Verification of the Fiasco IPC Implementation

by
Endrawaty

Thesis Report
International Master Program
Computational Logic
March 2005

Institute of Theoretical Computer Science
Department of Computer Science
Dresden University of Technology - Germany

Overseeing Professor : Prof. Dr.-Ing. habil. Horst Reichel
Supervisor: Dr. Hendrik Tews

Master Thesis Assignment (Computational Logic)
Student: Endrawaty Endrawaty   Matr. No. 2982167

Verification of the Fiasco IPC implementation

Context The microkernel Fiasco is an implementation of the L4 interface in C++. It is used in the context of the DROPS-project. One of the services that is provided is inter-process communication (IPC). In Fiasco IPC is the only means to send data from one process to an other one. Fiasco's IPC is special in the sense that

- the kernel is peemptible almost everwhere (so an ongoing IPC might be preempted by some other activity)
- the IPC is stateless, that is the data send is not cached in the kernel, it is copied directly from the sender to the receiver (so the kernel must suspend one IPC partner until the other one is ready)

Assignment The aim of this master thesis is to develop and verify an abstraction of Fiascos IPC implementation. The abstraction should have the form of a finite state automaton, where one automaton models one process that is engaged in an IPC. For the verification several such automaton are set in parallel. Properties of interest for the verification are dead lock, live lock, safty properties (of availability) or liveness properties (message delivery).

The student can choose his preferred formalism for the IPC model. All of CCS, CSP, π-calculus, or promela are possible. The properties should be formulated in a suitable modal logic. They should be verified with a suitable proof assistent or model checker.

overseeing Professor: Prof. Horst Reichel

Advisor: Dr. Hendrik Tews

Begin: October 1st 2004

Due date: April 1st 2005

Dresden, October 29th 2004

Statement of Academical Honesty

Hereby, I declare that the work of this master's thesis is completely the result of my own work, except where otherwise indicated. All the resources I used for this master's thesis are given in the list of references.

Date: March 16th, 2005

Signature:

Acknowledgements

I would like to express my gratitude to the following people below, who has helped me in numerous kinds of way, during the work of my master's thesis:

- Hendrik Tews: for all his support and guidance during the time I worked on my thesis. I learned a lot of new important knowledge from him, ranging from technical stuff until various strategies in tackling big and delicate problems. This master's thesis has been a wonderful experience for me thanks to him.
- Michael Hohmuth: for his time and patience in answering my questions about Fiasco IPC.
- Gerard J. Holzmann: for his insightful guidance on Promela and SPIN.
- Ari Saptawijaya: for proof-reading my thesis report.
- Ren Jie Yun: for being a kind dorm-mate. She often prepared meals for me voluntarily, because she knew I was pretty much occupied with my thesis.
- Andi Wijaya Tan: for being my pillar and my shelter. His love and encouragement has been my source of inspiration in completing my master study in these two and a half years.
- My parents, sisters, and brother: for always believing in me.

I could pursue this master study thanks to the financial support that I have received from the International Quality of Network scholarship, for one year and 5 months (Oct 02 - Feb 04). Without this scholarship, I would not be able to come to and study in Dresden. I am grateful to get this chance to study in Computational Logic, which to me personally is a very challenging subject to stimulate a new way of thinking. Therefore, I owe my thanks to the people who have established Computational Logic program and who have given me the scholarship.

Abstract

Fiasco is a microkernel which is developed in the context of DROPS (Dresden Real-Time Operating Systems)-project. One of Fiasco's system calls is IPC (Inter-Process Communication). In this master's thesis, a model for Fiasco IPC has been built. The real Fiasco IPC code which was written in C++ has been reverse-engineered into a model in Promela. After the model was built, some properties were proposed and verified using SPIN. Those properties are safety properties (freedom of deadlock, some "bad" things that should never happen) and liveness properties (freedom of certain bad cycle, some "good" things eventually happen).

Contents

1 Introduction

Software systems may range from simple to complex ones. The more complex a system, the bigger possibilities it might have errors. Some errors can be so subtle that they are usually unseen at the initial testing phase. Meanwhile, the ultimate goal of software engineering is to deliver a system which is bugs-free and provably correct. But most situations show that software development spends most of its time in fixing the errors.

Two significant aspects of systems which contribute to their complexity are concurrency and nondeterminism. Concurrent and nondeterministic systems produce a large number of possible executions. When they contain errors, testing method would not be a wise solution to trace the sources of errors. It will be hard to have a testing method that could provide a complete coverage for all possible errors of complex systems. A famous quote from Dijkstra applies in this case: "Program testing can be used to show the presence of bugs, but never to show their absence." The two main problems in testing such distributed systems are [1]:

1. limited controllability of events in distributed systems executions and
2. limited observability of those events.

In order to prove the correctness of distributed systems, we can turn to the so-called *formal methods*. Taken from http://www.fmeurope.org/, formal methods are defined as mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems. Two well-established approaches to formal verification of hardware and software systems are theorem proving and model checking.

Theorem proving is a technique where both the system and its desired properties are expressed as formulae in some mathematical logic [11]. This mathematical logic defines a set of axioms and a set of inference rules. Strictly speaking, theorem proving is the process of finding a proof of a property from the axioms of the system.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model [11]. The check is performed as an exhaustive state space search which is guaranteed to terminate since the model is finite. Model checking produces counterexamples, which usually represent subtle errors in design, and thus can be used to aid in debugging [11]. Despite these advantages, model checking suffers from the state explosion problem.

In this master's thesis, we developed a model abstraction for Fiasco IPC (Inter-Process Communication) and applied model checking to verify it. The microkernel Fiasco is an implementation of the L4 interface in C++. It is used in the context of DROPS (Dresden Real-Time Operating System Project). DROPS is a research project aiming at the support of applications with Quality of Service requirements. The Fiasco IPC abstraction is focused on short IPC with features such as thread lock and zero timeout. The abstraction should have the form of a finite state automaton, where one automaton models one process that is engaged in an IPC. For the verification several such automaton are set in parallel.

The preferred formalism for the IPC model is Promela (Process Meta Language). Promela is a nondeterministic language which is loosely based on Dijkstra's guarded command language notation. Promela also borrows the notation for input output operation from Hoare's CSP (Communicating Sequential Processes). The choice on Promela is decided based on the fact that Fiasco is written in C++, while Promela's syntax resembles the syntax of C programming language.

SPIN (Simple Promela Interpreter) will be used as the model checker. SPIN receives models specified in Promela. The properties can be expressed as LTL (Linear Temporal Logic) formulae for SPIN to verify. Properties of interest for the verification are deadlock, livelock, safety properties or liveness properties. SPIN has been developed at Bell Labs since the eighties. It has been available freely since 1991 at [2]. It continues to be upgraded to keep in line with new developments in the field of formal verification. In April 2002, SPIN was awarded the System Software Award for 2001 by ACM. Some documentations, manuals and tutorials about Promela and SPIN can be found in [1, 2, 8, 9, 10].

This master's thesis report is organized as follows.

- Section 2 gives a brief overview of Fiasco IPC.
- Section 3 describes some interesting features of Promela which are used in modeling the Fiasco IPC.
- Section 4 provides explanation of how we abstract the real Fiasco IPC code until we come up with our model.
- Section 5 reports the various simulation and verification efforts for our model.
- Section 6 draws conclusions from this master's thesis, both in the context of our experience in translating C++ code to Promela and our experience with Promela and SPIN. This section also proposes some future works.

2 Fiasco IPC

This section is intended to provide an overview about Fiasco IPC. Before doing so, we feel the necessity to provide some brief information about L4 and Fiasco itself in Section 2.1. Further, we summarize the important facts about IPC in Section 2.2. For more detail and comprehensive reference about Fiasco IPC, we suggest to consult [3, 4, 5].

2.1 L4 and Fiasco

Fiasco is a new implementation of the L4 interface for the x86 architecture. L4 itself is an operating system microkernel [7]. By microkernel, we mean that it alone is not an operating system in the traditional sense, but rather constitutes a minimal base on which a variety of complete operating systems can be built. A microkernel is an operating system kernel which provides only essential services such as tasks, threads, inter-process communication (IPC), and memory management primitives.

L4 is a microkernel interface defined by Jochen Liedtke [7]. Currently, there exist L4 implementations for:

- x86 architecture (by Jochen Liedtke),
- MIPS CPU (by University of New South Wales),
- Alpha CPU (by Sebastion Schonberg).

Fiasco is meant to replace the L4/x86 microkernel and it is designed with the following intended characteristics[5]:

- having good real-time properties. This means that Fiasco kernel should be preemptible: interrupts shall be deliverable at virtually any time.
- maintainable. To achieve this, Fiasco is written in a high-level language, C++. Only five percent of assembler codes is allowed when we need to implement low-level operations or to meet the performance requirements. The source code of Fiasco shall be organized in subsystems with clean programming interfaces so that it is easy to replace subsystem implementation.
- not need to be portable, but it should serve as a model for future L4 implementation,
- distributed under a freeware (open source) license. This means that Fiasco shall not contain components which are not freeware, and that all tools and documentation necessary to build and use Fiasco should also be freeware.

2.2 How IPC Works

IPC is one of the system calls in Fiasco. It is the secure, kernel-assisted message transfer between a sender and a receiver [3]. Parties that are involved in an IPC are threads which act as a sender and a receiver, respectively. A successful IPC operation consists of a handshake and a message transfer between a sender and a receiver [3]. In L4, there is only one kind of IPC receiver, that is the thread, and two kinds of IPC senders, those are thread and (kernel-internal) hardware-interrupt signal. In the operating system world, thread is defined as a basic unit of CPU utilization and it must live inside a process [7].

During IPC, the sender executes a send operation and the receiver executes a receive operation. Send operations are always addressed to a receiving thread. Receive operations can be parameterized to only accept messages from a specified sender (*closed wait*) or from any sender (*open wait*). Both operations have a timeout parameter that limits the duration of blocking while waiting for an IPC partner.

The maximum possible operations in one IPC system call are one single send operation, followed by one single receive operation. This particular system-call mode is named *call* or *reply-and-wait*, depending on whether the involved receive operation is a closed or an open wait, respectively.

The server usually uses a zero timeout for its send operations to prevent unresponsive clients from stalling the server. L4 supports this programming convention by guaranteeing that a client that sent a request using a call is ready to receive an answer as soon as the server received the request. In other words, shift from send mode to receive mode in calls and reply-and-wait is atomic.

2.2.1 IPC States

Two threads involved in an IPC keep track of their own state and also modify their IPC partner's state. That is the reason why IPC states need to be represented as explicit data, i.e, as part of the each thread's state word [3]. The states are defined by bit patterns in the state word, as shown in Table 1.

State	State flags used in IPC								
	ready	receiving	ipc	send	busy	busy long	poll	poll long	cancel
Sender states									
send prepared	+		+	+			+	-	
sleeping	-		+	+			+	-	
woken up	+		+	+			-	-	
long IPC in progress	+		+	+			-	-	
page fault in IPC window	+		+	+			-	+	
page-in wait	-		+	+			-	+	
after send	+			-					
Receiver states									
setup	+		+						
prepared	+	+	+	-	-	-			
going to rendezvous	+	+	+	-	+	-			
waiting	-	+	+	-	+	-			
in long IPC	-	+	+	-	-	+			
page-in	+	+	+	-	-	+			
after receive	+			-					

Legend: + = flag set; - = flag cleared; otherwise, flags can be set or cleared.

Table 1: Sender and Receiver States

Each state transition changes the bit pattern in a unique way. But two states can have the same pattern if there is no transition between them. It is important to notice, that not all bits are significant for each state. This allows a thread to stay in receiver-setup state while carrying out a send operation. Therefore, for a combined send-receive IPC, Fiasco can set up the receive operation before carrying out a send operation. This allows the thread to atomically switch from a sender state to a receiver state by clearing the `"send"` flag. In the sender states, the additional flags used for receiving are irrelevant.

Fiasco must check the validity of every state transition. Since Fiasco allows preemption and parallel execution of sender and receiver, state checking and modification should be atomic from the IPC's partner point of view.

2.2.2 IPC Phases

An IPC can maximally contains one send operation and one receive operation. A thread which is engaged in such an IPC performs the IPC in the following order:

1. setup for receive operation (as a receiver) ,

2. do send operation (as a sender),

3. do receive operation (as a receiver).

It is important to note from the order above that send operation is *always* performed before receive operation. But receive setup phase always takes place before a send operation [3]. This adopted convention facilitates atomic switching from the send part to the receive part. In effect, this fulfills the L4 specification requirement that threads must accept reply IPC immediately, without requiring a timeout in the sender.

In general, both send and receive IPC operations can be grouped into four phases. Those phases are described as follows:

- **Setup**
  During setup, a thread can do the following based on send or receive part:
  - In receive part: the thread sets up its TCB (Thread Control Block) so that it can be a receiver. For a closed-wait receive, the thread sets its TCB's IPC-partner attribute to point to the desired IPC partner's TCB.
  - In send part: the thread set its TCB's IPC send-partner attribute to point to the desired IPC partner's TCB. Then it puts itself into state `"send prepared"`. Next, it enqueues its TCB in the receiver's sender list.
- **Rendezvous**
  During rendezvous, the sender is the active party and the receiver becomes passive (goes to sleep). The sender will wake up the receiver when the receiver needs to page in a virtual-memory page (in the receiver's address space), or when the IPC has been finished.

  Once the receiver has entered the `"prepared"` state, senders can asynchronously rendezvous using method `ipc_send_regs`, for example, by putting the receiver into the final state (for short IPC) or into state `"in long IPC"` (for long IPC).

  However, the receiver would normally proceed to state "going to rendezvous" where it checks if a sender has queued in the receiver's sender queue. If that is the case, the receiver wakes up the sender using the `ipc_receiver_ready` operation; the sender becomes active, rendezvouses and executes the IPC. The sender in the end will put the receiver into final state or state `"in long IPC"`.

  If there is no sender waiting for a rendezvous, the receiver tries to proceed to state waiting where it sleeps until a sender rendezvous. This change from state "going to rendezvous" to state waiting only succeeds if no sender has asynchronously put the receiver into another state (prepared, in long IPC, or final), in which case execution proceeds there. Meanwhile, the sender prepares for IPC by entering state send prepared. It queues in the receiver's sender queue and attempts a first rendezvous using `ipc_send_regs`. If successful, the sender directly proceeds to its final state (for short IPC) or "long IPC in progress" (for long IPC). Otherwise it starts sleeping by going to state sleeping. When a receiver sends an `ipc_receiver_ready` request, the sender continues in state woken up. It then switches back to state send prepared where it retries the `ipc_send_regs` operation.
- **Data Transfer**
  The receiver enters this phase only when the sender starts a long message transfer by putting the receiver into state `"in long IPC"`. On the other hand, the sender

enters this phase when `ipc_snd_regs` cannot complete the IPC without looking at IPC-message buffers in user memory. Our model does not include this phase since we only model short IPC.

- **Finish**
  For the receiver, IPC finishes when the `"ipc"` state flag is removed from its state word. For the sender, send operation finishes when an error condition removes the `"ipc"` state flag, and it also has to monitor its state word to detect this conditon. Otherwise, the send operation ends when the `"send"` flag is removed, that is when the sender enters its final state. When reaching that state, the sender needs to check if the send operation is followed by a receive operation. If it is not followed by a receive operation, then the `"ipc"` state flag is removed also.

3 Promela

This section is meant to provide an overview of Promela. Promela is a specification language which is accepted by SPIN. Promela provides formalisms to abstract distributed systems and produces models for verification in SPIN. SPIN is a model checker tool to check those verification models. More hands-on experience with SPIN will be covered in the simulation and verification phases. We will explain about those two phases in the Section 5.

As a specification language, Promela has features which would help its users in modeling process synchronization and coordination. It is intended for systems description language, instead of implementation language. In line with these facts, Promela has few computational functionalities. Furthermore, it has no notion of time or clock and it does not have floating point numbers.

For detail reference about Promela, we suggest to consult the book "The SPIN Model Checker: Primer and Reference Manual" by Gerard J. Holzmann [1]. In this report, we cover only certain features of Promela which are used in our model. Examples will be given along the way, taken from the actual Fiasco IPC model.

This section is organized as follows. In Section 3.1, we give a general comparison between Promela and programming languages. In Section 3.2, we explain about data types and data objects that we use in our Promela model. In Section 3.3, we explain about Promela processes. Then in Section 3.4, we give explanation about Promela meta terms and basic statements. Finally in Section 3.5, we cover the compound statements.

3.1 Promela vs Programming Languages

For people who have done some programming, especially in C language, Promela would seem familiar and learning about it would not be a difficult task. This is due to the fact that Promela adopts some syntaxes from C programming language. Those similar syntaxes include the following:

- boolean and arithmetic operators,
- assignment and equality, using a single equal and double equals,
- variable and parameter declarations,

- variable initialization and comments, and
- the use of curly braces to indicate the beginning and end of program blocks.

After learning the similarities, now we continue by mentioning some primary differences between Promela and C language, such as:

- semicolon in Promela is used as a statement separator, while in C, it is used as a statement terminator. So it will not be a syntax error in Promela as it is in C when the last statement of a sequence is not ended by a semicolon.
- Promela's main unit of execution is a process, while in C it is the `main` function.

From the above explanation, we get a brief overview of the similarities and differences between Promela and the C programming language. Now we try to compare Promela with common programming languages in broader sense. First, we will give features of most common programming languages which are ***not*** possessed by Promela:

1. functions that return values. Having no functions that return values leads Promela to have only two levels of scope. The first is global to the whole Promela model. The second level is local to a certain process.
2. expressions with side effects, which is allowed in C, for example:

   ```
   counter = x++;
   ```
3. pointers.

Features of Promela which can not be found in most programming languages are as follows:

1. the specification of nondeterministic control structures. Nondeterminism of Promela can arise as follows:
   - If there are multiple guard conditions which evaluate to true at the same time, SPIN will select one of these guards nondeterministically for execution.
   - If there is more than one process that has an executable statement and that could proceed at any point in an execution, then the semantics of Promela state that any of these processes may be selected for execution and the choice itself is nondeterministic.

   This nondeterminism will be clear as soon as we explain the whole Promela syntax and semantic in the next subsections.
2. primitives for process creation, and
3. a rich set of primitives for interprocess communication.

The basic building blocks of Promela models that we use are: structured data, asynchronous processes, and synchronizing statements. We will discuss in more detail each of these building blocks in the following subsections.

bit	0,1
bool	*false, true*
byte	0..255
pid	0..255
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$

Table 2: Basic Data Types

3.2 Data Types and Data Objects

Promela has some basic data types which are almost similar to C's data types. The range of values for each Promela's data type depends typically on the machine used. Table 2 summarizes the basic data types used in our Promela model and their typical value range on most machines.

Promela requires that all variables are declared before they can be used. We give below some examples of our model's variables declaration with the types from the basic data types above:

```
bool  have_sender;
short sender;
```

We can also declare one-dimensional arrays of variables in Promela as follows:

```
short rcv_partner[N];
short snd_partner[N];
```

The variable N which represents the number of threads in our model are used above to denote the size of the arrays. The definition of N as a macro is given as follows:

```
#define N 2
```

We can also define new data types of record structures. The following example introduces the structure `IPC_prm` and declares an array variable `ipc_g_prm` of this new structure:

```
typedef IPC_prm
{
   bool  has_receive_part;
   bool  open_wait;
   short partner;
   bool  has_send_part;
};

IPC_prm ipc_g_prm[N];
```

3.3 Processes

A process is the main unit of execution in Promela model. There should be at least one process declaration and one process instantiation for a model to be useful for modeling a distributed system. We have only one type of process in this master's thesis, the `thread` process type, which is given as follows:

```
active [N] proctype thread()
provided (ipc_lock_owner[_pid] == -1 || ipc_lock_owner[_pid] == _pid)
{
    ...
}
```

Process types are always declared globally in Promela. The keyword `proctype` followed by the identifier `thread` introduces a new type of process. So the `"proctype thread"` above defines the behavior of a process, not its execution. A `proctype` body may consists of zero or more data declarations and at least one statement.

The keyword `active` indicates that we want this process type to be instantiated. Following the keyword `active`, we have $[N]$ which is how we tell that we want N processes to be instantiated of the same process type `thread`.

In the example above, we also use `provided` clause following the empty parameter list of `proctype` declaration. A provided clause begins with the keyword `provided` and followed by a conditional statement in round bracket. It defines additional global constraints on process executions. In our model, the `provided` clause imposes that our processes of the process type `thread` cannot take any step unless the value of `ipc_lock_owner[_pid]` evaluates to -1 or to the process' `pid`. The absence of `provided` clause would be interpreted as the expression `true` which imposes no constraints on process executions.

Promela has other ways to instantiate a process, using the keyword `init` and `run`. But we will not use them here. More information about them can be found in [1].

Active processes can be differentiated from each other by the value of their process instantiation number, which is available in the predefined local variable `_pid`. The `_pid` is a non-negative integer starting from zero and assigned in order of process creation. Because Promela defines finite state systems, the number of processes is required to be bounded. SPIN limits the number of active processes to 255.

Since Promela supports asynchronous processes, a newly created process need not to start executing immediately after instantiation. Each process can interleave its executions with other processes in a nondeterministic way.

Promela processes can terminate and die. These two actions are different in the following sense:

- A process terminates when it reaches the end of its code. When a process terminates, it does not necessarily die. It indeed has no more statements to execute, but it will still be regarded as an active process in the system and the pid stays associated with this process.
- A process dies when it is removed from the system and its pid can be reused for another process.

3.4 Meta Terms and Basic Statements

In our Promela model, we use many Promela meta terms and basic statements. Meta terms that we use in our model include the following:

- `inline`,

- boolean values: `false` and `true`,
- comments (starts with a `/*` and ends with a `*/`),
- skip (performs no operation),
- macros, and
- LTL (Linear Temporal Logic),

Interesting meta terms to discuss are inline and LTL. We will discuss about `inline` in Subsection 3.4.1. LTL will be part of verification phase which is discussed in Section 5.

For basic statements, we use `assert`, `printf`, and conditional statement. All these basic statements are the same as what we would find in C programming language. More information about `assert` statement will be given in Section 5, when we speak about correctness property.

3.4.1 Inline

A Promela `inline` is very similar to C-style macro definition. It is very useful for structuring and organizing a Promela model. At an inline's point of invocation in the Promela code, the SPIN parser performs the following:

- a textual substitution of the body of the `inline` definition,
- a direct textual substitution of all actual parameter names (that are provided at the point of invocation) for the formal names that are used in the definition.

Formal parameters of an `inline` definition have no type specification. They are just place holders for the actual variable names that are inserted when the `inline` is invoked. The body of an `inline` definition can contain declarations for local variables, but they will be included in the text segment that is inserted at each point of invocation. Therefore, their scope depends on the point of invocation of the `inline`.

The general rules are that an `inline` definition must appear before its first use, and must always be defined globally. It may itself contain other `inline` calls, but it may not call itself recursively. Below is an example of an inline definition `do_send` from our model with three parameters: `ds_pid`, `ds_partner`, and `ds_ret`:

```
inline do_send(ds_pid, ds_partner, ds_ret)
{
    ...
}
```

3.5 Compound Statements

Compound statement is a construct which comprises of other smaller statements as constituents. There are five types of compound statements in Promela, viz. atomic sequences, deterministic steps, selections, repetitions and escape sequence. In the following subsections, we only discuss compound statements which we will use in our Promela model.

3.5.1 Atomic Sequences

Here is an example of atomic sequence:

```
atomic {
   assert(ipc_lock_owner[iu_rec_pid] == iu_snd_pid);
   ipc_lock_owner[iu_rec_pid] = -1
}
```

Atomic sequence is the simplest compound statement which is uninterruptable. This means that all statements inside an atomic sequence will be executed as one indivisible unit, non-interleaved with other processes. All steps in the sequence will complete before any other process is given the chance to execute.

If any statement in atomic sequence is unexecutable, then the atomic chain is broken and another process can take over control. When the blocking statement becomes executable later, control can nondeterministically return to the process and the atomic execution resumes as if it had not been interrupted.

3.5.2 Selection Construct

A Promela selection construct starts with the keyword `if` and ends with `fi`. The selection body should contain one or more option sequences, with every single option sequences starts with a double colon `::`. The first statement after the double colon `::` in each sequence is called the guard. The following is an example of an `if` selection construct taken from our model:

```
if
:: (so_snd_pid == rcv_partner[so_rec_pid]) ->
       so_ret = true
:: else ->
       so_ret = false
fi;
```

The above selection structure contains two option sequences. The first sequence's guard is (`so_snd_pid == rcv_partner[so_rec_pid]`). The second is `else` which represents the negation of the first guard. Removing the `else` option would make execution blocks until the first guard evaluates to true. This facilitates modeling interprocess synchronization.

Only one sequence from the list will be executed. A sequence can be selected only if its guard statement is executable. In the example, the two guards are mutually exclusive, but this is not required. If more than one guard statement are executable, then one of them will be selected nondeterministically. When none of the guards is true, executions will block and it is not regarded as an error. These two characteristics make Promela different from other programming languages. They are very useful to synchronize processes in Promela models.

Another difference between Promela and other programming languages is that Promela does not restrict the type of statements that can be used as a guard. It may be an assignment, `printf`, `skip`, etc. Below is another example of `if` construct which at first might seem a little bit confusing:

```
if
:: ipc_g_prm[_pid].has_receive_part = false
:: ipc_g_prm[_pid].has_receive_part = true
```

The above `if` construct contains two option sequences, where each sequence has an assignment as the guard. Because assignments are always executable, here Promela introduces its nondeterminism in choosing which option sequence to execute.

Another form of Promela selection construct can also be written in the following way:

```
(ipc_lock_owner[irr_snd_pid] == -1);
ipc_lock_owner[irr_snd_pid] = irr_rec_pid;
```

This example shows that Promela treats a condition as a full statement. In the above example, we use semicolon after the condition (`ipc_lock_owner[irr_snd_pid] == -1`). Actually we could also write the above construct with an arrow after the condition as the following:

```
(ipc_lock_owner[irr_snd_pid] == -1) ->
    ipc_lock_owner[irr_snd_pid] = irr_rec_pid;
```

The above two examples are equivalent because arrows and semicolons are equivalent in Promela. In general, usually only a condition is followed by an arrow as a statement separator. The reason is because a condition has the possibility to block the execution when it does not evaluate to true. All other statements are followed by a semicolon. But this preference between arrow and semicolon is not imposed by Promela grammar. Further analysis will lead us to the following busy wait cycle which is the long form of the above two examples:

```
wait_unlocked:
   if
   :: ipc_lock_owner[irr_snd_pid] == -1 ->
         ipc_lock_owner[irr_snd_pid] = irr_rec_pid
   :: else -> goto wait_unlocked
   fi;
   ...
```

3.5.3 Repetition Construct

A Promela repetition construct starts with the keyword `do` and ends with `od`. The repetition body should contain one or more option sequences. It is almost similar to Promela selection construct. The only difference between a selection and a repetition construct is that a repetition is automatically repeated from the start when the execution of an option completes. Whereas for a selection construct, execution moves on to the next statement. The execution of a repetition can be broken by either transfering control explicitly by a `goto` statement, or by executing a `break` statement. Here is an example of repetition construct:

```
do
:: ((sender_ls[ds_partner].snd_ls[i] == ds_pid) &&
    (sender_ls[ds_partner].last_index != 0)) ->
```

```
        in_snd_ls = true;
        break
:: ((sender_ls[ds_partner].snd_ls[i] != ds_pid) &&
    (i < (sender_ls[ds_partner].last_index - 1)))   ->
        i++
:: else ->
        break
od;
```

4 Modeling Design

This section reports the design process of the Fiasco IPC model. As we have mentioned before, the original Fiasco source code is written in C++. The IPC system call is performed by the `sys_ipc` method of `Class Thread`. There are a lot of abstractions which have to be made to convert from the object-oriented paradigm of the Fiasco C++ code into Promela formalism.

This Section is organized as follows. In Section 4.1, we will explain which conventions that we follow in modeling the Fiasco IPC. In Section 4.2 and 4.3, we explain how we translate the threads and variables of IPC, respectively, into Promela. In the last part, Section 4.4, we describe the `inlines` in general and we give an example of how we translate a C++ function into a Promela `inline`.

4.1 General Modeling Conventions

The IPC system call is quite complex with many additional features which are outside the scope of our model, such as long IPC, next-period IPC, page-fault IPC, or interrupt IPC. In this master's thesis, we only model short IPC with the following features:

- thread lock,
- timeout is always set to zero,
- data copying is represented by a boolean value which indicates whether the data has been copied or not. We do not model the actual message, since we are interested more in the executions of IPC rather than the result of a message delivery through IPC.

Even with such a simple modeling design, we come up with quite a complicated Promela model due to the delicate executions of IPC, with a lot of inline calls. The primary guidance rules in the modeling design are as follows:

- there is only one process type, that is the thread,
- the scope of the threads' IPC parameters and IPC states variables are global,
- the model performs the following nondeterministic choice of IPC parameters:
  - having receive part or not,
  - having send part or not,

  - whether or not the receive operation is an open receive,
  - which thread is to be the partner.

- thread runs continuously in a loop for doing IPC,

We assign integer values as error code to the following specific error messages:

- Transient Error = -1
- Send Error / Enotexistent = -2 (Receiver does not exist)
- Retimeout = -3 (Receive timeout)
- Setimeout = -4 (Send timeout)
- Recanceled = -5 (Receive canceled)
- Secanceled = -6 (Send canceled)
- Reaborted = -7 (Receive aborted)

4.2 Thread in Promela

Fiasco's threads are modeled as Promela's `proctype` construct. In our model, the only type of process is thread. We build our model to be able to have variable number of threads. To do that, we use a constant N to represent the number of threads. The definition of N as a macro has been given in Section 3.2. The complete definition of thread is given in the Appendix, with the specific location: "loc 90"

The process type thread performs an infinite loop, where in each loop, it is assigned to do an IPC with a nondeterministic choice of IPC parameters. Possibilities of thread id are as follows:

- nil thread: -2,
- invalid thread id or null pointer: -1,
- existing thread: $0, \ldots, N-1$,
- non-existing thread: $> N$

4.3 Global Variables

In this subsection, we will elaborate on the data objects used in the model. As we have explained in Section 3, there are only two levels of scope in Promela models, viz. global and process local scope. The scope of a variable is global if it is declared outside all process declarations. It is local if it is declared inside a process declaration.

The reason why we assign some variables to be in global scope is because IPC requires the sender and the receiver to access each other's state. This can be done easily using global variables.Another second important reason is because later on in the verification phase, we will need to specify properties in `never claims` construct which could only work based on global variables.

In the following, we will explain about the global variables used in our model. We start by giving the definition of data types `IPC_Prm` and `IPC_Output` as follows:

```
typedef IPC_Prm
{
   bool  has_receive_part;
   bool  open_wait;
   short partner;
   bool  has_send_part;
};

typedef IPC_Output
{
   short error = -10;
   short dope = -10;
   bool msg_copied;
   short rcv_source = -1;
}
```

The above two data types are intended to save the value of the IPC parameters and the IPC output for each thread. We derive these two data types as an abstraction from the registers described in [5]. Then we declare the array variable `ipc_g_prm` and `ipc_output` to be of type `IPC_Prm` and `IPC_Output`, respectively.

```
IPC_Prm ipc_g_prm[N];
IPC_Output ipc_output[N];
```

Below we give another data type definition, `IPC_State`, and a declaration of array variable `ipc_state` of this new type `IPC_State`. We model IPC states that we described in Section 2 by defining the data type `IPC_State` as follows:

```
typedef IPC_State
{
   bit ready;
   bit receiving;
   bit polling;
   bit ipc_in_progress;
   bit send_in_progress;
   bit busy;
   bit cancel;
   bit polling_long;
   bit busy_long;
   bit rcvlong_in_progress;
};

IPC_State ipc_state[N];
```

For the receiver's sender list, we introduce a new data type `Sender_List`. It consists of two fields: `snd_ls[N]` which is an array of type `short`, and `last_index` which points to the next empty index of the array `snd_ls`. The definition of `Sender_List` and a declaration of array variable `sender_ls` of this type are given below:

```
typedef Sender_List
{
   short snd_ls[N];
   byte last_index;
};

Sender_List sender_ls[N];
```

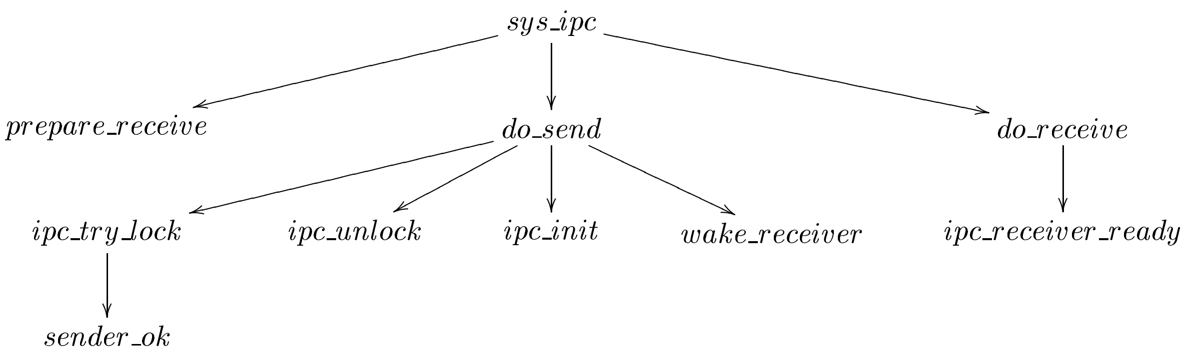
In the following, we give declarations of three arrays of type `short`: `ipc_lock_owner`, `rcv_partner`, and `snd_partner`:

```
short ipc_lock_owner[N] = -1;
short rcv_partner[N];
short snd_partner[N];
```

As the variable name describes, `ipc_lock_owner[x]` saves the process `pid` which locks process with `pid x`. We initialize `ipc_lock_owner` with -1 since this variable is used in the provided clause of process type `thread`. The executions of process `thread` depend on the value of this variable. Meanwhile, the array variables `rcv_partner` and `snd_partner` save the value of thread's partner when it acts as a receiver and as a sender, respectively.

4.4 The Inlines

Now we give the structure of the Promela implementation of IPC system call's functions. These functions are modeled as inlines in Promela. The connection between the main function `sys_ipc` and the auxiliary functions is depicted in the picture below.



We translate all the above auxiliary functions into Promela inlines. As an example, below we give the C++ function `ipc_try-lock` and further below we give the translation result, that is the Promela inline `ipc_try_lock` (note that the comment "loc" followed by a number is intended to match statement in C++ with statement in Promela):

```
inline int Receiver::ipc_try_lock(const Sender* sender)  // loc 220
{
   if (EXPECT_FALSE (state() == Thread_invalid))         // loc 221
      return Ipc_err::Enot_existent;                     // loc 222

   thread_lock()->lock();                                // loc 223
```

```
   if (EXPECT_FALSE (!sender_ok (sender)))           // loc 224
   {
      thread_lock()->clear();                        // loc 225
      return -1;                                     // loc 226
   }

   return 0;                                     // OK, loc 227
}


inline ipc_try_lock (itl_snd_pid, itl_rec_pid, itl_ret)  /* loc 220 */
{
   bool sok;

   printf("ipc_try_lock(snd:%d, rec:%d, ret:%d)\n",
          itl_snd_pid, itl_rec_pid, itl_ret);
   if
   :: (itl_rec_pid >= N) ->                            /* loc 221 */
         itl_ret = -2                                  /* loc 222 */
   :: else ->
         atomic {
            assert(ipc_lock_owner[itl_rec_pid] != itl_snd_pid);
            (ipc_lock_owner[itl_rec_pid] == -1);
            ipc_lock_owner[itl_rec_pid] = itl_snd_pid;  /* loc 223 */
         }

         sender_ok(itl_snd_pid, itl_rec_pid, sok);      /* loc 224 */
         printf("sender_ok returned %d\n", sok);

         if
         :: sok ->                                     /* loc 224 */
               itl_ret =  0                            /* loc 227 */
         :: else ->
               atomic {                                /* loc 225 */
                  assert(ipc_lock_owner[itl_rec_pid] == itl_snd_pid);
                  ipc_lock_owner[itl_rec_pid] = -1;
               }
               itl_ret = -1                            /* loc 226 */
         fi
   fi;
}
```

The C++ function `ipc_try_lock` is designed to run in the context of the `Receiver` object and it has one parameter which is the `Sender` object. Since Promela is a process-based language, these object-oriented paradigm cannot be matched. As a solution, we translated the `Sender` object into sender thread's pid: `itl_snd_pid` and add one additional parameter for the value of receiver thread's pid: itl_rec_pid which represents the context of object Receiver in the original code. For the integer return values in C++' ipc_try_lock, Promela inline does not have this feature either. Again, the solution was to add another additional parameter for the return value, viz. itl_ret.

Now we base the explanation on the Promela inline. For every statement with specific comment "loc" in Promela, the corresponding original C++ code can be traced by the same comment. The Promela inline starts by declaring a variable sok of type bool. Then it prints some information using printf which resembles C syntax. On loc 221, it checks the receiver pid's validity. If the receiver is not valid, then ipc_try_lock will set the value of parameter itl_ret into -2 which codes the error Enot_existent (receiver does not exist). If the receiver is valid, it is locked on loc 223. The locking action is done in an atomic sequence, which is preceded by two other statements. The first statement asserts that the receiver has not yet been locked by the sender. The second is a conditional statement which requires the process to wait until the receiver is not locked by any process, which is coded by -1.

After the receiver is locked, ipc_try_lock calls the inline sender_ok(itl_snd_pid, itl_rec_pid, sok) and check the return value in the parameter sok. Both actions are marked by the comment loc 224. If the return value sok is true, itl_ret will be set to 0 on loc 227. Otherwise, the receiver's lock will be cleared on loc 225. This is done in atomic sequence consisting of two statements. The first one is a conditional statement which requires that the receiver's lock owner is the sender. The second is the action of clearing the lock by setting the receiver's lock owner to -1. Later on loc 226, ipc_try_lock will return -1 which is saved in parameter itl_ret.

Below is a brief structure of the inline ipc_snd_regs which calls the inline ipc_try_lock and checks ipc_try_lock's return value by examining the third parameter isr_ret.

```
inline ipc_snd_regs (isr_snd_pid, isr_rec_pid, isr_ret)
{
    ...
    ipc_try_lock (isr_snd_pid, isr_rec_pid, isr_ret);
    ...

    if
    :: (isr_ret != 0)
    :: else ->
            ...
    fi;
}
```

5 SPIN Simulation and Verification

In this Section, we will explain about the simulation and verification of the model using SPIN. We have explained about how our model was built in the previous two sections. After the model is free from any syntax or typographical error, we do some simulations to get an idea of how the model really works, and then continue with the verification phase. When errors were found during verification, we sometimes switched back to simulation to trace the error. This shows how close is the role of simulation and verification in SPIN.

In essence, the two modes of SPIN can be described as follows:

- simulation mode: can give a quick impression of the types of the behaviour that are captured by system model. Simulation itself cannot prove the facts we are interested in.

- verification mode: can prove the facts. When the verifier finds a counterexample to a correctness claim, it relies on SPIN simulation mode to display the error trace using guided simulation.

Logical verification focuses on determining whether design requirements could possibly be violated, not on how likely/unlikely such violations might be. This focus will lead us to achieve system verification where we can state that there is no possible violation of a given requirement.

The verification was done in a series of steps repetitively with increasingly detailed models. Interesting properties to verify are: the absence of deadlock or livelock, the absence of fail condition, or some liveness properties. In SPIN, verification of safety and liveness properties must be done separately. Specifically, safety and liveness properties are classified as follows:

1. safety properties (which is SPIN default): assertion violations, deadlocks,

2. liveness properties: the absence of acceptance cycles.

We will explain briefly about acceptance cycles in Section 5.2.

This section is organized as follows. First, in Section 5.1, we explain briefly about simulation mode in SPIN. Then in Section 5.2, we give a general overview of various correctness property in SPIN. In Section 5.3, 5.4, and 5.5, we give explanation on how to generate, to compile and to run the verifier, respectively. Finally, in Section 5.6, we describe our efforts in veriying our model using the knowledge that we have explained in the previous four subsections.

5.1 Simulation

SPIN offers three kinds of simulation as follows:

- random simulation,

- interactive simulation, and

- guided simulation.

In the beginning of our modeling phase, we used mostly the random simulation to get an insight about the behaviour of the model. After all small modeling errors have been fixed, then we continue with the verification phase.

The beginning of verification phase reports many kinds of errors which we do not find in the simulation. When we find errors, we use guided simulation to trace the errors. SPIN's guided simulation uses trail files which are generated only in verification mode when the verifier finds a counterexample. The SPIN's graphical user interface, XSPIN, helps us conveniently debug the errors. More information about SPIN's simulation mode and XSPIN can be found in [1].

5.2 Correctness Property

From the verifier point of view, there are two types of correctness properties:

1. states properties: claims about reachable and unreachable states,
2. path properties: claims about feasible or infeasible executions, i.e sequence of states.

SPIN can check the above two claims, which are expressed in Promela. There are some basic properties which SPIN checks by default, such as, the absence of system deadlock states. Promela constructs that we use in our model to formalize correctness property are as follows:

1. Basic Assertion
   Basic assertions in Promela are statements of the form

   ```
   assert(expression).
   ```

   The characteristic of Promela basic assertion is that it is always executable, but has no effect as long as the expression specified evaluates to *true* or to a nonzero value. It has an implied correctness property that it is never possible for the expression to evaluate to *false* or zero. A failing assertion will trigger an error message. Basic assertion is the only type of correctness property in Promela that can be checked during simulation runs with SPIN.

2. Accept States

   An *accept-state* label is any label starting with the string `"accept"`. The implicit correctness claim that is expressed by accept-state label is that there should not exist any execution that can pass through an accept-state label infinitely often.

   It is important to note that we do not explicitly use accept -state labels in our model. They are only used in some `never claims` which are generated from LTL formulae. We explain about `never claims` below.

3. Never Claims

   Never claim is used to specify behavior that should ***never*** happen in the system executions. It is the only Promela construct which is capable of checking system properties just before and just after each statement execution. This checking is always performed no matter which process executes a statement.

   We can write our own never claims by hand or they can be generated mechanically from LTL formulae by using SPIN command-line option `-f`. Almost all Promela language constructs can be used inside a never claim, except the ones that have side effect, such as assignments. The limitation is that never claims can only access global variables of a Promela model.

   Never claims can be used to match either finite or infinite behavior. A finite behavior is matched if the claim can reach its final state, that is the last part of the claim (the closing curly brace). An infite behavior is matched if the claim contains acceptance cycle.

A simple example of never claim is the one which is used to check system invariance, which is a safety property. For instance, if `p` is a system invariant property, then the never claim can be written by checking the opposite behavior of the system, that is `!p`, as follows:

```
never {
   do
   :: !p -> break
   :: else
   od
}
```

As soon as `p` evaluates to false, then the never claim breaks from its loop and terminates, which indicates that error behavior has occured. On the other hand, if `p` remains true, then the never claim stays in its initial state and this is the behavior that we have expected.

We can include a never claim as part of the model. The other choice is to save it in a separate file. The latter case is normally chosen when there are various claims to be verified for the same model. For the latter case, we use the additional option `-N` followed by the name of the never claim file when generating the verifier, as follows:

```
> spin -a -N never_claim_file model_file
```

Never claims that we explain above are quite expressive and powerful. But it can be hard to be produce never claims which specify complex temporal properties of a distributed system. To answer this problem, SPIN allows us to specify properties in LTL formulae and it can translate those LTL formulae into never claims. More information about LTL formulae are given below.

4. LTL

SPIN has a separate parser which converts LTL formulae into Promela never claims. To invoke this converter, we use the command:

```
> spin -f LTL_formula
```

The argument `LTL_formula` given to the command above should be started and ended with a quote. The LTL formulae which can be accepted by SPIN should only consists of:

- propositional symbols (including `true` and `false`),
- unary temporal temporal operators: `[]` ("always" or "box") and `<>` ("eventually" or "diamond"),
- binary temporal operator: `U` ("until" - strong version),
- three logical operators: `!` (negation), `&&` (and), `||` (or), `->` (logical implication).

The propositional symbols which represent state properties are always written with lower-case symbols, for instance, `p` or `q`. It is important to remember that LTL formulae to be translated by SPIN cannot contain arithmetic and relational operators.

An example of more complex temporal property than just an invariance property is:

> *Every system state in which p is true eventually leads to a system state in which q is true*

This property can be captured by LTL formulae:

```
[](p -> <> q)
```

To show that the above property should not be violated, we must give the negated version of the formula above to the SPIN converter and the never claim is generated, as follows:

```
> spin -f '! [] ( p -> <> q)'
never {    /* ! [] ( p -> <> q) */
T0_init:
        if
        :: (! ((q)) && (p)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (! ((q))) -> goto accept_S4
        fi;
}
```

5.3 Generating The Verifier

The most important feature of SPIN is probably that it can generate optimized verifiers from a user-defined Promela model [1]. Assuming that our model is in a file called `sys_ipc.pml`, we can generate the source code for our model specific verifier by typing:

```
spin -a sys_ipc.pml
```

The verifier is generated as a C program that is stored in a number of files. Those files have a fixed set of names, starting with a three-letter prefix `pan`. They include the following [1]:

- pan.h
  is the generic header file for the verifier that contains, for instance, the translated declarations of all global variables and all process types.
- pan.m
  defines the executability rules for all Promela statements used in the model and their effect on the system state after successfully executed.
- pan.b
  defines how the effect of each statement from pan.m can be undone when the direction of the depth-first search is reversed.

- pan.t
  contains the transition matrix that encodes the labeled transition system for each process type.

- pan.c
  contains the computation algorithms, the state space maintenance and cycle detection algorithms, encoding optimized versions of either a depth-first or a breadth-first search.

The source code can further be compiled using various compile-time options to produce an optimized executable verifier. Next, the verifier is run, possibly with some run-time options to get the best verification run. Both compilation and run of the verifier are explained in the following two subsections.

5.4 Compiling The Verifier

To compile the verifier for a straight exhaustive verification, without using any compile-time option, we can just compile the file `pan.c` using the command:

```
gcc -o pan pan.c
```

The file `pan.c` includes all other files that are generated by SPIN. So, it is enough to give only this file to the compiler. The result of the compilation command above is an executable file called `pan`. From this point until the end of Section 5, we assume that we always compile `pan.c` into an executable file called `pan`.

The use of compile-time options is not a must. However, in our verification, we would need to use some specific compile-time options for the sake of efficiency. In fact, those options would turn out to be a great help for running the verification later. In principle, compile-time options help us to form the behavior of the verifier. We explain some compile-time options [1] that are useful in our verification as follows:

1. Basic options:

   - `-DBFS`
     tells the verifier to use a breadth-first search algorithm rather than the default depth-first search. This option is only usable for the search of safety properties violations. It runs slower than depth-first search and can consume more memory. But if memory is not a hindrance, it can help to find the shortest error path. This option can be combined with other options for reducing memory, except the option `-DSC`.
   - `-DMEMLIM=N`
     limits the maximum amount of memory that can be used by the verifier to N Megabytes. This option helps to avoid any paging behavior.
   - `-DNP`
     enables non-progress cycle detection, which later on, enables run-time option `-l` and simultaneously disables run-time option `-a` for acceptance cycle detection.

2. Options to increase speed:

- `-DSAFETY`
  optimizes the code for detection of safety properties violations. This is done when no cycle detection is needed. At run-time, this option disables the options `-l` and `-a`.
- `-DNOFAIR`
  disables the code for weak fairness algorithm. At run-time, this will disable the option `-f`.

3. Options to decrease memory use:

- `-DBITSTATE`
  Uses the bitstate storage algorithm runs to get a better impression of the complexity of the problem that is being tackled. This command does not guarantee exhaustive coverage.
- `-DHC`
  Compiles with hash-compact option which uses less memory at a small risk of incompleteness of the search.
- `-DCOLLAPSE`
  compiles with the memory collapse option. This preserves an exhaustive verification but uses less memory.
- `-DMA=N`
  Enables the minimized automaton storage method to encode state descriptors. This method can reduce a great amount of memory use, but the trade-off is that it requires quite long run-time. We use this option many times in our verification as we will explain in the next subsection. Further information about this method can be found in [1] page 202.
- `-DSC`
  is meant only for verification with very large depth-limit. It uses a stack-cycling method. The memory requirements for the stack increase linearly with its maximum depth. This stack-cycling allows only a small part of the stack to reside in memory, with the rest stays on disk. Compiling with this option will make the run-time option `-m` determine only the size of the in-core portion of the stack, but does not restrict the stack's maximum size.

5.5 Running The Verifier

There are some additional options that we can choose in running the verifier to improve its performance. For example, it might be helpful to do the following:

- provide an estimation about the number of reachable states,
- provide the maximum search depth of a non cyclic execution path,
- tell the verifier to search for violations of safety or liveness properties, since the two searches cannot be combined. The default setting is to search for violations of safety properties.

5.5.1 Setting the Number of Reachable States

There are two different search modes which have different effects on the setting of the number of reachable states. The two search modes are explained below:

1. Exhaustive search mode:

   Exhaustive search mode is performed when the verifier is compiled without any memory compression options. In this mode, the verifier stores all reachable states in a hash table, with a default size of 2^{18} slots. Logically, this state storage method would work effectively if the table as at least as many slots as there are reachable states that will have to be stored in it. If the table has too many slots, memory will be wasted. On the contrary when the table has too few slots, CPU cycles will be wasted. To change from the default size of the hash table to 2^{25} slots, we can do the following command:
   `./pan -w25`,

2. Bitstate mode:

   In bitstate mode, each reachable state is stored only using one bit of memory. Given a state, a hash function is used to compute the address of the bit in the hash table. The default size of the hash array is 2^{22} bits. To change the size of the hash array to 2^{29} bits, we do as follows:
   `./pan -w29`
   The optimal value to be used depends mainly on the amount of physical memory that is available to run the verification. A bitstate run with too small size of hash array will get less coverage, but will run faster.

5.5.2 Setting the Search Depth

The default search depth restriction of SPIN verifier is 10,000 steps. If this isn't enough, the search will truncate at 9,999 steps. We can define a different search depth of N steps by the following command:

```
./pan -mN
```

A deeper search depth requires more memory for the search, and this memory cannot be used to store reachable states. In the case that there is not enough memory to allocate a search stack for very deep searches, an alternative is to use SPIN's stack cycling algorithm that arranges for the verifier to swap parts of the search stack to disk during a verification run, retaining only a small portion in memory. Such a search can be set up and executed as follows:

```
spin -a filename
cc -DSC -o pan pan.c
./pan -m100000
```

In the above setting, the value specified with the `-m` option defines the size of the search stack that will reside in memory. There is no preset maximum search depth in this mode: the search can go arbitrarily deep, or at least it will proceed until the diskspace that is available to store the temporary stack files is exhausted.

5.5.3 Other Run-Time Options

Here we give description about other run-time options that we use in running our verifier:

- `-A`
  Suppresses the reporting of basic assertion violations. We use this when we want to get a different kind of errors, such as non-progress or acceptance cycles.

- `-a`
  Enables detection of acceptance cycles. This option is disabled when the verifier is compiled with the directive `-DNP`.

- `-E`
  Suppresses the reporting of invalid end-state violations.

- `-n`
  Disables the default reporting of all unreached states at the end of a verification run.

5.6 Various Verification Attempts

The main disadvantage with model checking is the state explosion problem. The Fiasco IPC model also faces this problem, which blows up the memory use during verification run. Due to this fact, at first, we do the verification by setting many cases individually for each verification attempts. During these verifications, we found errors that lead us to improve the model. But memory is still an issue.

In order to keep the nondeterminism of the model, but at the same time reducing memory, we also try to create an auxiliary program written in C, which sets up every possible combination and then run verification for each combination. During this phase, we still meet some problems with the memory use.

After experimenting with many kinds of options for compile-time and run-time, we found two helpful options to reduce the memory, viz. `-DSC` and `-DMA`. With these options, we try to set the IPC parameters nondeterministically from SPIN. During this phase, we found many interesting counterexamples which help us to further refine our models.

At this phase, memory is not a problem thanks, to the option `-DMA`. But still we are concerned about the huge search depth and the stack file produced by the option `-DSC`. After some consultation on the experts, we try to do the following improvements:

- use the new version GCC, that is gcc-3.4,

- use compile-time option `-O3` for compiler optimization,

- added compile time option `-march=opteron` to improve compilation. This option generates instructions for machine specific cpu-type that we use, that is AMD Opteron.

- try running the command:

  ```
  spin -A filename
  ```

  This gives some hints about things that may be redundant for the given properties.

- reexamining the data types, to make sure that we could use smaller domain. For example, changing `int` to `byte` or `short`.
- break the problem up into smaller pieces
- make stronger abstractions
- remove redundancy
- reduce the number of processes

5.6.1 SPIN Version and Hardware Used

All the simulation and verification reported in this master's thesis use a patched SPIN `Version 4.2.3` which was released on February 5, 2005. The original SPIN always produces error when producing a big stack file. That is why we use the patched SPIN. The patched SPIN was delivered by Hendrik Tews and it corrects the executions of SPIN when managing big stack files which is bigger than 2GB.

The operating system used is Linux 2.6 with a 32-bit system. The machine on which we perform all the simulation and verification totally has 2048 MB RAM and it has two processors, each with the following description:

- vendor_id : AuthenticAMD
- model name : AMD Opteron(tm) Processor 248
- cpu MHz : 2192.904
- cache size : 1024 KB

5.6.2 Default Verification

SPIN default verification is to check for violations of safety properties. The safety properties which are checked by SPIN are as follows:

- assertion violations:
- unreachable code,
- unintended end states:

We always use the compile-time option `-DSAFETY` for this default verification. We report the verification result of the model for two and three threads as follows:

1. two threads:
   The following commands depict the generation, compilation, and run of the verifier:

   ```
   > spin -a sys_ipc.pml
   > gcc-3.4 -O3 -o pan -DSAFETY -DMEMLIM=1500 -DSC -DMA=300
             -march=opteron pan.c
   > (date; time ./pan -m1000; date) >& log_ver_2_thread &
   ```

   From the log file `log_ver_2_thread`, we know that the verification succeeds, with some information as follows:

- verification time (user and system time): 14 hours 12 minutes,
- total memory used: 1,383.323 MB

2. three threads:
Due to lack of memory, we have to use the compile-time option `-DBITSTATE` and the run-time option `-w27` for the verification of the model with three threads. This time the model is saved in another file `sys_ipc_3_thread`. The commands are:

```
> spin -a sys_ipc_3_thread.pml
> gcc-3.4 -O3 -o pan -DSAFETY -DMEMLIM=1800 -DBITSTATE -DSC
          -march=opteron pan.c
> (date; time ./pan -m1000 -w27; date) >& log_ver_3_thread &
```

The verification succeeds, with some information as follows:

- verification time (user and system time): 2 hours 4 minutes,
- total memory used: 1,611.043 MB,
- hash factor: 2.36853

The run-time reduces greatly, but this bitstate method does not guarantee an exhaustive coverage. Especially since the hash factor (the maximum number of states divided by the actual number of states) is only 2.36853, which is too small. The reliable minimum hash factor is 100, which means a coverage of 100%.

5.6.3 Verification of Proposed Properties

In this subsection, we specify some properties of Fiasco IPC using never claims. Some claims are hand-written because it simply specified invariant property. Some others are generated from LTL formulae by SPIN. All these properties are proposed for the model with two thread processes instantiated. This is done by setting the number `N` to two.

1. The state flags `"busy"` and `"send_in_progress"` are never active at the same time for the same thread.
This invariant property is saved in a file called `busy_and_send_in_progress.ltl` below:

```
#define p ((ipc_state[0].busy == 1) &&
           (ipc_state[0].send_in_progress == 1))
never {
   do
   :: p -> break
   :: else
   od
}
```

The following commands depict the generation, compilation, and run of the verifier for the above property:

```
> spin -a -N busy_and_send_in_progress.ltl sys_ipc.pml
> gcc-3.4 -O3 -o pan -DMEMLIM=1400 -DSC -DMA=150
          -march=opteron pan.c
> (date; time ./pan -m5000; date) >& log_ver_busy_sip &
```

From the log file `log_ver_busy_sip`, we know that the verification succeeds, with some information as follows:

- verification time (user and system time): 8 hours 15 minutes,
- total memory used: 1,458.778 MB,
- we could add the compile-time option `-DSAFETY` to get more efficient verification

2. The state flag `"polling"` is only ever active when `"send_in_progress"` is also active.
   This is another invariant property as the previous one we have. It is saved in a file called `polling_and_sip.ltl` as follows:

```
#define p ((ipc_state[0].polling == 1) &&
              (ipc_state[0].send_in_progress == 0))
never {
   do
   :: p -> break
   :: else
   od
}
```

Now we use the compile-time option `-DSAFETY` to do the verification as follows:

```
> spin -a -N polling_and_sip.ltl sys_ipc.pml
> gcc-3.4 -O3 -o pan -DSAFETY -DMEMLIM=1500 -DSC -DMA=100
          -march=opteron pan.c
> (date; time ./pan -m1000; date) >& log_ver_polling_sip &
```

The verification succeeds and the log file `log_ver_polling_sip` gives the following:

- verification time (user and system time): 6 hours 18 minutes
- total memory used: 1,458.484 MB

3. The state flag `"busy"` is only ever active when `"receiving"` is also active. This invariant property is saved in a file called `busy_receiving.ltl`:

```
#define p ((ipc_state[0].busy == 1) &&
              (ipc_state[0].receiving == 0))
#define q ((ipc_state[1].busy == 1) &&
              (ipc_state[1].receiving == 0))
never {
   do
   :: (p || q) -> break
   :: else
   od
}
```

The verification is done as follows:

```
> spin -a -N busy_receiving.ltl sys_ipc.pml
> gcc-3.4 -O3 -o pan -DSAFETY -DMEMLIM=1500 -DSC
          -DMA=100 -march=opteron pan.c
> (date; time ./pan -m1000; date) >& log_ver_busy_receiving &
```

The verification succeeds and the log file `log_ver_busy_receiving` gives the following:

- verification time (user and system time): 6 hours 16 minutes
- memory: 1,458.484 MB

4. The value of each element of array `snd_partner` must always be greater than or equal zero.
   This is also another invariant property which is specified as follows:

```
#define p (snd_partner[0] < 0)
#define q (snd_partner[1] < 0)
never {
   do
   :: (p || q) -> break
   :: else
   od
}
```

   The options used for the generation, compilation and run of the verifier are the same as we used in the second verification. The verification succeeds with following result:

   - verification time (user and system time): 6 hours 9 minutes
   - memory: 1,458.727 MB

5. The value of each element of `rcv_partner` must always be greater than or equal -1.
   This is also another invariant property which is specified as follows:

```
#define p (rcv_partner[0] < -1)
#define q (rcv_partner[1] < -1)
never {
   do
   :: (p || q) -> break
   :: else
   od
}
```

   The options used for the generation, compilation and run of the verifier are the same as we used in the second verification. The verification succeeds with following result:

   - verification time (user and system time): 6 hours 19 minutes
   - memory:1,458.484 MB

6. Whenever a thread has a receive part and its partner exists, then eventually in the executions of the same IPC loop, its receiving flag will be set.
   This property requires us to add some code in our model. We need two global array variables and one global variables: `flag_receiving[N]`, `flag_has_rec_part[N]`, and `counter` of type byte. The LTL formulae for this property is given as follows:

```
#define p ((ipc_g_prm[0].partner != -2) &&
           (ipc_g_prm[0].has_receive_part == 1) &&
           (ipc_g_prm[0].partner < N))
#define q ((ipc_state[0].receiving == 1) &&
           (flag_has_rec_part[0] == flag_receiving[0]))
[]  ( p -> <> q)
```

   The generated never claim is the following:

```
never {    /* !([]  ( p -> <> q)) */
T0_init:
    if
    :: (! ((q)) && (p)) -> goto accept_S4
    :: (1) -> goto T0_init
    fi;
accept_S4:
    if
    :: (! ((q))) -> goto accept_S4
    fi;
}
```

   The options used for the generation, compilation and run of the verifier are now different, because now we want to verify something more than just an invariant property. We remove the compile-time option `-DSAFETY` and we add run-time option `-a`. The commands are as follows:

```
> spin -a -N receive.ltl sys_ipc_receive.pml
> gcc-3.4 -O3 -o pan -DMEMLIM=1500 -DSC -DMA=150
          -march=opteron pan.c
> (date; time ./pan -m1000 -a; date) > & log_ver_receive &
```

   The verification of this property suffers from a huge stack files. We leave it open for further investigation of the model and the LTL formulae.

6 Conclusions and Future Work

6.1 Conclusions

In this master's thesis, we have developed a Promela model for Fiasco IPC. In order to do this, we first examined the Fiasco IPC source code which was written in object oriented paradigm of C++. Since Promela is a process based language, the translation became quite tedious. We had to carefully track the recursion and inheritance in the original source code to be able to model it into Promela formalism. As a result, some classes were modeled as records and methods of a class were modeled by the Promela inline construct.

The problem with the above approach was that it took a lot of time to dig into all details of the Fiasco IPC code. To be able to soon grasp the behavior of Fiasco IPC, we also approach the modeling task by just translating line by line from C++ to Promela. This gives faster result, although we still have to rely on the knowledge acquired from the first approach. We found that it is wiser to combine the former approach together with the "rough translation" method from the original Fiasco IPC source code. The translation was done iteratively and each time we get a more refined Promela code.

Based on our experience, Promela is *not* hard to learn. Especially for people who have done some programming in C, Promela can be very easy. Furthermore, SPIN is also *not* a complicated application to use. It can be run by just typing some command. SPIN's graphical user interface, XSPIN, is even much more simpler since the user would not have to type in the various commands. XSPIN wraps all the complicated command-line options into a GUI where users can just click the mouse.

Various verification attempts have been done repetitively in order to evaluate the performance of the model. The properties we were interested in verifying are safety properties (assertion violations, unintended end states, and invariant properties) and liveness properties. We use SPIN default verification to check for assertion violations and unintended end of states. Whereas invariant properties and liveness properties are verified using never claims.

There were some points where the state space was too large to be exhaustively searched. This leads to huge memory use and incredibly deep search. Some optimization efforts on the Promela model via abstraction have been done to tackle the problems. But still, we have to rely on the use of Pan compile-time and run-time options to really solve the problem and get a complete search.

Collecting all the experiences and facts found during this master's thesis, we analyze them and we come to the following conclusion:

1. Promela lacks many features of implementation language such as C. We found it a little bit awkward using `inline`, because of the following reasons:

   - it cannot return values,
   - the scope of the variables declared in `inline` will be in the same scope as the point where the `inline` is invoked. This would be troublesome when there are more than one invocation of the `inline` from the original scope. The solution would be to declare the variables in the caller and then pass them as parameters everytime it needs to invoke the `inline`
   - The same problem as above arises when we have labels in `inline` which is called more than once from the same scope. The solution is not to use labels and try to find other ways to model the system which are suitable to the behavior of the system.

2. Other sources of obstacles in translating Fiasco IPC C++ code to Promela model:

   - the object-oriented paradigm of Fiasco code with its "object context" concept of a certain method invocation. This is not supported in Promela. Therefore, the translation requires careful inspection of this concept.

- the fact that Fiasco IPC code uses low-level assembler function and bit operations. For instance, one line of code in C++ of bit operations can be translated into many lines of code in Promela.

3. Although we finally were able to translate the significant C++ code into Promela, there are still some open questions about what to model and what to abstract.

4. We must take into account the following facts about Pan compile-time options for verification:

   - `-DREACH` cannot be combined with `-DBITSTATE` or with `-DMA`,
   - `-DBITSTATE` cannot be combined with `-DMA`,
   - `-DSC` cannot be combined with `-DBFS`,
   - `-DMA=N` is very helpful in compressing the states size,
   - `-DSC` helps us very well to tackle memory limitation problem, but it requires free disk space up to 10GB for the stack file.

5. *Spin Version 4.2.3 – 5 February 2005* always gives error `"pan: stackfile write error -- disk is full?"` when we compile the verifier using the option `-DSC`. This SPIN bug is solved thanks to Hendrik Tews. All the verification reported in this master's thesis uses the patched version of SPIN by Hendrik Tews.

6. Specifying correctness properties requires a careful inspection on the nature of the model's behavior. Since model checking examines all possible executions, any simple innocent situation could lead to a counterexample. Such situation, for instance, is the order of global variables initialization which are used in never claims.

7. Some correctness properties requires additional changes in the model to help monitor the behavior of the model. The worst case is that we may need to design one new model for each property.

6.2 Future Works

The model we have developed is only for normal short IPC, with timeout set to zero. Even with such a simple model, we face a very deep search in verification. We suggest to explore further to find out how to optimize the model. After it has been fixed, we suggest to develop more features of Fiasco IPC into the model.

References

[1] G. J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2004.

[2] Spin Website: http://spinroot.com/spin/index.html

[3] M. Hohmuth. The Fiasco Kernel: System Architecture. Technische Berichte - Technische Universität Dresden, Fakultät Informatik. TUD-FI02-06 (Not Yet Published). January 6, 2004.

[4] M. Hohmuth. Pragmatic nonblocking synchronization for real-time systems. PhD Dissertation - TU Dresden. October 2, 2002.

[5] M. Hohmuth. The Fiasco Kernel: Requirements Definition. Technical Report - Technische Universität Dresden, Fakultät Informatik. December 1998.

[6] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, G. Back. Formal Methods: A Practical Tool for OS Implementors. Department of Computer Science, University of Utah, 1997.

[7] A. Au, G. Heiser. L4 User Manual. School of Computer Science and Engineering, The University of South Wales. March 15, 1999.

[8] R. Gerth. Concise Promela Reference. Eindhoven University. June 1997.

[9] Basic Spin Manual. http://spinroot.com/spin/Man/Manual.html. 26 August 1997.

[10] T. C. Ruys. SPIN Beginner's Tutorial. SPIN 2002 Workshop, University of Twente, Formal Methods and Tools Group. 11 April 2002.

[11] E. M. Clarke, J. M. Wing. Formal Methods: State of the Art and Future Directions. Carnegie Mellon University, 1996.

[12] WetStone Technologies, Inc. Formal Methods Framework. October 26, 1999.

A Fiasco IPC Model in Promela

In the model below, we find many comments like `/* loc ... */`. Such comments are also found in the real Fiasco IPC code. They are meant to link the statements in the model to the statements in the real code.

Fiasco source code is available in public CVS-repository, in the branch `ipc_model_check`. Further information on how to download the source code can be found at: http://os.inf.tu-dresden.de/drops/download.html.

```
#define N 2      /* number of threads */

typedef IPC_Prm
{
   bool  has_receive_part;
   bool  open_wait;
   short partner;
   bool  has_send_part;
};

IPC_Prm ipc_g_prm[N];

typedef IPC_State
{
   bit ready;
   bit receiving;
   bit polling;
   bit ipc_in_progress;
   bit send_in_progress;
   bit busy;
   bit cancel;
   bit polling_long;
   bit busy_long;
   bit rcvlong_in_progress;
};

IPC_State ipc_state[N];

typedef Sender_List
{
   short snd_ls[N];
   byte last_index;
};

Sender_List sender_ls[N];

typedef IPC_Output
{
   short error;
   short dope;
   bool msg_copied;
```

```
   short rcv_source = -1;
};

IPC_Output ipc_output[N];

short ipc_lock_owner[N] = -1;

short rcv_partner[N];

short snd_partner[N];


inline dequeue_from_snd_ls(dfsl_snd_pid, dfsl_rec_pid, dfsl_i, dfsl_in_snd_ls)
{
   printf("dequeue_from_snd_ls(snd=%d, rec=%d, i=%d, in_ls=%d)\n",
          dfsl_snd_pid, dfsl_rec_pid, dfsl_i, dfsl_in_snd_ls);

   atomic {
      dfsl_i = 0;
      dfsl_in_snd_ls = false;

      do
      :: ((sender_ls[dfsl_rec_pid].snd_ls[dfsl_i] == dfsl_snd_pid) &&
          (sender_ls[dfsl_rec_pid].last_index != 0)) ->
            dfsl_in_snd_ls = true;
            break
      :: ((sender_ls[dfsl_rec_pid].snd_ls[dfsl_i] != dfsl_snd_pid) &&
          (dfsl_i < (sender_ls[dfsl_rec_pid].last_index - 1)))   ->
            dfsl_i++
      :: else ->
            break
      od;

      /* dequeue from sender queue if enqueued */
      if
      :: (dfsl_in_snd_ls) ->
            do
            :: (dfsl_i == (sender_ls[dfsl_rec_pid].last_index - 1)) ->
                  break
            :: else ->
                  sender_ls[dfsl_rec_pid].snd_ls[dfsl_i] =
                     sender_ls[dfsl_rec_pid].snd_ls[dfsl_i + 1];
                  dfsl_i ++
            od;
            sender_ls[dfsl_rec_pid].last_index --;

      :: else
      fi
   }
}
```

```
inline commit_ipc_success (cis_pid, cis_err)                  /* loc 80 */
{
   printf("commit_ipc_success(pid:%d, err:%d)\n", cis_pid, cis_err);
   ipc_output[cis_pid].dope = cis_err;                         /* loc 81 */
}


inline commit_ipc_failure (cif_pid, cif_err)                  /* loc 70 */
{
   printf("commit_ipc_failure(pid:%d, err:%d)\n", cif_pid, cif_err);

   /* remove loc 71 coz' we don't model delayed_ipc */
   ipc_output[cif_pid].dope = 0;                               /* loc 72 */
   commit_ipc_success(cif_pid, cif_err);
}


/* Receiver-ready callback.
   Receivers make sure to call this function on waiting senders when
   they get ready to receive a message from that sender.
   Class Thread's implementation wakes up the sender if it is still in
   sender-wait state.
*/
inline ipc_receiver_ready(irr_rec_pid, irr_snd_pid, irr_poll_temp)   /* loc 290 */
{
   printf("ipc_receiver_ready(snd:%d, rec:%d)\n", irr_snd_pid, irr_rec_pid);

   atomic {
      assert(irr_snd_pid >= 0 && ipc_lock_owner[irr_snd_pid] != irr_rec_pid);
      (ipc_lock_owner[irr_snd_pid] == -1);
      ipc_lock_owner[irr_snd_pid] = irr_rec_pid;               /* loc 291 */
   };

   atomic {
      irr_poll_temp = ipc_state[irr_snd_pid].polling;
      ipc_state[irr_snd_pid].polling = 0       /* wake up the sender:  loc 292 */
   };

   if
   :: (irr_poll_temp == 1) ->
         ipc_state[irr_snd_pid].ready = 1      /* sender is woken up:  loc 294 */
   :: else
   fi;

   atomic {
      assert(ipc_lock_owner[irr_snd_pid] == irr_rec_pid);
      ipc_lock_owner[irr_snd_pid] = -1;
   };
}


/* do_receive :: Receive an IPC message.
   Block until we can receive a message or the timeout hits.
```

```
   Before calling this function, the thread needs to call prepare_receive().
   @param dr_snd_pid : IPC partner we want to receive a message from.
                       -1 if we accept IPC from any partner (‘‘open wait’’).
   @param dr_rec_pid : receiver
*/
inline do_receive (dr_rec_pid, dr_snd_pid, dr_ret)                /* loc 250 */
{
   bit poll_temp;

   printf("do_receive(rec:%d, snd:%d, ret:%d)\n",
          dr_rec_pid, dr_snd_pid, dr_ret);

   assert(!(ipc_state[dr_rec_pid].send_in_progress ||              /* loc 251 */
            ipc_state[dr_rec_pid].polling ||
            ipc_state[dr_rec_pid].polling_long));

   if
   :: (ipc_state[dr_rec_pid].receiving &&                          /* loc 252 */
       ipc_state[dr_rec_pid].ipc_in_progress &&
       !ipc_state[dr_rec_pid].cancel) ->
         atomic {
            if
            :: (ipc_state[dr_rec_pid].receiving &&
                ipc_state[dr_rec_pid].ipc_in_progress) ->          /* loc 253 */
                  if
                  :: (!ipc_state[dr_rec_pid].busy) ->
                        ipc_state[dr_rec_pid].busy = 1             /* loc 253 */
                  :: else ->
                        goto dr1
                  fi
            :: else
                  goto dr1
            fi;
         }

         if
         :: (sender_ls[dr_rec_pid].last_index == 0)                /* loc 254 */
         :: else ->
               if
               :: (dr_snd_pid == -1 ||                             /* loc 255 */
                   dr_snd_pid == sender_ls[dr_rec_pid].snd_ls[0]) ->
                     ipc_receiver_ready(dr_rec_pid,
                                        sender_ls[dr_rec_pid].snd_ls[0],
                                        poll_temp)                 /* loc 256 */
               :: else ->
                     byte dr_i;
                     bool dr_in_snd_ls;

                     dr_i = 0;
                     dr_in_snd_ls = false;

                     do
                     :: ((sender_ls[dr_rec_pid].snd_ls[dr_i] == dr_snd_pid) &&
```

```
                    (sender_ls[dr_rec_pid].last_index != 0)) ->
                      dr_in_snd_ls = true;
                      break
                :: ((sender_ls[dr_rec_pid].snd_ls[dr_i] != dr_snd_pid) &&
                    (dr_i < (sender_ls[dr_rec_pid].last_index - 1))) ->
                      dr_i++
                :: else ->
                      break
                od;

                if
                :: (dr_in_snd_ls &&
                    dr_rec_pid == snd_partner[dr_snd_pid]) ->   /* loc 257 */
                      ipc_receiver_ready(dr_rec_pid,            /* loc 258 */
                                         dr_snd_pid, poll_temp)
                :: else
                fi
          fi
      fi;

      /* remove loc 259-261, 263-266, coz' timeout is set to be always 0 */
      atomic {                                                  /* loc 262 */
         ipc_state[dr_rec_pid].ipc_in_progress = 0;
         ipc_state[dr_rec_pid].busy = 0;
      }

:: else
fi;

/* ipc_continue */
dr1:
   assert(! (ipc_state[dr_rec_pid].ipc_in_progress ||             /* loc 276 */
             ipc_state[dr_rec_pid].send_in_progress ||
             ipc_state[dr_rec_pid].polling ||
             ipc_state[dr_rec_pid].polling_long));

   if
   :: (ipc_state[dr_rec_pid].receiving ||                         /* loc 277 */
       ipc_state[dr_rec_pid].busy ||
       ipc_state[dr_rec_pid].rcvlong_in_progress ||
       ipc_state[dr_rec_pid].busy_long) ->

         if
         :: (ipc_state[dr_rec_pid].busy) ->                       /* loc 278 */
               commit_ipc_success (dr_rec_pid, -7);    /* Reaborted: loc 279 */
         :: else ->
               if
               :: (ipc_state[dr_rec_pid].cancel) ->               /* loc 280 */
                     commit_ipc_success (dr_rec_pid, -5) /*Recanceled:loc 281*/
               :: else ->
                     commit_ipc_success (dr_rec_pid, -3)  /*Retimeout:loc 282*/
                     atomic {                                     /* loc 283 */
                        ipc_state[dr_rec_pid].receiving = 0;
```

```
                        ipc_state[dr_rec_pid].busy = 0;
                        ipc_state[dr_rec_pid].rcvlong_in_progress = 0;
                        ipc_state[dr_rec_pid].busy_long = 0;
                     }
                fi
           fi
      :: else
      fi;

      dr_ret = ipc_output[dr_rec_pid].dope;
}


inline wake_receiver(wr_snd_pid, wr_rec_pid)                         /* loc 200 */
{
   printf("wake_receiver(snd:%d, rec:%d)\n", wr_snd_pid, wr_rec_pid);

   /* Remove loc 201 coz' we don't model delayed_ipc */
   /* Just update the receiver's state */

   atomic {                                                          /* loc 202 */
      ipc_state[wr_rec_pid].ipc_in_progress = 0;
      ipc_state[wr_rec_pid].send_in_progress = 0;
      ipc_state[wr_rec_pid].polling = 0;
      ipc_state[wr_rec_pid].polling_long = 0;
      ipc_state[wr_rec_pid].receiving = 0;
      ipc_state[wr_rec_pid].busy = 0;
      ipc_state[wr_rec_pid].rcvlong_in_progress = 0;
      ipc_state[wr_rec_pid].busy_long = 0;

      ipc_state[wr_rec_pid].ready = 1
   }
}


inline ipc_init(ii_snd_pid, ii_rec_pid)
{
   printf("ipc_init(snd:%d, rec:%d)\n", ii_snd_pid, ii_rec_pid);
   rcv_partner[ii_rec_pid] = ii_snd_pid
}


/* Unlock a receiver locked with ipc_try_lock(). */
inline ipc_unlock(iu_rec_pid, iu_snd_pid)
{
   printf("ipc_unlock(rec:%d, snd:%d)\n", iu_rec_pid, iu_snd_pid);
   printf("ipc_lock_owner[%d] = %d\n", iu_rec_pid, ipc_lock_owner[iu_rec_pid]);

   atomic {
      assert (ipc_lock_owner[iu_rec_pid] == iu_snd_pid);             /* loc 191 */
      ipc_lock_owner[iu_rec_pid] = -1                                /* loc 192 */
   }
}
```

```
/* Return whether the receiver is ready to accept a message from the given sender.
   @param so_snd_pid thread that wants to send a message to @param so_rec_pid
   @return so_ret true if receiver is in correct state to accept a message
   right now (open wait, or closed wait and waiting for sender).
*/
inline sender_ok(so_snd_pid, so_rec_pid, so_ret)                    /* loc 240 */
{
   printf("sender_ok(snd:%d, rec:%d)\n", so_snd_pid, so_rec_pid, so_ret);

   /* Skip Calculating ipc_state: loc 241 */

   /* If Thread_send_in_progress is still set, we're still in the send phase */
   if
   :: (!ipc_state[so_rec_pid].receiving ||
       !ipc_state[so_rec_pid].ipc_in_progress ||
       ipc_state[so_rec_pid].send_in_progress) ->                   /* loc 242 */
         so_ret = false                                             /* loc 243 */
   :: else ->
         /* Check open wait; test if this sender is really the first in queue */
         if
         :: (rcv_partner[so_rec_pid] == -1) ->                      /* loc 244 */
               so_ret = true                                        /* loc 245 */
         :: else ->
               if
               /* Check closed wait; test if this sender is who we specified */
               :: (so_snd_pid == rcv_partner[so_rec_pid]) ->        /* loc 246 */
                     so_ret = true                                  /* loc 247 */
               :: else ->
                     so_ret = false                                 /* loc 248 */
               fi
         fi
   fi;
}


/* Try to start an IPC handshake with this receiver.
   Check the receiver's state, checks if the receiver is acceptable at
   this time, and if OK, "lock" the receiver and copy the sender's ID
   to the receiver's lock.
   @param itl_snd_pid: the sender that wants to establish an IPC handshake
   @return  0 for success,
           -1 in case of a transient failure,
           an IPC error code if an error occurs.
*/
inline ipc_try_lock (itl_snd_pid, itl_rec_pid, itl_ret)             /* loc 220*/
{
   bool sok;

   printf("ipc_try_lock(snd:%d, rec:%d, ret:%d)\n",
          itl_snd_pid, itl_rec_pid, itl_ret);
   if
```

```
   :: (itl_rec_pid >= N) ->                                       /* loc 221 */
        itl_ret = -2                        /* Ipc_err::Enot_existent; loc 222 */
   :: else ->
        atomic {
           assert(ipc_lock_owner[itl_rec_pid] != itl_snd_pid);
           (ipc_lock_owner[itl_rec_pid] == -1);
           ipc_lock_owner[itl_rec_pid] = itl_snd_pid;             /* loc 223 */
        }

        sender_ok(itl_snd_pid, itl_rec_pid, sok);                 /* loc 224 */
        printf("sender_ok returned %d\n", sok);

        if
        :: sok ->                                                 /* loc 224 */
             itl_ret =  0                                     /* OK, loc 227 */
        :: else ->
             atomic {                                             /* loc 225 */
                assert(ipc_lock_owner[itl_rec_pid] == itl_snd_pid);
                ipc_lock_owner[itl_rec_pid] = -1;
             }
             itl_ret = -1                                         /* loc 226 */
        fi
   fi;
}


inline ipc_snd_regs (isr_snd_pid, isr_rec_pid, isr_ret)          /* loc 150 */
{
   printf("ipc_snd_regs(snd:%d, rec:%d, ret:%d)\n",
          isr_snd_pid, isr_rec_pid, isr_ret);

   ipc_try_lock (isr_snd_pid, isr_rec_pid, isr_ret);              /* loc 151 */
   printf("ipc_try_lock returned %d\n", isr_ret);

   if
   :: (isr_ret != 0)                                              /* loc 152 */
   :: else ->
        if
        :: (ipc_state[isr_snd_pid].cancel) ->                     /* loc 155 */
             ipc_unlock(isr_rec_pid, isr_snd_pid);                /* loc 156 */
             isr_ret = -6                           /* Ipc_err::Secanceled */

        :: else ->
             ipc_init(isr_snd_pid, isr_rec_pid);                  /* loc 157 */

             /* Skip Checking Deceiving IPC:  loc 158 */

             ipc_output[isr_snd_pid].dope = 0;                    /* loc 159 */
             isr_ret = 0;            /* loc 160: status code: IPC successful*/

             byte isr_i;
             bool isr_in_snd_ls;
             isr_i = 0;
```

```
               isr_in_snd_ls = false;

               /* dequeue from sender queue if enqueued */          /* loc 161 */
               dequeue_from_snd_ls(isr_snd_pid, isr_rec_pid,
                                   isr_i, isr_in_snd_ls);

               /* copy message register contents */
               ipc_output[isr_snd_pid].msg_copied = true;           /* loc 168 */
               ipc_output[isr_rec_pid].msg_copied = true;

               /* copy sender ID */
               ipc_output[isr_rec_pid].rcv_source = isr_snd_pid;    /* loc 169 */

               /* IPC done -- reset states */
               atomic {                                             /* loc 178 */
                  ipc_state[isr_snd_pid].polling = 0;
                  ipc_state[isr_snd_pid].send_in_progress = 0;
               }

               wake_receiver (isr_snd_pid, isr_rec_pid);            /* loc 179 */

               ipc_unlock(isr_rec_pid, isr_snd_pid);                /* loc 189 */
         fi
    fi;
}


/* @param ds_pid  : sender
   @param ds_partner : receiver
   @param ds_ret : sender's IPC error code
*/
inline do_send(ds_pid, ds_partner, ds_ret)                          /* loc 1 */
{
   printf("do_send(pid:%d, partner:%d, ret:%d)\n",
          ds_pid, ds_partner, ds_ret);
   if
   :: (ds_partner == -1 || ds_partner == -2 || ds_partner >= N) ->   /* loc 2 */
         printf("do_send: Error Not Existent Partner\n");
         ds_ret = -2                                                /* loc 3 */
   :: else ->

         /* ========================= SETUP PHASE ========================= */

         snd_partner[ds_pid] = ds_partner;                          /* loc 5 */

         /* putting the sender into state "send_prepared" */        /* loc 6 */
         atomic {
            ipc_state[ds_pid].polling = 1;
            ipc_state[ds_pid].ipc_in_progress = 1;
            ipc_state[ds_pid].send_in_progress = 1;
         }

         if
```

```
:: (ipc_state[ds_pid].cancel) ->                          /* loc 7 */
      atomic {                                            /* loc 8 */
         ipc_state[ds_pid].send_in_progress = 0;
         ipc_state[ds_pid].polling = 0;
         ipc_state[ds_pid].polling_long = 0;
         ipc_state[ds_pid].ipc_in_progress = 0;
      }

      /* ds_ret = Send canceled */
      ds_ret = -6

:: else ->
      /* enqueue in sender list of partner */
      printf("enqueue myself (%d) in sender list of partner
             (partner:%d, current index:%d)\n",
             ds_pid, ds_partner, sender_ls[ds_partner].last_index);

      atomic {                                            /* loc 9 */
         sender_ls[ds_partner].
            snd_ls[sender_ls[ds_partner].last_index] = ds_pid;
         sender_ls[ds_partner].last_index =
            sender_ls[ds_partner].last_index + 1;
      };

      /* =================== RENDEZVOUZ PHASE ==================== */

      /* try a rendezvous with the partner */
      ipc_snd_regs(ds_pid, ds_partner, ds_ret);           /* loc 10 */
      printf("ipc_snd_regs returned %d\n", ds_ret);

      if
      /* transient error*/
      :: (ds_ret == -1) ->                                /* loc 11 */
            /* TIMEOUT */
            atomic {
               ipc_state[ds_pid].polling = 0;
               ipc_state[ds_pid].ipc_in_progress = 0;
               ipc_state[ds_pid].send_in_progress = 0;
            }

            /* dequeue in sender list of partner */
            byte ds1_i;
            bool ds1_in_snd_ls;

            ds1_i = 0;
            ds1_in_snd_ls = false;
            dequeue_from_snd_ls(ds_pid, ds_partner,       /* loc 14 */
                                ds1_i, ds1_in_snd_ls);

            /* Send timeout */
            ds_ret = -4;

      :: else ->
```

```
byte i;
bool in_snd_ls;

i = 0;
in_snd_ls = false;
atomic {
   do
   :: ((sender_ls[ds_partner].snd_ls[i] == ds_pid) &&
       (sender_ls[ds_partner].last_index != 0))->
         in_snd_ls = true;
         break
   :: ((sender_ls[ds_partner].snd_ls[i] != ds_pid) &&
       (i < (sender_ls[ds_partner].last_index - 1))) ->
         i++
   :: else ->
         break
   od;
};

if
:: (in_snd_ls) ->                                /* loc 28 */
      assert (ds_ret != 0);                      /* loc 29 */

      /* dequeue in sender list of partner --      loc 30 */
      dequeue_from_snd_ls(ds_pid, ds_partner, i, in_snd_ls);

:: else
fi;

if                                               /* loc 32 */
:: (ds_ret == 0 && ipc_state[ds_pid].send_in_progress) ->
      printf("ret = do_send_long(ds_partner\n");/* loc 32 */
      assert(false);                    /* no long ipc yet */
:: else
fi;

if
:: (ipc_state[ds_pid].receiving ||               /* loc 33 */
    ipc_state[ds_pid].busy ||
    ipc_state[ds_pid].rcvlong_in_progress ||
    ipc_state[ds_pid].busy_long) ->
      atomic {                                   /* loc 33 */
         ipc_state[ds_pid].send_in_progress = 0;
         ipc_state[ds_pid].polling = 0;
         ipc_state[ds_pid].polling_long = 0
      }
:: else ->
      atomic {                                   /* loc 33 */
         ipc_state[ds_pid].send_in_progress = 0;
         ipc_state[ds_pid].polling = 0;
         ipc_state[ds_pid].polling_long = 0;
         ipc_state[ds_pid].ipc_in_progress = 0
      }
```

```
                    fi
              fi
        fi
   fi;
}


/* Prepare an IPC-receive operation.
   This method must be called before do_receive() and, when carrying out
   a combined snd-and-receive operation, also before do_send().
   @param pr_snd_pid: IPC partner we want to receive a message from.
        -1 if we accept IPC from any partner (``open wait'').
*/
inline prepare_receive(pr_rec_pid, pr_snd_pid, pr_has_send_part)     /* loc 50 */
{
   printf("prepare_receive(rec:%d, snd:%d, has_send:%d)\n",
          pr_rec_pid, pr_snd_pid, pr_has_send_part);

   /* pr_snd_pid might be -1 for open_wait / receive from nil thread */
   rcv_partner[pr_rec_pid] = pr_snd_pid;                             /* loc 52 */
   ipc_state[pr_rec_pid].receiving = 1;                              /* loc 51 */

   if
   :: (!pr_has_send_part) ->                                         /* loc 53 */
         ipc_state[pr_rec_pid].ipc_in_progress = 1;                  /* loc 54 */
         if
         :: (ipc_state[pr_rec_pid].cancel) ->                        /* loc 55 */
               ipc_state[pr_rec_pid].ipc_in_progress = 0             /* loc 56 */
         :: else
         fi
   :: else
   fi;
}


/* L4 IPC system call */
active [N] proctype thread() provided (ipc_lock_owner[_pid] == -1 ||
                                       ipc_lock_owner[_pid] == _pid) /* loc 90 */
{
   bool have_sender;                                                 /* loc 92 */
   short sender;                                                     /* loc 93 */

   do
   ::
        /* initialize local var */
        have_sender = false;
        sender = -1;

        /* initialize global var ipc_state */
        ipc_state[_pid].receiving = 0;
        ipc_state[_pid].polling = 0;
        ipc_state[_pid].ipc_in_progress = 0;
        ipc_state[_pid].send_in_progress = 0;
```

```
ipc_state[_pid].busy = 0;
ipc_state[_pid].cancel = 0;
ipc_state[_pid].polling_long = 0;
ipc_state[_pid].busy_long = 0;
ipc_state[_pid].rcvlong_in_progress = 0;

/* initialize global var ipc_output */
ipc_output[_pid].error = -10;
ipc_output[_pid].dope = -10;
ipc_output[_pid].msg_copied = false;
ipc_output[_pid].rcv_source = -1;

assert(ipc_lock_owner[_pid] != _pid);

/* nondeterministic choice of IPC parameters */
atomic {
   if
   :: ipc_g_prm[_pid].has_receive_part = false
   :: ipc_g_prm[_pid].has_receive_part = true
   fi;

   if
   :: ipc_g_prm[_pid].open_wait = false
   :: ipc_g_prm[_pid].open_wait = true
   fi;

   if
   :: ipc_g_prm[_pid].partner = -2
   :: ipc_g_prm[_pid].partner = 0
   :: ipc_g_prm[_pid].partner = 1
   :: ipc_g_prm[_pid].partner = 2
   fi;

   if
   :: ipc_g_prm[_pid].has_send_part = false
   :: ipc_g_prm[_pid].has_send_part = true
   fi;
};

printf("IPC thread:%d partner:%d snd:%d rcv:%d open:%d\n",
       _pid, ipc_g_prm[_pid].partner,
       ipc_g_prm[_pid].has_send_part,
       ipc_g_prm[_pid].has_receive_part,
       ipc_g_prm[_pid].open_wait);

ipc_state[_pid].ready = 1;

/* Skip Next Period IPC */

if
:: (ipc_g_prm[_pid].has_receive_part) ->                    /* loc 99 */
      if
      :: (ipc_g_prm[_pid].open_wait) ->                     /* loc 100 */
```

```
            have_sender = true                              /* loc 101 */

        :: else ->
            if
            /* partner == nil_thread?? */
            :: (ipc_g_prm[_pid].partner == -2)              /* loc 102 */

            :: else ->

                /* Skip Checking Partner is IRQ */

                if
                /* partner null or does not exist? */
                :: (ipc_g_prm[_pid].partner == -1 ||        /* loc 113 */
                    ipc_g_prm[_pid].partner >= N) ->
                      printf("commit_ipc_failure(Enot_existent)\n");
                      commit_ipc_failure(_pid, -2);         /* loc 114 */
                      goto noop
                :: else ->

                    /* Skip Checking Preemption IPC */
                    sender = ipc_g_prm[_pid].partner;       /* loc 117 */
                    have_sender = true
                fi
            fi
        fi;

        if
        :: (have_sender) ->                                 /* loc 118 */
            printf("prepare_receive(sender = %d)\n", sender);
            prepare_receive(_pid, sender, ipc_g_prm[_pid].has_send_part)
        :: else
        fi

    :: else
    fi;

    if
    :: (ipc_g_prm[_pid].has_send_part) ->                   /* loc 119 */
                                                            /* loc 120 */
        do_send(_pid, ipc_g_prm[_pid].partner, ipc_output[_pid].error)
        printf("do_send returned %d\n", ipc_output[_pid].error);

    :: else ->
        ipc_output[_pid].dope = 0                           /* loc 121 */
    fi;


    /* Send Finished, Do Receive Now */

    if                                                      /* loc 122 */
    :: (ipc_g_prm[_pid].has_receive_part && ipc_output[_pid].error >= 0) ->
        if
```

```
            :: (have_sender) ->                                 /* loc 123 */
                  do_receive(_pid, sender, ipc_output[_pid].error) /* loc 124 */
                  printf("do_receive returned %d\n", ipc_output[_pid].error);

                  if
                  :: (ipc_output[_pid].error != -3) ->          /* loc 125 */
                        goto success
                  :: else
                  fi
            :: else                    /* Skip Checking Partner is Valid IRQ */
            fi;

            /* Skip Checking Partner is a Free IRQ */

            printf("commit_ipc_failure(Retimeout)\n");
            commit_ipc_failure(_pid, -3);                      /* loc 132 */

      :: else ->
            atomic {                                           /* loc 134 */
               ipc_state[_pid].ipc_in_progress = 0;
               ipc_state[_pid].send_in_progress = 0;
               ipc_state[_pid].polling = 0;
               ipc_state[_pid].polling_long = 0;
               ipc_state[_pid].receiving = 0;
               ipc_state[_pid].busy = 0;
               ipc_state[_pid].rcvlong_in_progress = 0;
               ipc_state[_pid].busy_long = 0
            }
      fi;

      success:
         /*commit_ipc_success */
         printf("commit_ipc_success(ret)\n");
         commit_ipc_success(_pid, ipc_output[_pid].error)       /* loc 135 */
      noop:
         skip
   od
}
```