion through channels c_1 and c_2. In this configuration, process A receives an integer value from its server port q, and passes this value to its client port p. Similarly, process B receives an integer value from its server port p, and passes this value to its client port q. The program is safe with respect to communications on both channels, but the system as a whole deadlocks because both processes wait to receive from their server ports.

It is not possible to statically check the presence of deadlock in cyclic communications using the protocols and the satisfaction relation introduced in this chapter. Our experience in designing Erasmus programs shows that a cyclic program usually turns out to be a design error, and well constructed programs are acyclic. We don't prohibit cycles, but rather discourage them. For this reason, for cyclic communication configurations the compiler generates warnings, and allow programmers to invoke either the *model-checker* (based on mCRL2 [J.F08]), or a *dynamic debugger* (as it is done in Scala programming language [BL12]).

Figure 39 summarizes this section by illustrating the automatic transformations (on the right) of some Erasmus codes (on the left) that is performed by the compiler.

```
process A1 = p1: −prot; ..., pn: −prot{    | process A2 = p1: −prot; ..., pn: −prot{
 p1!f;                                     |  p1!f;  ...;  pn!f;
 ...                                       |  p1.close;  ... pn.close;
 pn!f;                                     |  exit;
}                                          | }
-------------------------------------------+-------------------------------------------
                                           | process B2 = p: +prot; {
process B1 = p: +prot; {                   |  loop select{
 loop{                                     |    || p?f;  ...;
   ...;  p?f;  ...;                        |   |orelse|  exit;
 }                                         |  }
}                                          | }
-------------------------------------------+-------------------------------------------
process C1 = p: +prot; q: +prot{           | process C2 = p: −prot; q: −prot{
 loop select{                              |  loop select{
   ||  p?f;  ...                           |    ||  p?f;  ...
   ||  q?f;  ...                           |    ||  q?f;  ...
  |orelse|  scrln("Error");                |   |orelse|  scrln("Error");  exit;
 }                                         |  }
}                                          | }
```

Figure 39: Codes on the left, and transformation done by the compiler on the right

4.8 Summary

We have proposed an extension to the Erasmus programming language that would significantly extend the expressive power of client-server communications by allowing the communication pattern on channels to be specified using protocols. Protocols allow communications between client and server processes to be defined as conversations consisting of several messages in both directions, hence going beyond the traditional request-response message exchange pattern.

In addition, we have defined the protocol satisfaction relation, and shown how it can be used to check safety of inter-process communications at compile-time. Safety of communications guarantees that every messages is correctly received by the destination process. Safety of a program is then obtained by constructing a chain of satisfaction relations on the labelled transition systems of pair of communicating processes and their associated protocols. Satisfaction relation not only ensures that communications between any two-connected processes proceed in a consistent manner, but also allows the compiler to detect the following program errors at compile-time:

- Starvation: The client process waits for a message that the server process never sends (or *vice-versa*).
- Type conflict: The client process expects a message of type T_1 but the server process offers a message of type T_2 (or *vice-versa*).
- Sequence error: The client process sends x followed by y, but the server process expects y followed by x (or *vice-versa*).

We have also discussed some problems (and provided solutions) that we encountered developing Erasmus language, particularly the problems of protocol termination, reachability of final states, processes with multiple ports, and cyclic communication patterns. Although protocols and satisfaction relation allow the compiler to detect several program errors, but they do not guarantee that the system as a whole never deadlock. Detecting deadlocks at compile-time is known to be a difficult problem to solve [RT85], and is out of scope of our work.

We also have plans to develop the Erasmus client-server interface further. We would like to add the idea of *protocol inheritance*, which allows the fields from one or more existing protocols to be incorporated into a new protocol. Protocol inheritance which was first introduced by Occam-π would simplify many programs. As an example, we would like to be able to write codes similar to the code in Listing 4.5.

Listing 4.5: Protocol Inheritance

```
protocol A = {coin; ↑tea}
protocol B = {coin; coin; ↑coffee}
protocol C = {#(A | B)}

process P = p:+C{
  loop select{
    || p.coin; p.tea;
    || p.coin; p.coin; p.coffee;
  }
}
```

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The spread of multicore architectures has a huge effect on the performance of software. The increase in the performance depends on how programmers make effective use of hardware parallelism. In order to get the true performance gains, programs need to be parallelized completely. But parallel programming is hard, mainly because mainstream programming languages do not provide suitable abstractions for expressing and controlling concurrency.

In this thesis we introduced the Erasmus project. A process oriented programming language that is based on Hoare's *Communicating Sequential Processes* (CSP) that fully specifies thread synchronization by algebraic notations. The goal is to make the CSP paradigm more practical. Erasmus addresses concurrency by providing *processes* as the primary abstraction. Processes interact with one another through *synchronous channels*. Channels and processes are associated with *protocols* that specify the interprocess communication pattern.

This thesis focused on two problems:

1. In Chapter 3 we presented an efficient implementation of the CSP *generalized alternative* construct that allows a process to non-deterministically choose between several possible communication. Particularly, we used ACP process algebra to model our design (Section 3.4 and Section 3.6), and we used mCRL2 model checker to verify the correctness of it (Section 3.4.3, Section 3.5.2, and Section 3.6.2). The use of ACP and mCRL2 is invaluable in this work, and we believe that we would not have achieved a correct design without them. The proposed model removes previous restrictions on the use of this construct by allowing branches of an alternation to be guarded by both send and receive operations, and by allowing both ends of a synchronous channel to participate in an alternation. In the model, we have also considered

the concepts of *closing of channels* (Section 3.5), and *priorities* that can be imposed on the alternative construct (Section 3.7). Several tests on different configurations have been also given (Section 3.8). The test results gave very acceptable performance, and greatly increased our confidence in our design.

2. In Chapter 4 we have proposed an extension to the Erasmus programming language that would significantly extend the expressive power of client-server communications. We introduced *protocols* and explained how they can be used to specify communication pattern between client and server processes (Section 4.3). The client-server communication pattern was defined as conversations consisting of several messages in both directions, hence going beyond the traditional request-response message exchange pattern. The notion of *protocol satisfaction* was also defined (Section 4.5), allowing us to perform safety checking of inter-process communications at compile-time. Safety of communications guarantees that every messages is correctly received by the destination process. Safety of a program is then obtained by constructing a chain of satisfaction relations on the labelled transition systems of pair of communicating processes and their associated protocols (Section 4.4 and Section 4.6). Satisfaction relation not only ensures that communications between any two-connected processes proceed in a consistent manner, but also allows the compiler to detect program errors such as starvation, sequence errors, and deadlocks. We have also discussed several problems (and solutions) that we encountered during development of Erasmus language (Section 4.7).

5.2 Future Work

In the following, several issues are mentioned which are interesting to investigate.

- In Chapter 3 a model for the CSP generalized alternative construct is proposed. The proposed model assumes that processes and channels are running on the same computer in the same memory space. The same model can be extended to cover network channels that allow processes to communicate around networks. We believe that the standard implementation for this case uses a broker on each computer. A process communicates with its broker to set up communication with components on other computers.
- It would be also interesting to extend the **select**-construct model to include timeouts. When no channels become ready in a specified time, then the **select**-construct executes the timeout branch.
- It would be also interesting to extend the model to include barrier synchronization that allows multiple processes (not just two) to synchronize together.

- In Chapter 4, we proposed protocols as a means to specify client-server communication. It would be interesting to add the idea of protocol inheritance that allows the fields from one existing protocols to be incorporated into a new protocol. This idea, which was first introduced in Occam-π, would simplify many programs.
- In Section 4.7, we discussed several problems (and solutions) that we encountered during the development of Erasmus language. Particularly, we discussed that our approach is not suitable to find deadlocks in cyclic communication patterns. Static detection of deadlock is a difficult task mainly because of the state-explosion problem. Thus, it would be useful to develop at least one of the following:
  - A translator that translates Erasmus programs into mCRL2 programs in case of cyclic configurations, as is implemented in [HSMG07]. This would allow programmer to statically check their programs in a brute force manner.
  - A dynamic debugger that allows programmer to dynamically detect deadlock, as is developed in [BL12].

The above list of research topics emphasizes the further development of techniques for the Erasmus programming language, for which this thesis provides a starting point.

Bibliography

[ADZ$^+$12] Torben Amtoft, Josiah Dodds, Zhi Zhang, Andrew Appel, Lennart Beringer, John Hatcliff, Xinming Ou, and Andrew Cousino. A Certificate Infrastructure for Machine-Checked Proofs of Conditional Information Flow. In Pierpaolo Degano and Joshua Guttman, editors, *First conference on Principles of Security and Trust (part of ETAPS 2012)*, volume 7215 of *LNCS*, pages 369–389. Springer-Verlag, 2012.

[And94] Henrik Reif Andersen. A Polyadic Modal μ-Calculus, 1994.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to The ISO Specification Language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, March 1987.

[BBR10] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, December 2010.

[BK82] J. A. Bergstra and J. W. Klop. Fixed Point Semantics In Process Algebra. 1982.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of Communicating Processes with Abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BL12] Andrew Bate and Gavin Lowe. A Debugger for Communicating Scala Objects. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Baekgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 135–154, August 2012.

[Boe05] Hans-J. Boehm. Threads Cannot Be Implemented As a Library. *SIGPLAN Not.*, 40(6):261–268, June 2005.

[BR00] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and Its Applications. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 322–323, London, UK, UK, 2000. Springer-Verlag.

[Bro84] Brookes, S. D. and Hoare, C. A. R. and Roscoe, A. W. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560–599, 1984.

[Bro07] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, July 2007.

[BS83] G. N. Buckley and Abraham Silberschatz. An Effective Implementation for the Generalized Input-Output Construct of CSP. *ACM Trans. Program. Lang. Syst.*, 5(2):223–235, April 1983.

[Cor84] Corp, Inmos. *Occam Programming Manual*. Prentice Hall Trade, 1984.

[Dem98] Erik D. Demaine. Protocols for Non-Deterministic Communication over Synchronous Channels. In *In Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP98*, pages 24–30. IEEE Press, 1998.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, August 1975.

[Eas05] Ian R. East. Interfacing with Honeysuckle by Formal Contract. In Jan F. Broenink, Herman Roebbers, Johan P. E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, pages 1–11, September 2005.

[Flo67] R. W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[Geo98] Geoff Barrett. *Occam 3 Reference Manual*. Technical report, Inmos Limited, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJ10] Peter Grogono and Nima Jafroodi. A Fair Protocol for Non-deterministic Message Passing. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 53–58, New York, NY, USA, 2010. ACM.

[GMR$^+$07] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The Formal Specification Language mCRL2. In *In Proceedings of the Dagstuhl Seminar*. MIT Press, 2007.

[GP90] J.F. Groote and A. Ponse. *The Syntax and Semantics of mCRL*. Centrum voor Wiskunde en Informatica, 1990.

[GS08a] Peter Grogono and Brian Shearing. Concurrent Software Engineering: Preparing for Paradigm Shift. In *Canadian Conference on Computer Science & Software Engineering (C^3S^2E)*, pages 99–108, May 2008.

[GS08b] Peter Grogono and Brian Shearing. MEC Reference Manual. Technical Report TR E-06, February 2008.

[GS08c] Peter Grogono and Brian Shearing. Modular Concurrency: A New Approach to Manageable Software. In *3rd International Conference on Software and Data Technologies (ICSOFT 2008)*, pages 47–54, July 2008.

[Han02] Per Brinch Hansen. The Origin of Concurrent Programming. chapter Joyce: A Programming Language for Distributed Systems, pages 464–492. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[HM85] Matthew Hennessy and Robin Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM*, 32(1):137–161, January 1985.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of The ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of The ACM*, 21(8):666–677, August 1978.

[Hol03] Gerard Holzmann. *Spin Model Checker, The Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HSMG07] H. Hojjat, M. Sirjani, M. R. Mousavi, and J. F. Groote. Sarir: A Rebeca to mCRL2 Translator. In *Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, ACSD '07, pages 216–222, Washington, DC, USA, 2007. IEEE Computer Society.

[J.F08] J.F. Groote and J. Keiren and A. Mathijssen and B. Ploeger and F. Stappers and C. Tankink and Y. Usenko and Weerdenburg, M. van and W. Wesselink and T. Willemse and Wulp, J. van der. The mCRL2 toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008.

[JG13] Nima Jafroodi and Peter Grogono. Implementing Generalized Alternative Construct for Erasmus Language. In *Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE 2013, part of Comparch '13 Federated Events on Component-Based Software Engineering and Software Architecture, Vancouver, Canada, June 17-21 2013. ACM.

[Kna92] Frederick Knabe. A Distributed Protocol for Channel-Based Communication with Choice. In *Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '92, pages 947–948, London, UK, UK, 1992. Springer-Verlag.

[Led83] Henry Ledgard. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[Lee06] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[Low11] Gavin Lowe. Implementing Generalised Alt. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 1–34, June 2011.

[Mat04] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *In PADL, volume 3057 of LNCS*, pages 56–70. Springer, 2004.

[Mca63] John Mcarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[MW97] J. M. R. Martin and P. H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4):215–232, 1997.

[OB04] B. Orlic and Dr.ir. J.F. Broenink. Redesign of the C++ Communicating Threads Library for Embedded Control Systems. In F. Karelse, editor, *5th Progress Symposium on Embedded Systems, Nieuwegein, The Netherlands*, pages 141–156. STW Technology Foundation, 2004.

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[Pet05] Peter H. Welch and Frederick R. M. Barnes. Communicating Mobile Processes: Introducing Occam-π. In *In 25 Years of CSP*, pages 175–210. Springer Verlag, 2005.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[RT85] Thomas Räuchle and Sam Toueg. Exposure to Deadlock for Communicating Processes is Hard to Detect. *Inf. Process. Lett.*, 21(2):63–68, 1985.

[Sam08] Adam T. Sampson. Two-Way Protocols for Occam-π;. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 85–97, September 2008.

[SE08] Matthew Sackman and Susan Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, June 2008.

[SHW00] Nan C. Schaller, Gerald H. Hilderink, and Peter H. Welch. Using Java for Parallel Computing: JCSP versus CTJ, a Comparison. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226, 2000.

[Sim03] Simon Gay and Vasco Vasconcelos and Antonio Ravara and Simon Gay and Vasco Vasconcelos and Antnio Ravara. Session Types for Inter-Process Communication, 2003.

[SS71] Dana Scott and Christopher Strachey. Toward A Mathematical Semantics for Computer Languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., April 1971. Polytechnic Press.

[Suf08] Bernard Sufrin. Communicating Scala Objects. In Peter H. Welch, Susan Stepney, Fiona Polack, Fred R. M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *CPA*, volume 66 of *Concurrent Systems Engineering Series*, pages 35–54. IOS Press, 2008.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 1994.

[Wel00] Welch, Peter H. and Martin, Jeremy M. R. A CSP Model for Java Multithreading. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, PDSE '00, pages 114–122, Washington, DC, USA, 2000. IEEE Computer Society.

[Wel10] Welch, Peter and Brown, Neil and Moores, James and Chalmers, Kevin and Sputh, Bernhard. Alting Barriers: Synchronisation With Choice in Java Using JCSP. *Concurr. Comput. : Pract. Exper.*, 22(8):1049–1062, 2010.

Appendix A

A Fair Protocol for Non-deterministic Message Passing

A.1 Overview

This appendix describes our initial design of the generalized alternative construct of the Erasmus programming language. Our algorithm [GJ10] is similar to Knabe's algorithm [Kna92] in the sense that we use asynchronous (buffered) messages to implement the (synchronous) generalized alternative construct, but has significant extensions.

The rest of this appendix is structured as follows. Section A.2 describes the Knabe's algorithm. Section A.3 describes our fair distributed protocol, and presents the analysis of the algorithm. Finally, Section A.4 summarizes.

A.2 The Distributed Protocol

The protocol described in this section is essentially Knabe's algorithm [Kna92]. We describe it to provide background for our protocol, which is similar but has significant extensions.

Processes communicate with one another via *channels*. Each process is connected to any number of channels and each channel is connected to at least two processes. In order for processes P_1 and P_2 to communicate via channel C, there must be a *match*. A match occurs if either: P_1 is ready to send a message with tag T and P_2 is ready to receive a message with tag T, or *vice versa* (i.e., P_1 receives and P_2 sends). The *tag* of a message specifies its type and possibly other characteristics; the important point is that matched tags ensure meaningful communication. The protocol requires an ordering on processes; we will assume that each process has a unique identifier (UID) and that, if u_1 and u_2 are UIDs of distinct processes, then either $u_1 < u_2$ or $u_1 > u_2$.

A channel is an active process that executes as a single, non-terminating loop. The task of a channel is to find matches between pairs of processes. A process that is ready to communicate sends a *request* to one or more of its channels. The channels maintain two FIFO queues, one for send requests and the other for receive requests from processes. A channel remains idle as long as either queue is empty and becomes active when both of its queues are non-empty, at which point it enters a synchronization sequence with two phases, see Figure 40.

```
procedure CHANNEL()
    step ← Phase 1;
    Phase 1:
    $(P_<, P_>)$ ← FINDMATCH(Queue$_1$, Queue$_2$);
    query ← GENERATEQUERY($(P_<, P_>)$);
    query.ReqType ← L;
    SEND($P_<$, query);
    reply = RECEIVE();
    if reply = Yes then
        step ← Phase 2;                                ▷ A potential match
    else
        Put $P_>$ back in the Queue;
        step ← Phase 1;
    Phase 2:
    query.ReqType ← H;
    SEND($P_>$, query);
    reply ← RECEIVE();
    if reply = Yes then                                ▷ Match found
        INFORM($P_<, P_>$);
        step ← Phase 1;
    else                                               ▷ Match failed
        query.ReqType = Abort;
        SEND($P_<$, query);
        Put $P_<$ back in the Queue;
        step ← Phase 1;
```

Figure 40: Pseudocode for channels

During the first phase, the channel, C say, chooses two complementary processes: that is, a process that wishes to send a message of type T and another processes that wishes to receive a message of type T. First, C picks the process $P_<$ with lower UID, regardless of whether it is the sender or the receiver. It sends the message L to $P_<$, requesting $P_<$ to temporarily commit to this communication and defer other signals it might receive. Then the channel waits for a reply.

If $P_<$ replies with `No`, C returns to its waiting state. If $P_<$ replies with `Yes`, C enters the second phase of the synchronization sequence. It sends the message H to the other process of the pair, $P_>$, requesting it to commit to this communication and reject all other signals. Again, C waits for a reply.

If $P_>$ replies `Yes`, C sends a `Ready` signal to $P_<$ and $P_>$, informing them that they can communicate, removes the corresponding entries from its queues, and returns to its waiting state. If $P_>$ replies `No`, C discards $P_>$'s request and sends a `Release` message to $P_<$, releasing it from its commitment. However, $P_<$'s request remains in C's channel.

A process P must also follow a procedure when it enters a selection. Its first step is to send a send or receive request to each of the channels involved in the selection; then it waits. A channel may reply with either H or L, depending on whether the process has the higher or lower UID of the proposed communication. If P receives H, it replies `Yes` to this channel, sends `No` to all of the other candidates for communication, and starts to transfer data.

If P receives L from C, it replies `Yes` and waits. Any signals that it receives from channels other than C are queued. Eventually, P will receive either a `Ready` message or a `Release` message. If the message is `Ready`, P sends `No` to all the losing channels and communicates. If the message is `Release`, P processes the first message on its queue, if there is one, otherwise waits for a message. The procedure ensures that exactly one communication occurs each time the selection is processed.

A.2.1 Analysis

Knabe proves that the distributed protocol cannot deadlock [Kna92]. The essence of the proof is that deadlock requires symmetry and that the ordered UIDs break the symmetry. However, the protocol does not ensure fairness. Although the whole system cannot become blocked, an individual process may wait indefinitely to communicate. This is called *starvation*.

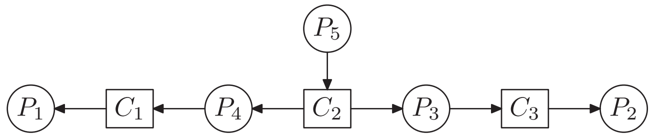


Figure 41: Knabe's algorithm allows P_2 to starve

To clarify it, consider a group of connected processes Figure 41, in which the P_i are processes and the C_j are channels. A directed edge from a channel to a process denotes an outstanding receive request, and similarly a directed edge from a process to a channel denotes an outstanding send request. With the distributed protocol, it is possible for P_2 to be starved. All three channels will each start by attempting to acquire their low-numbered process. Without loss of generality consider a case in which C_1, C_2, and C_3 win to acquire processes P_1, P_3, and P_2 respectively. Now all three channels will attempt to acquire their high-numbered process. Lets assume that C_1 acquires P_4 and C_2 acquires P_5. C_3 cannot acquire P_3 since it is being held by C_2. This leads to a situation in which C_1 and C_2 end up finding matches (P_1, P_4), and (P_3, P_4) respectively.

If this cycle happens repeatedly;that is P_1 and P_4 communicate frequently using C_1 and P_3 and P_5 communicate frequently using C_2, then signals from C_3 will always be discarded, starving P_2.

A.3 The Fair Distributed Protocol

We describe an implementation of the select statement that provides nondeterministic choice, avoids deadlock, and treats all processes fairly. A select statement may have several branches that are a mixture of sends and receives. Channel behavior is a bit different from the distributed protocol described in the previous. The only difference is that each channel piggybacks the attributes of the other process in the match along with the signal it is about to send to a process. These attributes are shown in Figure 42.

Field name	Description
`messageId`	Unique identifier for this signal
`messageTag`	The tag/type associated with the message
`thisBranchNum` `otherbranchNum`	The numbers of the branches within the **select** statement
`thisBranchWeight` `otherBranchWeight`	The weights of the branches
`thisMinWeight` `otherMinWeight`	The minimum weights of the branches
`thisProcessId` `otherProcessId`	The UIDs of the processes

Figure 42: Data structure of a signal

The additional feature of the protocol is a weight attached to each branch of a select statement. The weight may be an integer counter or a time-stamp; the important point is that it increases monotonically as the program runs. A counter is easier to implement but may overflow. A timestamp is preferable, but must be fine-grained because communications may occur very frequently.

Each process must follow a procedure when it enters a selection (see Figure 43). Its first step is to send a send or receive request to each of the channels involved in the selection; then it waits. These requests carry all the attributes of the requesting process such as the weight of the branch, minimum weight of all the branches in the select statement, and etc. Unlike Knobe's algorithm in which all processes always respond `Yes` to the very first signal they receive, our implementation takes a different approach.

To clarify this, let's consider the case in which the process $P_<$ with lower UID has received its first signal from a channel. This signal indicates that if $P_<$ can commit itself temporarily to the signaling channel and delay others. Process $P_<$ examines the weights W of the branches that will be used to communicate; that is if $W_<$ is the lowest weight in the branches of $P_<$'s select statement, and the chosen branch of $P_>$'s select statement, say $W_>$, also has the lowest weight, then $P_<$ replies `Yes` and waits; otherwise it replies `No`.

If $P_<$ replies `No`, it has nothing further to do: the attempted match has failed, and $P_<$ continues to wait in its select statement. However, if $P_<$ replies `Yes`, it waits for another signal. If it receives `Ready`, it should have been sent by the higher UID process, so it sends `No` to any other waiting channels and proceeds with communication. It finally increases the weight of the branch of the select statement that was used in the communication.

But if it receives `Abort`, it should have been sent by the channel indicating that the match has failed; $P_<$ remains in its select statement, considering requests from other channels. The other case in which $P_>$ has received its first signal from a channel is almost same as above. In this case, process $P_>$ examines the weights W of the branches that will be used to communicate; that is if $W_>$ is the lowest weight in the branches of $P_>$'s select statement, and the chosen branch of $P_<$'s select statement also has the lowest weight, then $P_>$ replies `Yes` to the signaling channel, it then sends an abort signal to any other pending channels followed by a `Ready` signal to the lower UID process for the actual data communication, and finally it increases the weight of the branch of the select statement that was used in the communication. Otherwise it replies `No` to the signaling channel and proceeds with any other signals it receives.

The foregoing discussion has a few gaps in it that we will now fill. First, each process makes send or receive requests only to channels of those branches having the lowest weights. Each process can have more than one branch having the lowest weight. By doing so each process avoid sending extra requests which are guaranteed to be responded with `No`. This results in receiving less signals which reduces the number of messages needed to find a match.

Second, in the case where there is a sender or a receiver process with no select statement, the requesting process only sends a committed send or a committed receive request to the channel and waits for the actual data communication. These committed requests inform channels that the requesting process has already pre-committed itself and that there is no need for the channel to ask for it.

Third, a process executing a select statement may receive `Abort` signals from all of its branches. If this happens, it simply starts everything all over again by sending new requests. Failing to do this could lead to deadlock.

Finally, it is clear that signals contain more information than just the type of the data. In fact, a signal is a fixed-size block of data containing the fields shown in Figure 42. Fields after the first two come in pairs with a `this` field referring to the initiating process, and a `other` field referring to the responding process. Using the names in this table, each process needs to check the fairness condition by performing —*fair(signal)* before replying to any queries. The body of this function is also given in Figure 43.

```
procedure PROCESS()
    step ← STEP 1;
    STEP 1:
    for all $C \in$ branches do
        SEND($C$, req)                                        ▷ $req \in \{send, receive\}$
        step ← STEP 2;
    STEP 2:
    query ← RECEIVE();
    if FAIR(query) = true then
        step ← query.ReqType;                                 ▷ $ReqType \in \{L, H\}$
    else
        SEND(query.QueryingChannel, NO);
        branch[ query.branchNO ] ← Aborted;
        step ← STEP 2;
    STEP L:
    SEND(query.QueryingChannel, Yes);
    reply ← RECEIVE( );                                       ▷ $reply \in$ { Ready, Abort }
    if reply = Ready then
        **Actual Data Transfer**
        ++Weight[reply.branchNo];
        ABORTOTHERCHANNELS()
    else if reply = Abort then
        branch[ reply.branchNO ] ← Aborted;
        if branch[ i:0 to n ] = Aborted then
            step ← STEP 1;                                    ▷ Start from scratch
        else
            step ← STEP 2;                                    ▷ Continue processing other queries
    STEP H:
    SEND(query.QueryingChannel, Yes);
    SEND(query.$P_<$ , Ready);                                ▷ $P_<$ = process having lower UID
    **Actual Data Transfer**
    ++Weight[query.branchNO];
    ABORTOTHERCHANNELS();

procedure FAIR(signal: query)
    if signal.thisBranchWeight=signal.thisMinWeight &&
                        signal.otherBranchWeight=signal.otherMinWeight then
        return true
    return false;
```

Figure 43: Pseudocode for processes

A.3.1 Deadlock

We show that deadlock cannot occur if there is a feasible match. Assume the contrary: there is a feasible match and that deadlock has occurred. This would imply that some processes have received a phase-one signal from one of their channels but have failed to find a match. However, this cannot happen because the process UIDs are ordered and only the low-numbered process of a pair receives a request during the first phase. Therefore, there must be at least one process, the one with the highest UID in the system, that has *not* received a phase-one signal and is available for matching.

A.3.2 Starvation

There are two situations in which starvation might occur. As an example of the first situation, consider a system in which a channel connects a single server P_0 to multiple clients P_1, $P_2, \ldots,$ P_n, as shown in Figure 44. Client requests are stored in the FIFO queue of the channel. There is a possibility that one or more of the clients might be starved. However, provided that the server continues executing, the protocol ensures that every request from a client will eventually be served. In this example, our protocol behaves in exactly the same way as Knabe's protocol.

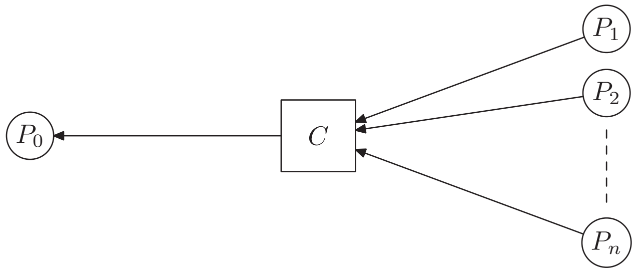


Figure 44: Starvation avoided

Figure 45 shows the other situation. With the distributed protocol, P_2 may be starved. P_0 is the process with lowest UID and the protocol allows it to send `Yes` to all signals from C_1 but none from C_2, thereby starving P_2. With our protocol, the weights on the branches prevent starvation. After P_0 and P_1 have communicated once, $W(P_0, C_1) = 1$ and $W(P_0, C_2) = 0$. This ensures that the next time P_0 communicates, it will be with P_2. Neither P_1 nor P_2 can starve.

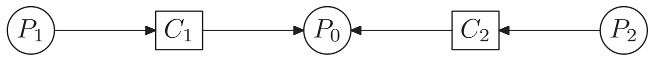


Figure 45: Starvation with the distributed protocol

In the form described, the fair distributed protocol would have a serious problem: one slow process, treated fairly, could slow down the entire system. The **select** statement in Erasmus, however, provides for the declaration of a *policy*. Communication is implemented as above for **select** statements that specify the policy **fair**. If **fair** is not specified, the process is not obliged to use branch weights in channel selection.

A.3.3 Cost

The cost of the algorithm is measured by the number of messages required to establish a communication. These messages include initial send or receive requests by processes, channels signals, responses to the signals, actual data communication, abort messages, and finally the wake-up signal from the sender to the receiver process.

To compute the cost of our algorithm several cases should be considered. The easiest and simplest case is where there are only two processes connecting through a channel, without any `select` statements. The total cost of the algorithm is 5 messages; two committed send and receive requests, a `Ready` message to the sender process, actual data transmission, and a `Ready` message to the receiver process.

The other case is where both processes execute their select statements. For this case we can define a lower and a higher bound for the cost. The lower bound is achieved when the process is the one with the highest UID in the system. This process responds `Yes` to the first signal it receives from one of its channels. Without loss of generality assume that this process, the one with the highest UID, has n branches and that the other communicating process has m branches; So, the lower bound on the cost is: $n + m$ requests, two signals, two `Yes` messages, a `Ready` message to the sender process, actual data transmission, and a `Ready` message to the receiver process, followed by $n + m - 2$ `Abort` messages to the losing channels leading to the following formula for the lower bound, L:

$$L = 2(n + m) + 5.$$

Deriving an upper bound for the number of messages is difficult. As we have seen, it is possible for a process to continuously receives `Abort` or `No` messages from all of its branches for a while forcing the process to resend all of its requests again. The question is that for how many times would a process have to resend its requests? Equivalently, what is the maximum number of times that a process receives `Abort` messages from all of its branches?

To answer the above questions let's consider a process P having the lowest UID in the system. Without loss of generality, assume that P has n branches and $P_1, , P_n$ are n distinct processes which are connected to these branches. If b_i is the number of branches of P_i, and if all processes are executing their select statements for the first time then after exactly $B - 1$ times failing where $B = \min\{b_1, \ldots, b_n\}$ the process P eventually communicates with a process.

Therefore, in each round process P makes n requests, followed by n signals, n `Yes` messages, followed by n `Abort` messages. Eventually, in round B, there are n requests, n signals, two `Yes` signals, a `Ready` message to the sender process, actual data transmission, and a `Ready` message to the receiver process. This leads us to the following formula for the upper bound, U:

$$U = 4n(B - 1) + 2n + 5.$$

Finding an upper bound in the case where channels supports many to many communications is similar to the above. The upper bound of messages is calculated separately for each process in the system depending on the number of channels it has, the number of processes sharing channels, and the number of branches its communicating processes have.

A.4 Summary

We have described a fair, distributed protocol that allows an arbitrary network of processes linked by channels to communicate fairly and without deadlock. Processes may perform selection on both sends and receives and may be connected to an arbitrary number of channels. Conversely, a channel may be connected to an arbitrary number of processes. The general case requires extensive handshaking, but run-time analysis might allow more specialized and efficient techniques to be used in particular cases. For example, a JIT compiler could determine the number of processes connected to a channel and generate simplified code for the probably common case in which only two processes are involved. Since global static analysis is not required, processes may be compiled independently and linked dynamically. This last feature is essential for the construction of large-scale, distributed systems.