Verification of Fiasco's IPC implementation

Master thesis
Number 560

by

E.G.H. Schierboom
<eschierb@sci.kun.nl>

at

Radboud Universiteit Nijmegen
Computing Science Department
Toernooiveld 1
6525 ED Nijmegen
Holland

supervised by

dr. M.C.J.D. van Eekelen
dr. J.E.W. Smetsers
dr. W. Tews

June 21, 2007

Abstract

Software nowadays is often designed with dependability in mind. Our thesis combines two approaches in creating a more dependable system: using microkernels instead of monolithic kernels and formally verifying software. We have tried to verify three properties of inter-process communication in the Fiasco microkernel. As Fiasco has been written in C++, which does not support verification, we converted the source code to a model in the PVS proof system. To keep the model and proofs compact, we abstracted away many details of inter-process communication.

Two of the three properties were verified; both dealt with threads possibly waiting forever. The third property, verification of the assertions in the source code, posed several problems. One problem proved insurmountable, probably due to the abstractions applied. Another problem led to the finding of a bug in Fiasco's IPC implementation. Although finding the bug had clear, practical use, we consider the fact that our abstract model could find the bug more important. It shows that one does not have to create a one-on-one model to apply (partial) verification; even our model in which essential components were abstracted away sufficed to find a bug.

Preface

To me, computers are fascinating machines in all their aspects. I have always favored software over hardware though, writing my own software was what got me into studying computer science after all. Even though computers play a vital role in society nowadays, one must not forget that computer science is a relatively young field of research[1]. It is therefore not surprising that there are still many problems with computers, although progress is being made rapidly (which is another interesting aspect of computer science). As hardware leaps forward[2], the general impression is that the software lags behind. This impression stems from the fact that most problems with computers are due to the software running on them.

Lately, one of the most interesting developments in writing software, at least to me, has been the shift in focus from performance to dependability[3]. This shift can, for a large part, be attributed to the changing needs of people. Where one was previously happy that a computer could execute a task, people are now so accustomed to computers that they expect that task to be executed dependably. Unfortunately, if one would ask a random person if he or she has had any bad experiences with the dependability of software, chances are very high that most would answer with a resounding "yes". The prime example of software not being dependable is when it, or the whole system, crashes.

Almost all software on business or private computers runs on top of (and thus depends on) an operating system, the most well known probably being Microsoft Windows, Unix/Linux and Mac OS. To me, operating systems are the most interesting pieces of software available because of their vital role and the very diverse tasks they execute. When one wants to create dependable software (which relies on the operating system), it makes sense to create a dependable operating system. In an operating system, the most vital part (or "heart") is the kernel, which dependability is therefore absolutely critical.

In line with the desire to create dependable software, verification of software is a rapidly expanding field of computer science. What can be better to its dependability than to formally verify that a piece of software does what it is supposed to do? In our thesis, we will combine both our interest in operating system kernels and software verification. Our research shows how one can verify (a part of) a real kernel. This is particularly useful as kernels are among the most likely candidates for verification because of their vital role. We hope that our thesis will kindle your interest in verification of software, as we expect it to become an integral part of software development in the, hopefully near, future.

[1] Especially when compared to other exact sciences such as mathematics and physics, which predate it by many centuries.
[2] Processors are an obvious example, as Moore's law has successfully been applied to them for the last 30 years.
[3] Which loosely translates to software behaving like one expects it to.

Acknowledgements

In writing this thesis, I have received help from a great many people. First and foremost I would like to thank Marko van Eekelen, Hendrik Tews and Sjaak Smetsers for the great assistance they have given me. They have helped me with many different aspects of this thesis, from the verification of individual lemmas to planning the research itself. Secondly, I want to thank René Reusner and Adam Lackorzynski for never becoming tired of answering all my questions regarding the Fiasco IPC implementation. In this regard, the L4 hackers mailinglist has also been very helpful; their responses were always fast and to the point. The structure and contents of this document have been extensively commented upon by Jens Wagemakers, for which I owe him many thanks. For a more general discussion of my thesis I often turned to Sietse Overbeek, with whom I had some very useful discussions. When a technical problem arose, Engelbert Hubbers was always quick to solve it.

Besides the abovementioned people, who were directly involved with the thesis' contents, I have also received support from a lot of people that did not involve the contents of this thesis. Most notably, these include my friends, colleagues and family and of course my girlfriend, Jeltine Jungheim. I want to specifically mention my dad who has always supported me in whatever I did, but unfortunately passed away far too young; this thesis is dedicated to him.

I hereby apologize to anyone that has not made it in this list, that does not mean I did not appreciate your help but merely that my memory has failed me.

Contents

List of Figures

List of Tables

List of Sources

Chapter 1

Introduction

"To err is human - and to blame it on a computer is even more so."

Robert Orben

"I do not fear computers. I fear the lack of them."

Isaac Asimov

Writing computer programs has changed a lot in the last decennia. The programs written for the first computers had to keep in mind that its resources (such as processing power and memory) were very limited and thus efficiency was a key design goal. As computers advanced, its resources became more abundant so efficiency became less an issue. Instead, as reliance on computers increased, building a more dependable system became a key design goal. The concept of dependability is defined by the IFIP[1] [IFI05] as follows:

> "The notion of dependability, defined as the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers, enables these various concerns to be subsumed within a single conceptual framework. Dependability thus includes as special cases such attributes as reliability, availability, safety, security."

To create a dependable computer system it makes sense to start looking at the foundations of the software running on a computer, namely the kernel. Critical functions such as memory allocation, scheduling, process management[2] and inter-process communication (IPC) are all handled by the kernel. Errors in the kernel decrease its dependability and without a dependable kernel we cannot reasonably expect the whole system to be dependable[3]. One of the main issues in creating a dependable system is thus: how to create a dependable kernel?

One approach to create a dependable kernel is to make it a microkernel. The main design motivation when designing a microkernel is summarized by Jochen Liedtke in [Lie95]:

> "[...] a concept is tolerated inside the microkernel only if moving it outside the kernel [...] would prevent the implementation of the system's required functionality."

A microkernel is thus a specific type of kernel in which anything that does not necessarily belong in the kernel is moved outside the kernel. This results in a smaller and more robust system. A monolithic kernel however includes, often many, concepts that do not necessarily belong in the kernel. A typical

[1] The International Federation For Information Processing.
[2] Such as the creation and stopping of processes. A process is sometimes also referred to as a task.
[3] Remember that each task relies on the kernel for its core functionality.

concept that a monolithic kernel includes but a microkernel does not is the file server (or file system). The main benefit of including a concept in the kernel is improved performance, as switching between user- and kernel-mode is quite expensive [HHL+97]. Its main disadvantage is that a crash of the concept often brings down the entire kernel (and thus the entire system). When a concept resides in user space, a crash of that concept has a significantly smaller chance of bringing down the entire system as it cannot directly access the kernel space (in which the kernel resides).

Although a microkernel is more compact than a monolithic kernel, errors are still likely to exist. Each error in the kernel negatively influences its dependability, the question is therefore: how to minimize the number of errors in the kernel? One approach is to formally verify the kernel. Formally verifying software is very similar to creating a mathematical proof[4]. Just as a mathematical proof lets you say things with absolute certainty about the proven statement, formal software verification lets you say things with absolute certainty about the verified software. Being able to guarantee that certain properties hold when executing software is incredibly helpful in developing a dependable system; one could for example verify that the system will never be in a certain, erroneous state.

This thesis will report on our attempt to formally verify a part of the Fiasco microkernel, which is based on Jochen Liedtke's L4 microkernel specification [Lie96] and has been developed by the Technische Universität Dresden. We will try to formally verify certain aspects of inter-process communication in Fiasco. Our research question is as follows:

What properties of inter-process communication on the Fiasco microkernel can be proven?

The verification will be done by creating a model of inter-process communication in Fiasco and then verify properties of that model. The verification system PVS[5] will be used to verify and create the model. We have chosen specifically for Fiasco because it is the subject of a larger verification project, called VFiasco [HTS03], in which our research can easily be integrated. PVS has been chosen mainly because it has thus far been the verification system of choice in the VFiasco project.

After this short introduction into the problem area, we continue at chapter two with the thesis' backgrounds. Chapter three discusses how the PVS model was created. The verification of several properties of the model is detailed in chapter four. A discussion of the obtained results is presented in chapter five and in the sixth and final chapter conclusions and suggestions for future work are presented.

[4] Computerized mechanical proofs are often regarded upon with high scepticism by mathematicians though.
[5] Prototype Verification System.

Chapter 2

Backgrounds

"There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C. A. R. Hoare

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Knuth

2.1 Improving software dependability

Dependability (of which security is a special case) has become more and more important of late. The definition of dependability has been given in the introduction, so we will now focus on how to create dependable software. There are many approaches on how to create dependable software; we will list those we consider to be the most important. Please note that the proposed solutions are not mutually exclusive, in fact some of them have already been combined [HTS03, HLA+05].

2.1.1 Safe language

A lot of software errors are not due to incorrect designs, but due to incorrect programming. The most well known problem is probably that of a *buffer overflow*. Two of the most used languages, C and C++, offer no inherent protection against buffer overflows and are thus vulnerable to this type of memory error. A solution is to design a language that is memory safe, which means that no buffer overflows can occur. Examples of these languages are C# and Java. The problem with these languages is that they offer no fine-grained control over memory (de)allocation, which is needed for common performance optimizations to be applied.

As maximizing performance is still vital to many programs (such as device drivers, computer games or kernels), these safe languages are often not an option. The approach taken by Cornell University was to develop a dialect of C, named Cyclone [JMG+02], which preserves its syntax and semantics, but also prevents some of the most common errors in the C language (such as the aforementioned buffer overflow) by using new (albeit very similar) syntax.

Another approach is to define a completely new, safe language. This approach is taken by the John

Hopkins University in their Coyotos project [Uni06]. They have developed a safe language, called BitC [JS06], with fully specified semantics. Their approach is directly related to the next solution for a dependable system: code verification.

2.1.2 Code verification

One of the most rigorous methods to improve the dependability of software is to verify its correctness. Verification of code is done by first creating a specification of the code in a verification language and then use that language's verification capabilities to verify the required properties[1]. Examples of verification languages, which are specifically tailored to constructing proofs, are PVS [Owr], Isabelle/HOL [NPW02], BoogiePL [DL05] and Abstract Machine Notation [SN01].

The problem with specifying code in a verification language is that often no executable code can be generated from it. To still be able to prove properties of executable code, there are two options: convert the verification language to an executable language or vice versa. For both options it is of vital importance that the conversion retains the exact semantics of the source language, otherwise the proofs would not necessarily apply to the corresponding source code or the executable code might not do what the source specification said it would do. The problem with a semantics-retaining conversion is that both languages need to have a clear and well-defined semantics. Unfortunately, the widely used C++ language lacks in this area. Although attempts have been made to develop a clear and well-defined semantics for C++ (which will be discussed later in this chapter), the results so far are still lacking in their applicability to real-world scenarios. The Java language though *does* have a well-defined semantics; the LOOP compiler [vdBJ01] proved that it was feasible to automatically convert Java to a higher-order logic. As a demonstration of its applicability to real-world code, Huisman et. al. used the LOOP compiler to successfully verify a non-trivial property of Java's Vector class [HJvdB99]. Unfortunately, the LOOP compiler is not yet generally applicable as it currently does not support threads[2].

Even if one has verified properties of source code, it is ultimately the machine code that gets executed. Therefore, it is the machine code that ultimately has to be verified. If one assumes that the conversion from source code to machine code, which is done by the compiler, retains semantics, verification of the source code suffices. Curzon described in [Cur92] how a compiler can be verified.

As said, conversion from certain languages (such as C++) to a verification language has proven to be very hard. An alternative to an automated conversion is to create a model of the source code in the verification language. This has two clear problems; first of all the conversion needs to be done *by-hand* most of the time, which is a far more tedious and error-prone operation than automatic conversion, and secondly there can be no guarantee that proofs in the verification language apply to the source code (as the semantics of the model might not match those of the source code).

To assist the prover[3] reason about correctness, some languages support the use of source code annotations, which can be used in the verification language. Annotation systems have been developed for a wide variety of languages, such as C [EGHT94], Java [LBR99, HHJT98] and a dialect of C# called Spec# [BLS04].

2.1.3 Minimized kernel

The most important part of an operating system (OS) is the kernel, as it handles critical OS parts such as process communication, scheduling and memory management. There are basically two different types of kernels: monolithic- and microkernels. A monolithic kernel executes all its code in the same address space in order to improve performance, whereas a microkernel tries to execute as much of its

[1] An example of such a property is requiring that the program always terminates.

[2] This is especially unfortunate considering the latest trend of developing multi-threaded applications, which is due to multi-processor systems becoming increasingly widespread.

[3] The program with which proofs are constructed in a verification language.

functionality in user space. A microkernel can be seen as a minimized version of a monolithic kernel. The basic requirements for a microkernel are given in [Lie95].

Currently two types of microkernels are identified: first generation microkernels such as Mach [RJO+89] and Minix [HBG+06] and second generation microkernels such as QNX [Hil92] and Fiasco [HTS03]. First generation microkernels still contained code in the kernel which were not strictly necessary for executing its core tasks, second generation microkernels moved this code outside the kernel. Second generation microkernels are thus more *strict* microkernels.

Usually, a microkernel's tasks are limited to address space management, thread management and IPC. By restricting its functionality to these core concepts, a microkernel is typically much smaller than a monolithic kernel and thus less likely to contain bugs. Furthermore, the removal of many concepts from the kernel results in a system less likely to crash. Take for example the file server, a typical concept that is included in monolithic kernels (because of its importance on the performance of the kernel), but is not part of a typical microkernel. In the monolithic kernel, the file server has access to all kernel data; a crash of the file server can therefore result in a crash of the whole kernel (and thus everything running on the kernel). However, in the microkernel the file server executes in user space and cannot modify the kernel data. Therefore, a crash of the file server is not likely to crash the microkernel on which it is running.

2.1.4 Isolation

Another way to create a more dependable system is to contain processes in small, isolated spaces. When a process is isolated, it cannot do any harm to other (isolated) processes. There are two types of isolated processes: software isolated processes (SIPs) and hardware isolated processes (HIPs)[4]. In practice, many operating systems do not enforce strict process isolation, which results in a system where programs can modify other programs, with potentially very damaging results (for example a single driver's failure crashes the whole system).

Hardware isolation of processes can be done by the OS, which limits a processes' memory access to specific pages of physical memory. The OS usually achieves this through standard hardware mechanisms, for example by running user processes in a less priviledged mode than kernel processes[5]. The mechanism an OS typically uses to hardware-isolate processes is virtual memory. Even though virtual memory is directly supported by most processors, studies show that this form of process isolation is in fact quite costly [AFH+06, MHH02].

Software isolation depends on the software to isolate processes. The Singularity operating system [HLA+05] is designed with software isolation in mind and also heavily uses language safety [AFH+06, FL06, HAB+06]. In short, in Singularity each application runs in its own, private software *box*, which ensures that the application can only modify data in its own box.

Another form of software isolation is called virtualization. Normally the OS is the lowest software layer on a system, but virtualization adds another layer below the OS. Each virtualized OS thinks it has access to all hardware, but in fact hardware access is controlled by the virtualization layer. The process that manages the virtualization of hardware is often referred to as the *hypervisor*[6]. Now several OSes (and its processes) can run simultaneously without interfering with each other (although the hypervisor might allow some limited and controlled communication between OSes), because the hypervisor ensures that an OS cannot access the hardware issued to another OS.

Traditionally, virtualization of x86 processors is hard as the x86 architecture does not meet the Popek and

[4] The terms SIP and HIP are taken from [AFH+06].

[5] On x86 processors, processes running at ring 0 are given full control of the processor and lower rings have limited access.

[6] This is a reference to the OS sometimes being referred to as a supervisor. Obviously the virtualization layer has more control than the OS, therefore the term hypervisor as it supersedes supervisor.

Goldberg virtualization requirements [PG74]. The main x86 processor manufacturers, AMD and Intel, acknowledged the problems with virtualization and have recently built extensions in their processors to support virtualization[7]. The virtualization layer can now translate the *difficult* OS calls to alternative, virtualization suitable calls, resulting in better virtualization performance.

Another way to deal with the *difficult* system calls is to apply paravirtualization. This means that the OS is modified to the extent that no *difficult* calls will be made, therefore no processor-specific virtualization extensions have to be used. Unfortunately, OS modification is not possible for closed-source systems without their creators' explicit consent.

2.2 L4 microkernel

2.2.1 Introduction

A performance analysis of first generation microkernels, such as Mach, revealed that they failed to deliver a high performance microkernel [HHL+97]. Jochen Liedtke realized that one of the key reasons why Mach did not perform very well, was its complex IPC implementation. He then set himself to creating a microkernel specification that *would* offer high performance; this resulted in the L3 microkernel specification [Met96]. The L3 specification had a small and efficient IPC specification, which indeed offered massive performance gains when compared to earlier microkernels such as Mach. However, Liedtke noted that the specification still contained concepts that could (and should) be moved out of the microkernel. The removal of these concepts resulted in the L4 specification. By definition the L4 microkernel specification is thus a second generation microkernel[8].

The original L4 microkernel specification is the L4.V2 specification [Lie96]. When the L4 specification is referred to, one usually refers to the L4.V2 specification. However, the L4.V2 specification had some problems associated with it. One of the biggest problems was with the thread ids. The thread ids as specified in L4.V2 were found to be rather inflexible and unwieldy. Another problem was that the *clans and chiefs* concept[9] was too inefficient for most purposes. These problems (and more) were addressed in the L4.X0 specification [Lie99]. The main goal of this specification was not to solve all problems of the L4.V2 specification, but to use it as an experimental test-case. The aim of the L4.X2 release [Lie04] though *was* to solve the problems in the L4.V2 specification. Some of the more notable differences between the L4.V2- and L4.X2 specifications are: the separation of task- and thread management, support for multiprocessing and a clear separation between API[10] and ABI[11]. Lastly, Kauer and Völp from the Technische Universität Dresden have developed the L4.Sec specification [BK05], which enhances the L4 specification with security features.

2.2.2 Design

Although the L4 kernel is created with minimalism in mind, there are a couple of basic concepts that have to be in the kernel:

- Address spaces
- Threads
- Inter-process communication

[7] Respestively called AMD Virtualization [AMD05] and Intel Virtualization Technology [Cor05].
[8] As mentioned, a second generation microkernel is designed with both minimalism and performance in mind.
[9] The clans and chiefs concept was designed to enable the implementation of arbitrary security policies.
[10] Application Programming Interface. The API is the source code interface with which a computer program or library lets other programs or libraries use its services.
[11] Application Binary Interface. An ABI allows object code to be run without changes on any system using a compatible ABI.

- Mapping
- Scheduling

Address spaces An address space is a set of mappings from virtual- to physical memory. Address spaces form the mechanism through which task-isolation[12] is achieved. A flexpage is a contigous region of virtual address space.

Threads A thread is the single unit of execution. Each thread belongs to a single address space, with a predefined maximum number of threads per address space. An L4 thread is characterized by a set of registers, with the instruction pointer, stack pointer and state information being required and stored in the kernel in a structure called the thread control block (TCB[13]).

Inter-process communication The sole mechanism through which threads can communicate and exchange data is inter-process communication, which as a side-effect also increases independence between system components. IPC in L4 is always synchronous, which means that no data is exchanged unless both parties agree to. IPC is also unbuffered; the kernel does not temporarily store messages which are to be sent later. L4 offers two different types of IPC: short- and long IPC. Short IPC does not involve access of user space memory and thus cannot generate page-faults; long IPC *does* access user space memory and thus has to take possible page-faults into account. Short IPC only allows a limited amount of data to be be sent, messages exceeding the short IPC limit have to be sent using long IPC. When possible, short IPC is used as it offers significant performance benefits. The IPC mechanism is also used to handle both hardware- and software- interrupts[14] and to map-, grant and unmap flexpages. A timeout can be set to enable (unsuccessful) IPC operations to be cancelled after a specified time.

Mapping When a page-fault is generated in a thread's address space, a notification is sent to the thread's associated pager (which can be different for each thread); that pager can then insert a memory page to resolve the page-fault. Threads can map-, grant- and unmap any of its flexpages to another thread. Mapping flexpages to another thread means that the flexpages will be shared between the mapper and the mappee. Granting a flexpage to another thread passes the control of that flexpage to the grantee, the granter afterwards cannot use the granted flexpage. The unmapping of a flexpage is the inverse operation of mapping a flexpage (it can only be invoked on mapped flexpages, not on granted flexpages). The mechanism through which this is possible is IPC, where a special IPC message is sent between the two threads involved in the memory mapping.

Scheduling The scheduling of threads in L4 is priority-based, which means that the ready thread with the highest priority is always the one to be allocated CPU-time. Each thread has a priority level (of which there are 256) and time quantum associated with it. Scheduling threads with the same priority level is done through a round-robin schedule[15].

The L4 specification also mentions the concept of a *task*, but this is essentially equivalent to the concept of an address space. The main difference between a task and an address space is a conceptual one: the former focuses on the system's memory and the latter focuses on the threads running in an address space. The L4 definition of a task is the set of threads sharing an address space. Creating a new task results in the creation of an address space with one thread in it.

As one of the L4 design goals was to implement no policies, how memory is allocated is left to the application(s) running on L4; the kernel only provides the means to allocate memory. The L4 specification

[12] Task-isolation is the inability of tasks to tamper with the data of other tasks without their consent.
[13] Not to be confused with Trusted Computing Base.
[14] An example of a software interrupt in L4 is a timeout.
[15] Round-robin scheduling lets the first thread of a list of waiting threads use the CPU for a period up to its time quantum. Should the thread not be finished when it's time quantum expires, the thread is moved to the end of the list and scheduling resumes with the first item of the list.

specifies though that the initial address space, called σ_0, should comprise all available memory (excluding kernel-reserved memory).

IPC

As said, IPC communication is always synchronous and unbuffered. Each IPC call therefore involves exactly one sender and one receiver. L4 offers five different IPC calls: *send*, *receive*, *wait*, *reply-wait* and *call*. An invocation of the send call results in the sending of a single message, after which the invoker continues his work. Invoking the receive call results in the invoker waiting to receive a message from *any* sender. The wait call does the same as the receive call, but only accepts messages from a single, specified sender. The last two calls are basically predefined combinations of the aforementioned calls, where each combination reflects a common real-world scenario. The call method results in the sending of a message after which the sender waits for a return message from the receiver it just sent the message to. The reply-wait operation is one in which a single message is sent after which the invoker waits for a reply from any source.

A real-world example in which both the call- and reply-wait operations are used is a webserver. The webserver handles a request from a client, returns the results to the client and is then ready to receive a request from any client; this is equal to the reply-wait operation. From the perspective of the client, a call operation is done as the client subsequently requests data from the webserver and waits for a response from that same webserver.

For performance guarantees, the L4 specification requires that the transitition between the send- and receive states in a single IPC call requires no time[16] (also referred to as an atomic switch). Having an optimized path for frequently used situations clearly benefits the system's real-world performance. Another result of combining the send and receive operations in the call and reply-wait methods is that it saves system calls (without the call method, both a send- *and* receive call would have to be made to achieve the same result). As Liedtke showed in [Lie93], the switching between user- and kernel mode as the result of a system call is very expensive. In both combined calls a single system call is saved, which helps to improve IPC performance.

2.2.3 Implementations

The original L4 implementation by Jochen Liedtke, sometimes referred to as L4/x86, was completely written in assembly language to maximize performance. Writing a program in assembly language has some obvious disadvantages, of which poor readability- and maintainability are probably the most important ones. Therefore, and because of licensing issues, the Technische Universität Dresden developed an L4-based kernel in C++ named Fiasco, which demonstrated that an implementation of the L4 specification in a higher-level language was feasible. Apart from implementing the L4 specification, Fiasco also offers hard real-time support (which is not part of the L4 specification). Similarly, Jochen Liedtke and his team at the University of Karlsruhe developed L4Ka::Hazelnut, once again as proof that an L4 microkernel could be implemented in a higher level language and still offer high performance.

Up until this point, all L4 implementations were inheritly bound to the underlying architecture [Lie95]. This changed with the advent of the L4.X2 specification. One of the first implementations of the L4.X2 specification was also developed at Karlsruhe and was named L4Ka::Pistacchio. The main aim of Pistacchio was on both performance *and* portability. The new portability demand was clearly met by Pistacchio, as the supported platforms included Alpha-, ARM-, IA32-, MIPS- and PowerPC architectures. A relatively new specification and implementation originates from the National ICT Australia group. They have developed an L4 version specifically aimed at embedded systems called NICTA [Aus05].

[16] This also protects the receiver (server) against Denial Of Service attacks.

Implementation	Versions	Architectures
NICTA::Pistachio-embedded	N1, N2	IA-32, ARM, Mips64
L4Ka::Pistacchio	X2	IA-32, IA-64, ARM, AMD64, PowerPC-32, PowerPC-64, Alpha, Mips64
L4Ka::Hazelnut	X0	IA-32, ARM
Fiasco	V2, X2	IA-32, ARM, AMD64, UX
L4/x86	V2, X0	IA-32

Table 2.1: Overview of several L4 implementations

2.3 Fiasco

2.3.1 Introduction

The Fiasco kernel is based on the aforementioned L4 specification. Fiasco is a second generation microkernel, which means that it is created with minimalism (nothing exists in the kernel that cannot be moved out of it) and performance in mind. The effect of minimalism can clearly be seen in the number of lines of code the kernel comprises: around 20.000 lines of code for Fiasco compared to around 3.2 million for the Linux kernel. The Fiasco kernel is developed with real-time features in mind, which means that the system is fully preemptable[17].

2.3.2 History

DROPS[18] [HBB+98] is a research project which aims to find design techniques to implement a distributed, real-time operating system where every component can guarantee a certain level-of-service to applications. The foundation of DROPS is based on the L4 specification. As the original L4/x86 implementation by Jochen Liedtke had some serious disadvantages (readability, maintainability and licensing issues), the Technische Universität Dresden decided to create their own implementation of the L4 specification, called Fiasco, which could be used by DROPS. Besides implementing the L4.V2 and L4.X2 specifications, Fiasco sets itself apart from other L4 implementations with its real-time features, obviously created with DROPS' real-time focus in mind.

Another project related to Fiasco is Fiasco-UX [Sch01], which is a port of the Fiasco microkernel to the Linux system-call interface. This means that Fiasco-UX can be run as an application on a Linux system and due to its special design (no need for kernel-level priviliges) it can even be run as a regular user-level application. One of the main benefits of this approach is its ease of use, particularly when developing applications for Fiasco. Rebooting the machine due to a (kernel) crash is no longer necessary, a simple restart of the Fiasco-UX process suffices. Another advantage is that several instances of Fiasco-UX can run in parallel.

We will now discuss in more detail the features of Fiasco that are relevant to our research.

2.3.3 Threads

Besides implementing the L4 thread specification, Fiasco also implements some additional mechanisms for performance- and real-time support purposes. Some of the performance optimizations stem from [Lie93].

[17] Execution can be interrupted at almost any time, which means the work is temporarily halted in favor of (at that moment) higher priority work. After the prioritized work has finished, the system continues with the interrupted execution.
[18] Dresden Real-time OPerating Systems project.

Scheduling

As the L4 specification dictates, a thread has properties such as a time quantum and priority associated with it; these properties of a thread are called its *scheduling context* and are used in the scheduling of threads. To support the real-time features of Fiasco, each thread can also have an additional real-time scheduling context[19]. An *execution context* is a runnable, schedulable thread. At all times there is only one active execution context (or thread), as Fiasco does not support multi-processor execution (it will only use one of the processors available). A thread's current state of execution is stored in its thread control block (TCB), which resides in the kernel. Switching execution of threads can therefore be done through a simple TCB switch. The scheduler uses the *ready-list* (containing all threads ready to be run) to decide what thread to run next. More details on scheduling in Fiasco can be found in [Ste04].

Ready-list

As said, the system keeps a list of all threads ready to be executed: the ready-list. Although it might seem odd at first, the ready-list can contain threads that are *not* ready to be executed. If the scheduler finds such a thread while traversing the ready-list, it is immediately removed from it. The goal of this *lazy-scheduling* is improving the performance of IPC. As an example, we look at a call to the *call* IPC function, in which a sender sends a message to a receiver and then waits for that receiver to send a message back. If we were to strictly adhere to the property that the ready-list only contains ready threads, this IPC call would result in the following modifications to the ready-list:

- After the sender has sent its message, the sender enters a waiting mode and has to be dequeued.
- After the sender has sent its message, the receiver becomes ready and has to be enqueued.
- After the receiver has sent its message, the receiver enters a waiting mode and has to be dequeued.
- After the receiver has sent its message, the sender becomes ready and has to be enqueued.

We see that this common scenario results in four different changes to the ready-list. Obviously, this is detremental to IPC performance. However, if we allow non-ready threads in the ready-list and exclude the current executing thread from it (which is ready by definition), all four modifications can be saved. After the sender has sent its message, the receiver does not need to be enqueued as it has become the current executing thread. Similarly, once the receiver has sent its message back, the sender does not need to be enqueued as it will be the currently executing thread. Both dequeue actions can be saved as we allow threads in the ready-list that are not ready, which is precisely what entering a waiting mode signifies. To successfully apply this lazy-scheduling, the kernel has to make sure that the current executing thread will be enqueued in the ready-list if it cannot finish its task within its timeslice. If we would leave out this clause, the current executing thread is not guaranteed to get scheduled again (remember that it is not contained in the ready-list while executing), which is necessary in order to finish its task.

To switch execution from sender to receiver (and vice versa) without ready-list scheduling, the system uses their execution contexts. Say we want to switch execution from thread A (which is the current active thread) to thread B, but without using the ready-list. To do so, a simple switch of execution context from A to B suffices, as the current execution context determines what is executed. It is not necessary to also switch the scheduling context from A to B, in fact it is better not to for the following reason. If the scheduling context is not changed when the execution context is switched, thread B gets to execute as if it were A, which means that it can execute for the remainder of A's timeslice. This optimization gives B more execution time than it would normally have, thereby allowing it to finish sooner. In our *call* example, this means that after the sender has sent its message, an execution switch is made to the receiver which gets to execute in the sender's timeslice. Therefore, it is likely that the receiver is able to send a message back sooner, as the sender's timeslice remainder would otherwise be wasted by waiting on the receiver.

[19] This is not part of the L4 specification.

In some situations, it might not be useful to immediately switch to the receiver but instead enqueue the receiver in the ready-list (which is the normal way in which a thread gets to execute). For this situation the *deceit-bit hack* can be used. Originally, the deceit-bit was used in the L4 clans & chiefs concept, but as this concept proved too inflexible it is almost never implemented. Therefore, the deceit-bit was *free* for other purposes and in Fiasco it is used to signal that no lazy-scheduling should be applied.

2.3.4 Synchronization

Having a fully preemptable system requires some form of synchronization for critical parts. Fiasco supports two different types of synchronization: *lock-free-* and *wait-free* synchronization. The main difference between the two synchronization types is in their intended use: lock-free synchronization should be used for time-critical synchronization (such as the synchronizing of frequently accessed, global data) and wait-free synchronization should be used for non time-critical operations (such as the synchronizing of local data). The two synchronization types are implemented as follows:

- Lock-free synchronization: this is achieved through atomic updating of memory[20]. It tries to exchange old data for new data and if this fails it simply retries. In order to prevent the system from retrying infinitely (which would invalidate the real-time properties of the system), a specific retry-count can be set. While setting the data, interrupts will be temporarily disabled to prevent another thread from modifying the data.
- Wait-free synchronization: this is a locking-type of synchronization, in which exclusive access to resources can be obtained by creating a lock. Fiasco expands on this well-known mechanism by introducing *switch locks* (also referred to as *helping locks*). Suppose thread A has locked a resource X which thread B also wants to access. Normally, B would just use up all of its execution time waiting on A to release the lock on X. However, instead of wasting execution time on waiting, B can also help A release its lock sooner by donating its remaining timeslice to A. Now A has more execution time available and will likely release the lock on X sooner, which is exactly what B wants. One might note the similarity of this optimization to the thread switch optimization in the ready-list description.

A full description of the design philosophy and implementation of these two synchronization mechanisms can be found in [HH01].

2.3.5 Locks

Wait-free synchronization in Fiasco can be achieved through the use of locks. Basically there are two types of locks: regular- and switch locks. Of the former there exists just one in Fiasco, namely the CPU lock, which temporarily disables interrupts. This lock should only be used for very short intervals as it can negatively influence the real-time features of the system. The other two locks in Fiasco are switch locks. The first switch lock is the thread lock, which locks access to a thread. The second switch lock is the helping lock, which is similar to the switch lock save from the fact that it also works when the scheduling system has not yet been loaded.

Switch locks are designed explicitly with IPC performance in mind; its basic optimization principle strongly resembles that discussed in the ready-list discussion. The implementation is as follows. When the current thread tries to acquire a lock on a resource by using a switch lock but detects that the resource is already locked, the lock count of the current thread is incremented. After incrementing the lock count, the switch lock keeps on switching to the lock owner (by switching the execution context to that of the lock owner) until the lock is free and can be acquired. Afterwards, when the lock is released, the lock count is decremented and the lock checks if it has been helped by another thread (which donated execution time to the thread holding the lock), if so it switches the execution context to that of the helper.

[20] The x86 processor provides the compare-and-swap (CAS) instruction for exactly this purpose.

When a thread is locked, it will not be selected by the scheduler. Should a switch be made to a locked thread, the system will immediately switch to that thread's lock owner. Although locked threads normally do not get scheduled, an exception arises when a thread is killed. A thread that is to be killed should hold no locks at all, therefore the system keeps scheduling such a thread until it has released all its locks. The thread's lock count is used to determine if any locks are still present.

2.3.6 Timeouts

Fiasco recognizes three different types of timeouts: IPC-, deadline- and timeslice timeouts. An IPC timeout is used to restrict an IPC call to a specific maximum time, which can be used to prevent senders and receivers from waiting endlessly on each other. A deadline timeout is used to set timed deadlines and a timeslice timeout is used to signal the end of a timeslice.

Internally, timeouts are stored in a list that is ordered in ascending timeout order (the head of the list is the oldest timeout and the first to occur). When the system checks if timeouts occured, the timeout list can thus be traversed sequentially. Therefore, detecting timeouts is very fast whereas enqueuing or dequeuing is rather slow. For more details on the implementation of timeouts see [Reu05].

2.3.7 IPC

IPC in Fiasco supports all L4 IPC calls, namely *send*, *receive*, *wait*, *reply-wait* and *call*. All these functions are internally supported through one generic function: *sys_ipc()*, which parameters determine the actual call made. We will now describe the general setup of IPC in Fiasco.

Sender and receiver roles

IPC in Fiasco is synchronous and thus always involves a single sender- and receiver communicating (sending messages) with each other. In Fiasco, there are separate definitions for the sender- and receiver roles. As threads need to both send- *and* receive messages, they extend (inherit from) both the receiver- and the sender role. In Fiasco, only threads extend the receiver role, in other words only threads can receive IPC messages. However, there are two more objects extending the sender role, namely the IRQ and preemption objects. The first translates hardware interrupts into IPC messages, which demonstrates the very generic applicability of Fiasco's receiver/sender setup. The second sends messages dealing with preemptions. As both the IRQ and preemption objects can only send messages (as they do not extend the receiver role), they are also known as *passive senders*.

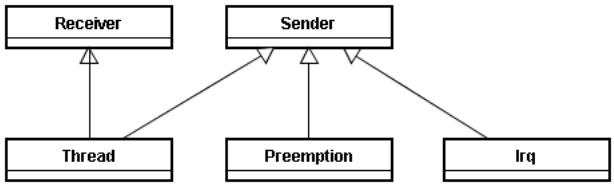

Figure 2.1: Fiasco IPC roles diagram.

Due to IPC's synchronous nature, a receiver can only receive a message from a single sender. As it is possible that a receiver is unable to directly engage in IPC with a sender (for example because it is already engaged in IPC with another sender), the receiver should be able to queue send requests. This is achieved in Fiasco by letting the receiver have a sender list, which is a list of senders wanting to send a message to the receiver. Once a receiver engages in IPC with a sender, that sender is removed from the sender list (if he was it in).

When an IPC call is made, the transferring of a message is handled from the viewpoint of the sender, it is the sender that determines which actions to do first and how to continue. An exception to this situation occurs when the sender is a passive sender; in this case the receiver controls IPC.

IPC overview

Although there are five different IPC calls, internally there is only one function that handles IPC: *sys_ipc()*. IPC (and thus the *sys_ipc()* function) can be divided into a send and receive part. Obviously, the send part handles the sending of an IPC message and the receive part handles the receiving thereof. The table below lists what IPC parts are involved in the five IPC calls:

IPC call	Send part	Receive part
send	x	-
receive	-	x
wait	-	x
reply-wait	x	x
call	x	x

Table 2.2: Mapping of L4 IPC calls to Fiasco send and receive parts.

The following schema gives an overview of how the send and receive parts are handled in Fiasco:

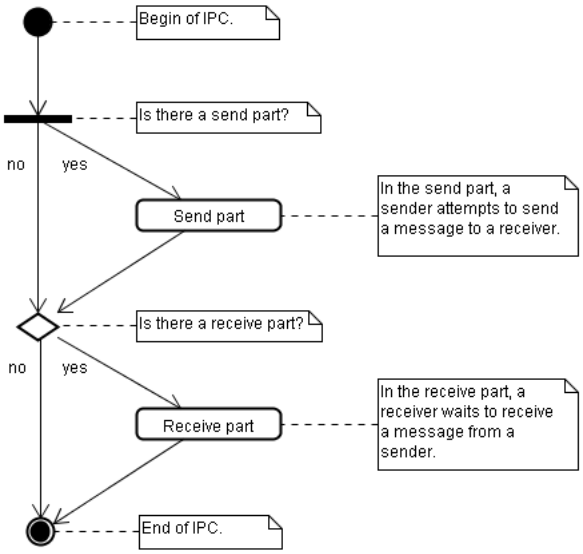

Figure 2.2: IPC overview.

One might remember that in the *reply-wait* and *call* calls, the message sending preceded the receiving of a message. In Fiasco, the receive part is handled *after* the send part to allow the aforementioned two calls to be handled in a single execution of the IPC path (this sequential structure has no influence on the functionality of the other three IPC calls).

Before a message can be sent in the send part, the sender and receiver have to agree upon engaging in IPC; this is referred to as the *handshake*. It is possible that the receiver is not immediately ready to receive a message from the sender, for example because it is still engaged in IPC with another sender; in that case the sender is added to the receiver's sender list and waits for the receiver to become ready. If an error occured in the handshake, IPC is aborted. However, if the handshake was successful, the sender can then send its message to the receiver. The following schema displays this set-up:

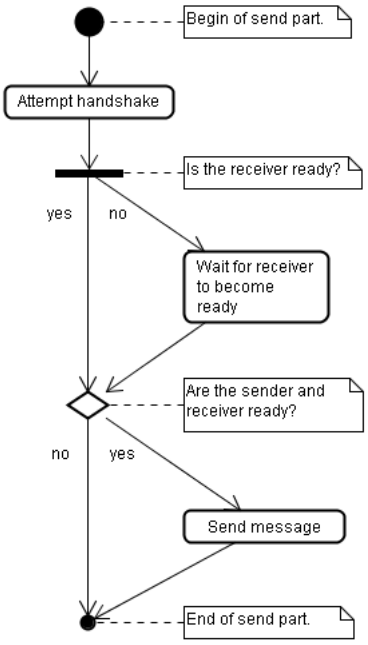

Figure 2.3: IPC send part overview.

As control of IPC is mostly handled by the sender (and thus in the send part), the receive part does relatively little. What *is* done in the receive part is that the receiver enters a loop. With each iteration, it checks if it is ready to receive a message and if there is a sender that wants to send a message (which is indicated by a non-empty sender list). If there is such a sender, the receiver sends it a signal to indicate that it is ready to receive a message. That sender is subsequently removed from the sender list by the receiver. Only when the receiver is not ready to receive a message is the receive part aborted.

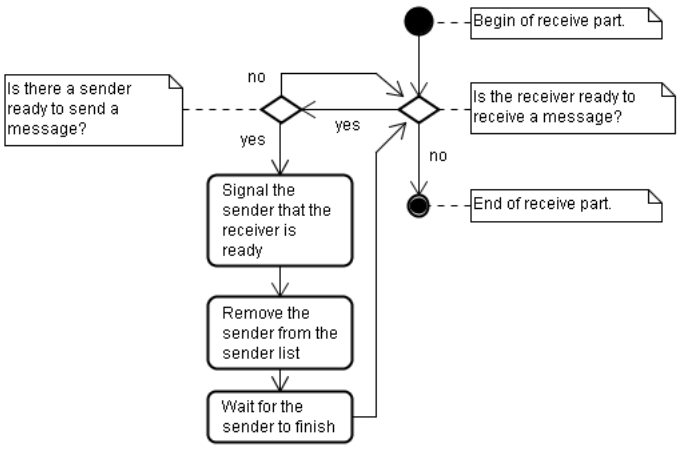

Figure 2.4: IPC receive part overview.

IPC paths

Because IPC in Fiasco is unbuffered, no messages are temporarily stored by the kernel; instead messages are transferred directly between sender and receiver. As described in the L4 specification, Fiasco discerns between three different message types: untyped data (direct strings), memory-buffer references (indirect strings) and memory mappings (flexpages).

There are two paths in Fiasco through which an IPC message can be transferred: a short- and long IPC path.

- Short IPC path: this path uses only the system registers for transferring messages. The size of the data which can be transferred is thus limited to the size of the available registers. The advantages of only using the system registers for message transfer are that it is very fast and that no page-faults can occur (as no user-space memory is accessed). As an example of where the short IPC path is put to good use, when a page-fault IPC message is sent to a pager, that pager should respond with a flexpage IPC message to resolve the page-fault. Because a single flexpage does not exceed the size of the registers, it can be sent using the short IPC path allowing for a fast response by the pager. Secondly, when a direct string is sent, the system puts as much of that direct string into the registers; if the direct string does nog fit completely into the registers, the rest of the direct string is transferred using the long IPC path.
- Long IPC path: all message types can be transferred through the long IPC path, that includes indirect strings. Where the short IPC path limits the sending of flexpages to a single value, the long IPC path can send several flexpages at once. The transferring of messages is done by creating an IPC window. The IPC window is a part of the receiver's address space, which is mapped to the sender's address space during the message transfer. The sender can now directly copy the messages into the address space of the receiver, which prevents the kernel from buffering the messages. The main disadvantage of long IPC is that it has to invoke user-space memory, which could generate page-faults. A generated page-fault is sent to the pager of the receiver, as it is the receiver's address space that is being written to. Also, the long IPC path is slower than the short IPC path.

IPC shortcut path

Besides the two aforementioned IPC paths, there is also an IPC shortcut path. Basically, the IPC shortcut is an alternative version of the short IPC path (it thus also transfers messages through the system registers), but with some further restrictions applied to it (which enable performance optimizations). One of the restrictions of the shortcut is that it does not support the receive- and wait calls. Another restriction is that a single flexpage cannot be sent using the shortcut, only direct strings which fit into the registers are thus eligible for use with the IPC shortcut. Furthermore, the supported timeouts are limited to zero- (immediate response required) and infinite (no time-out at all) timeouts. The IPC shortcut completely runs with interrupts disabled.

The reason why the IPC shortcut was developed is simple: most of the system calls are IPC calls and most of the IPC messages are short IPC messages. Therefore, when the transfer of these short messages is optimized, the performance of the whole system is likely to improve significantly. There are two different versions of the shortcut. The normal version is implemented in C++, but there is also an optimized version of the shortcut, which is written in assembler and has been developed by Michael Peter [Pet02].

IPC states

The state of an IPC operation is stored at the receiver; this state is stored as a bit mask in which each bit corresponds to a specific state the thread can be in (for example waiting for a receiver or sending an IPC message). There are several functions to modify the state, which can add-, delete- or at once add- and delete bits in the state. These functions are respectively *state_add()*, *state_del()* and *state_change()*. All these functions are atomic operations, which means that they are guaranteed

to succeed. The disadvantage of these atomic operations is that they are unnecessarily expensive when interrupts are already disabled. In this case regular operations would have exactly the same result as they too are then guaranteed to succeed, but without incurring the performance penalty of atomic operations. Fiasco therefore offers non-atomic variants of the state modification functions which are suffixed with "_dirty" (leading to the *state_add_dirty()*, *state_del_dirty()* and *state_change_dirty()* functions). Please note that these *dirty* functions assume that interrupts are disabled, they do not check this themselves; it is therefore the responsibility of the caller to make sure interrupts are disabled when calling these functions.

Priority inversion

One of the classic scheduling problems is priority inversion, where a lower priority task is prioritized over a higher priority task. This situation occurs when a lower priority task has locked a shared resource that a higher priority task also wants to access. The higher priority task has no choice but to wait for the lower priority task to release the lock; the priorities are thus temporarily inversed. In Fiasco's old IPC path this problem existed when a combined send- and wait call was made. Let us consider a situation in which a sender A is engaged in a combined send- and wait operation with receiver B. As soon as A has sent its message to B, sender A enters a waiting state and an immediate execution context switch is made to receiver B (this optimization is described in the ready-list section). Because the receiver B gets to execute in the scheduling context of A, it will temporarily inherit the priority of A. Now suppose just before the switch to B, a second sender (which we call C) tries to send a message to B. If the priority of C is less than or equal to the priority of A, there is no problem. However, when C has a higher priority than B, the switch from B to A (which has B's priority) violates the priority-based scheduling invariant[21].

This problem was fixed in the current IPC implementation; its solution was actually quite simple: before switching to the receiver, the sender should check if there is a sender with a higher priority than its own that wants to send a message to the same receiver. Should this be the case, the receiver is added to the ready-list and the system switches to the higher priority sender. If no other higher priority sender exists, the immediate switch to the receiver can be done without any problem.

Real-time features

Because IPC is of vital importance to the performance of Fiasco (in fact, it is of vital performance to *any* microkernel), improving IPC is thus one of the key performance improving strategies. However, with Fiasco a constant eye has to kept at the real-time features of the system. Often, there is a trade-off between performance and real-time support. This can clearly be seen in René Reusner's attempt to improve the performance of IPC in Fiasco, which includes clear graphs detailing the trade-off results [Reu05].

Probably the best example of such a performance versus real-time features trade-off is the decision on which parts of the IPC path should be non-interruptible. If one makes the IPC path non-interruptible, this prevents the use of synchronization mechanisms, which are required in an interruptible path and thus improves performance. The real-time features are invalidated though. The current IPC path is non-interruptible in the following phases: the setup, rendez-vous and finish phase. Furthermore, when sending direct strings in the register transfer (short IPC) phase, the system remains non-interruptible. However, when a flexpage is sent in the register transfer phase or the memory transfer phase is used, the system *is* interruptible. An interruptible path has good real-time features, but this comes at the cost of decreased performance.

Although most phases are non-interruptible, in- and between phases interrupt points and -regions have been inserted to preserve the real-time features. An interrupt region is created by calling *Proc::sti()* (enable interrupts) at the beginning- and *Proc::cli()* (disable interrupts) at the end of the region. An

[21] Which states that higher priority threads should always be scheduled before lower priority threads.

interrupt point is created by calling the *Proc::preemption_point()* function[22]. Please note that there also is a preemption point at the very beginning of an IPC operation, before the *sys_ipc()* function is called. The disadvantage of using interrupt points and -regions is that additional checks are needed because the state might have changed while interrupts were allowed.

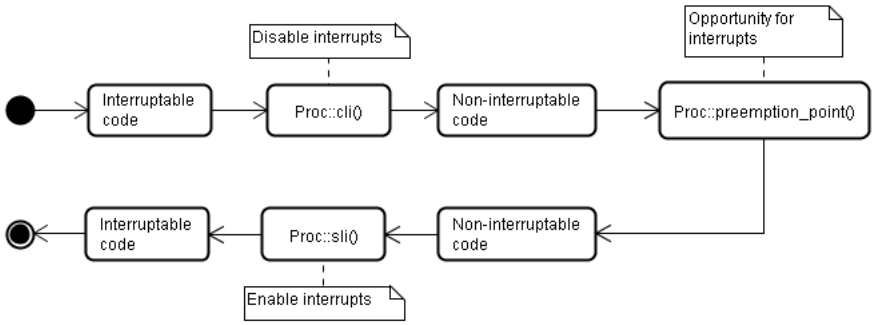

Figure 2.5: Fiasco interrupt transitions.

2.4 PVS

2.4.1 Introduction

PVS stands for Prototype Verification System and has been developed by SRI International, an independent, non-profit research institute [Int06]. The PVS language is a simply typed, higher-order logic language. The language offers built-in types such as booleans and integers, but also supports custom defined types. There are several type-constructors available, ranging from functions to enumerations. PVS also supports (and includes) inductively defined data types such as lists and binary trees. To help in creating proofs, PVS by default supplies the user with many lemmas that can be used in creating a proof[23].

In PVS, specifications can be written just like in any regular functional language. Specifications are however not created to generate an executable, but with the goal of proving properties of those specifications. PVS thus cannot be used directly to create a provably correct program, but it could create a provably correct program specification [MS95]. Having a provably correct program specification is particularly useful when designing a life-critical application[24] such as the software running a space shuttle. In fact, NASA has been using PVS for it's space shuttle program [CV98] and has actively participated in PVS-related research [Vit03].

The PVS theorem prover lies at the heart of every proof in PVS; it has a predefined collection of inference procedures which the user can apply interactively within a sequent calculus framework. Examples of inference procedures are rewriting, simplifying and applying induction. There are also predefined proof strategies, which are essentially just combinations of the inference procedures. PVS also allows the user to create his or her own strategies.

For those unfamiliar with sequent calculus, a short description will now follow. The objective of a proof is to generate a proof tree of *sequents* in which all leaves are trivially true. Each node of a proof tree

[22] Internally, this function first enables interrupts by calling *Proc::sti()*, than executes the *nop()* instruction (which does not do anything) twice to give interrupts the chance and then disables interrupts again by calling *Proc::cli()* after which it returns.

[23] An example is the lemma *reverse_reverse*, which states that reversing a list twice results in the original list.

[24] We define a life-critical application as one where a failure in the application might result in the loss of one or more lives.

is such a sequent. A sequent is a combination of *antecedents* and *consequents*. Sequents are interpreted such that the conjuction of the antecedents implies the disjunction of the consequents. Less formally, the antecedents are what is assumed to be true and the consequents are what one wants to prove to be true.

2.4.2 Features

We will now discuss some of the more noteworthy PVS features which have found their way into our research.

Theory importing

Every declaration (such as lemmas, functions or types) defined in PVS belongs to a single, named theory. For ease of use, it is possible to have one theory import other theories (as long as there are no circular references), which definitions can then be used. This allows one to create modular theories, where general purpose declarations can be defined once and used more than once.

Extensible records

From PVS version 4.0 and up, the concept of extensible records has been introduced into PVS. This concept allows one record to extend another record. This means that when a record extends another record, it automatically contains (or inherits) all fields defined in the record which it extends. This mechanism can be used to define a simple type of inheritance. The example below shows three record types: *record1* is a regular record type, but *record3* inherits from the *record2* by using the extensible records concept. The result of the extension is that the *record1* and *record3* types are equal (they have exactly the same fields), although they have used different methods to achieve this.

```
% record1 does not inherit any fields, it defines them all itself.
record1: TYPE = [# a: type_a, b: type_b #]

% Have record3 inherit all fields from record2, this results in record3 having exactly
% the same fields as record1.
record2: TYPE = [# a: type_a #]
record3: TYPE = record2 WITH [# b: type_b #]
```

Source 2.1: PVS: example of the extensible records mechanism.

2.4.3 Example proof

As a short example of creating a proof in PVS, we will prove that the square of any odd number is also an odd number. This corresponds to the following lemma in PVS:

odd_square_is_odd: LEMMA $\forall$ (number: int): odd?(number) $\implies$ odd?(number $\times$ number)

In this lemma, we use odd?, which is only true if the supplied argument is an odd number in PVS (the definition of odd? can be found in the PVS prelude). When we start proving this lemma, we are initially presented with the following sequent:

```
|
|-------------------------------------------------------------------------------
| {1}   ∀ (number: int): odd?(number) ⟹ odd?(number × number)
```

As can be seen, we now have one consequent (the formula numbered 1, appearing below the line) that corresponds to the statement we want to prove. Currently, this consequent deals with *any* possible number (referred to as a universal quantifier), but we want to limit it to a single instance (or constant) for easier proof creation. To replace the universal quantifier with a constant, we issue the (skolem!) command.

|
|---|
| {1} odd?(number′) ⟹ odd?(number′ × number′) |

The result is now that the universal quantifier has been replaced by a constant. In the rest of our proof, we will use this constant.

For our proof, we need an antecedent stating that number′ is an odd number. At the moment, this assumption is still contained in the formula in consequent 1. To remedy this, we apply the (flatten) command.

{-1} odd?(number′)
{1} odd?(number′ × number′)

The result of the (flatten) command is that the odd number assumption has been removed from consequent 1 and added as antecedent -1.

To be able to continue our proof, we have to look at how odd numbers are defined in PVS. To do so, we expand the definition of odd? by issuing (expand "odd?").

{-1} ∃ j: number′ = $1 + 2 \times j$
{1} ∃ j: number′ × number′ = $1 + 2 \times j$

Once again, quantifiers have been introduced into our proof. As we want to use the constant number′ in consequent 1, we have to make sure there are no quantifiers in antecent -1 (in which the constant number′ is defined). This is done by issuing the (skolem!) command again.

{-1} number′ = $1 + 2 \times j'$
{1} ∃ j: number′ × number′ = $1 + 2 \times j$

The result is that the number′ constant is now removed of its quantifiers, which means that we can use it in consequent 1.

At this point, we have to decide how we can use the odd number constant in antecedent -1 to prove consequent 1. For this, it is best to look at how one would verify this statement mathematically. If we look at consequent 1, we want to find a number j such that:

$$number' * number' = 1 + 2 * j \tag{2.1}$$

If we could find such as number j, $number' * number'$ would conform to the definition of an odd number (which is what we wanted to prove). We can find this number j by rewriting the definition of number′:

$$\begin{aligned} number' &= 1 + 2 * j' \\ number' * number' &= (1 + 2 * j') * (1 + 2 * j') \\ &= 1 + 4j' + 4j'^2 \\ &= 1 + 2 * (2j' + 2j'^2) \end{aligned} \tag{2.2}$$

This simple rewrite has provided us with the value we can use for j:

$$j = 2j' + 2j'^2 \tag{2.3}$$

Returning to our proof, we can use the value of j we found in our proof by instantiating consequent 1. We can now initialize the quantifier in consequent 1 with the value found: $2 \times j' + 2 \times j' \times j'$:

{-1} number′ = $1 + 2 \times j'$
{1} number′ × number′ = $1 + 2 \times 2 \times j' + 2 \times j' \times j'$

Everything is now in place to finish the proof, which can be done by applying (assert!).

This completes the proof of `odd_square_is_odd`.

Q.E.D.

2.5 Related work

Although the Fiasco kernel is in continuous development, there already has been some effort in formally verifying it. In this respect, Fiasco also serves as a test project to study the feasibility of a verified microkernel. The attempts to verify the Fiasco kernel belong to the VFiasco project [HTS03]. Besides describing why they want to verify the Fiasco kernel, Hohmuth and Tews also describe why they decided to verify the C++ Fiasco source and not re-implement the kernel in a 'safe' programming language. It is interesting to see that the approaches taken to verify parts of Fiasco differ on many parts.

2.5.1 Verifying memory management in Fiasco

One of the first attempts to verify the Fiasco kernel focused on Fiasco's memory management [Tew00]. The approach taken by Tews was to create a co-algebraic model of memory management in Fiasco in the CCSL language [HHJT98]. After Fiasco's memory management was modelled in CCSL, this model was compiled to a PVS specification in which properties of the co-algebraic model were proven. The approach taken was successful in formally verifying several properties of Fiasco's memory management. There were however two problems with the approach taken. First of all, CCSL lacks support for imperative programming, which lessens its applicability to real-world situations as imperative programming is widely used. Therefore, creating a model of Fiasco (which was written in an imperative language) was not as intuitive in CCSL as it could be. The second problem was the creation of the CCSL model from the Fiasco source, which had to be done by hand. So far, no conversion can be made from C++ to CCSL, which is mainly due to the complexity of C++. This leaves one no choice but to convert the source code by hand, which can be a daunting task for large programs. A manual conversion also has the disadvantage that, strictly speaking, the properties proven only apply to the CCSL model and not to the actual C++ source. This is because one cannot guarantee that the hand-made C++ to CCSL conversion retains the source code's semantics. As said, defining a formal semantics of C++ is extremely hard due to its complexity. According to the author, this leaves us with two solutions: only verify source code in a language with well-defined semantics or use a well-behaved subset of C++ with a clear semantics.

2.5.2 Developing a C++ semantics compiler

In line with the conclusions of the work described above, development of a C++ semantics compiler was attempted by Matthias Daum in [Dau03]. His compiler uses the earlier work by Hohmuth and Tews in describing the semantics of C++ data types [HT03]. Although the developed C++ compiler partly met its design goals (such as flexibility and architecture independence), there was still a reasonable number of unsupported C++ features. The list of unsupported features included *function pointers*, *enumerations*, *dynamic casts*, *recursive functions* and *access specifiers*. The author deemed some of these features unneccessary because Fiasco didn't use them (such as function pointers and recursive functions) and declared others as rare and usually avoidable (for example unions and bit-fields). The proposed testcase for the compiler was to compile the page-fault handler in Fiasco. Unfortunately, the C++ compiler failed to compile even the relatively small page-fault handler. The inability to compile the page-fault handler indicated that the compiler was not yet ready to be used extensively in Fiasco.

2.5.3 Verifying IPC in Fiasco

Memory management in Fiasco has not been the only component which has been subject to formal verification. The thesis by Endrawaty describes his attempt to formally verify (a part of) inter-process

communication in Fiasco using model checking [End05]. His approach is based on creating a model in the Promela language which could then be verified in SPIN [HPV00]. Promela is a non-deterministic language derived from Dijkstra's guarded commands [Dij75]. Verification in SPIN is basically done by extensively checking the whole state space. To verify a property, the system checks if the property holds in every possible state. This brute-force[25] method has some disadvantages, most notably poor performance and high memory usage because of the extensive state space.

To simplify the creation of the IPC model (and to combat the aforementioned state space problems), the author abstracted away a lot of details and focused on the short IPC path only. Some of the features that were not modelled include *page-faults* and *interrupts*. Furthermore, the author assumed that the timeout was always equal to zero and represented the data copying process with a single boolean value indicating the success of the data copying itself. Even though the model itself was not that big and abstracted away a lot of details, state space problems did occur. Unfortunately, the properties proven by Endrawaty do not exceed that of simple state checks.

Just as in the memory management verification, one of the biggest problems is in translating the C++ source to the language in which the verification will be done (in this case Promela). At first the author tried to create a model by looking at the functionality IPC offers, but later on he just continued with translating the C++ source line-by-line to Promela (albeit in an iterative way). The author concludes by recommending to look into possible optimizations of the model, after which more features could be added (for example by removing some of the abstractions), though he does not give any indication on how to the model might be optimized.

2.5.4 Improved IPC path

After signalling that there were several performance problems with IPC in Fiasco, René Reusner set out to create a new (long) IPC path that solved (several of) these problems [Reu05]. The author chose to optimize the IPC calls most frequently used, namely the *call-* and *reply-wait* calls. As mentioned before, both calls involve a send- and receive operation and switch atomically between these two modes. In the old IPC version almost every part of the call- and wait calls was fully preemptable. In the improved IPC path, most of those parts were *not* preemptable, which saved synchronization costs and thus improved performance. To still be able to guarantee real-time response times, interrupt points were inserted at specific points in which IPC could be interrupted.

The implemented IPC path offered significant improvements over the old IPC path. However, some future work still remained. For one, when sending an IPC message the priority of the sender is always used; it would be more correct to use the maximum of the sender- *and* receiver's priority. Furthermore, the problem of priority inversion still existed, although a reference to a possible solution is given.

2.5.5 L4.verified

Although not directly related to Fiasco, the work in the L4.verified is similar enough to include it here. The goal of the L4.verified project is to provide a mathematical proof of the L4 microkernel specification; the required formal specification was created using the B method [SN01] and is described in [KK06]. Although this approach was fairly successful, the project still changed directions. Work was continued on an expanded L4 specification, which was specifically aimed at developing secure embedded systems. This specification, named seL4 (secure embedded L4), differed from L4 mainly in its use of capabilities, which are the sole providers of access to kernel services. The goal of the seL4 project was to create a formally verified implementation of the seL4 microkernel specification.

The development of the seL4 microkernel was based on incrementally developing a prototype in the Haskell functional language. To test the prototype conventionally, test applications were developed that

[25] A brute-force method implies that it does not use any intelligence to limit the state space, it just tries all possibilities.

ran on the seL4 prototype and simulated regular use thereof. Through these tests and further incremental improvement, the prototype ended up as a complete kernel implementation. At this point, verification of the Haskell kernel implementation was attempted. For this purpose, the Haskell specification was converted to a specification in the Isabelle/HOL proof system [NPW02], which allowed it to be formally verified. The conversion was done manually, which was considered to be faster than developing an application for automatic conversion. After verification of the prototype was successful, the Haskell code was converted to C code for performance purposes. The aim is to also verify this C implementation, for which a formal model of the C language was developed [TKN07].

Currently, the state of the project is that the seL4 kernel is precisely specified in both Haskell and Isabelle/HOL. Around 90 percent of this specification has been verified and its implementation in C is nearing completion. There has not been any large-scale verification attempt of the C implementation, but confidence exists in its feasibility as a case-study has shown that the L4 kernel memory allocator could be verified.

Chapter 3

Model creation

"Simplicity is prerequisite for reliability."

Edsger W. Dijkstra

"Fight Features. ...the only way to make software secure, reliable, and fast is to make it small."

Andrew S. Tanenbaum

3.1 Modeling approach

Our approach in creating the PVS model from Fiasco's C++ source code is explained in detail in this chapter, along with a look at the basics of our PVS model. At the moment, there is no converter which can completely convert the Fiasco C++ source code to our proof language of choice, PVS. We are therefore limited to creating a model of the code. When using a model as the basis for proofs, the obvious disadvantage is that we cannot claim to have proven anything about the code; strictly speaking those proofs only apply to the model and not to the code being modelled. However, if one trusts the model to be an accurate representation of the code (albeit probably on a higher level), any proof claims of the model would also apply to the code.

There are two ways to create a model of source code: create a one-on-one translation or create a higher-level model (which abstracts away some of the details in the code). The main advantage of the first method is that it can help increase confidence that the model is an accurate representation of the source code. This is due to the fact that each line in the source code can be directly linked to a part of the model. The disadvantage is that such a detailed model can quickly become very large, which makes proofs more difficult to establish and can also make the creation of the model a tedious process. The second method looks at the source code at a higher level, focusing more on *what* the source code does than *how*. The disadvantage here is that one has to have a good understanding of what the system does, which can be more difficult than it sounds. One of its advantages is that the model can choose to focus on a specific part of the source code, leaving out uninteresting or irrelevant parts. Therefore, the model will be more compact, making it easier to comprehend and proofs probably smaller.

3.2 PVS model

We will now discuss how the PVS model looks like, from the datatypes used to the abstractions applied. Furthermore, we will show how the original C++ source code was converted to PVS and what problems

occured. Where possible, we will use excerpts of the model to help clarify the text. While discussing our model, it is important that one keeps in mind that our verification attempts only look at IPC from the perspective of the current thread.

3.2.1 Properties

As our model has been created with the properties we wanted to prove in mind, we will first discuss those properties. We have looked at three different properties, each of which we will now discuss in more detail.

Property 1: removal of sender's thread lock on receiver

Before a sender can send a message to a receiver, they both have to agree upon engaging in IPC (referred to as the *handshake*). The message can only be sent if the handshake is successful. When sending a message, the sender has to make sure that the receiver thread is not locked by another thread. If it is, the sender will acquire a thread lock on the receiver before sending the message. After the sending has finished, any acquired thread lock on the receiver should be released as otherwise no other threads can access the receiver subsequently.

Property 2: waking up receiver in combined send/receive

As explained earlier, it is possible to make a combined send- and receive IPC call. This means that after having sent its message to the receiver, the sender then waits for the receiver to send a message back. Essentially this means that after the first message is sent, the sender- and receiver roles are swapped. At the moment where the receiver assumes the sender role, it should be ready to be scheduled (in other words, the receiver is awoken). If the receiver is not ready to be scheduled when the roles are swapped, it might never be scheduled again, which might result in the sender waiting forever to receive a message back.

Property 3: validation of assertions in the code

In the Fiasco C++ source code, there are several calls to the *assert()* function. Basically, a call to *assert()* is meant to ensure that the system is in a correct state. The *assert()* function takes a boolean expression as its parameter, which it evaluates. If the expression evaluates to true, execution is continued normally. However, if the expression evaluates to false, execution of the program is aborted. Assertions should thus only be used when a specific condition *has* to hold. As we have included the assertions in our model, we tried to verify that all assert calls evaluate to true. If we could verify this, that would give us some confidence in the correctness of our model.

3.2.2 Abstractions

To prevent the model and proofs from becoming too complex, we abstracted away many parts of Fiasco IPC. The abstractions depend in part on the above-mentioned properties, but also on the complexity of some parts of IPC. We will now list the most important abstractions:

- **Implementation of functions**: in some cases we kept the model simple by hiding the implementation of specific functions. In these cases we were only interested in *what* the effect of a function was, not on *how* it was achieved.
- **Different senders**: of the three possible IPC message senders, we have only modelled one: threads. The preemption- and interrupt IPC message senders have been abstracted away as inclusion would have required two additional send paths to be added to the model.
- **Scheduling**: we are not interested in *how* and *which* threads are scheduled, our only concern is with the effect this scheduling has on IPC. Anything related to scheduling, such as the execution- and scheduling context concepts, is thus not included in our model.

- **Receive part**: the properties we looked at mostly concerned the send part of IPC, which resulted in the abstraction of the receive part. The receive part does appear in a slightly modified form in our preemption point implementation.
- **Long IPC**: in contrast to the short IPC path, the long IPC path is fully interruptible. This greatly increases the complexity, as the system can interrupt ongoing IPC at any time. Furthermore, in long IPC page-faults can occur which require special handling. The long IPC path is thus far more complex than the short IPC path, which would lead to a more complex model and (much) harder proofs, and has therefore been abstracted away.
- **IPC shortcut**: as the IPC shortcut path has a fairly simple implementation, it was not likely to be an interesting subject for formal verification. Furthermore, the shortcut completely runs with interrupts disabled, which is precisely one of the features we were interested in modeling.
- **Short flexpage**: the transfer of a single, short flexpage is done by using the system registers. However, it is handled differently from other short IPC messages that are also transferred through the system registers. Furthermore, when memory is being mapped, interrupts have to be enabled which makes the model and proof harder to create. We have therefore omitted the mapping of a single, short flexpage.

3.2.3 Theory structure

As we tried to create a modular structure for our model and proofs, we split our PVS sources into various theories. The figure below shows the theories and their dependencies:

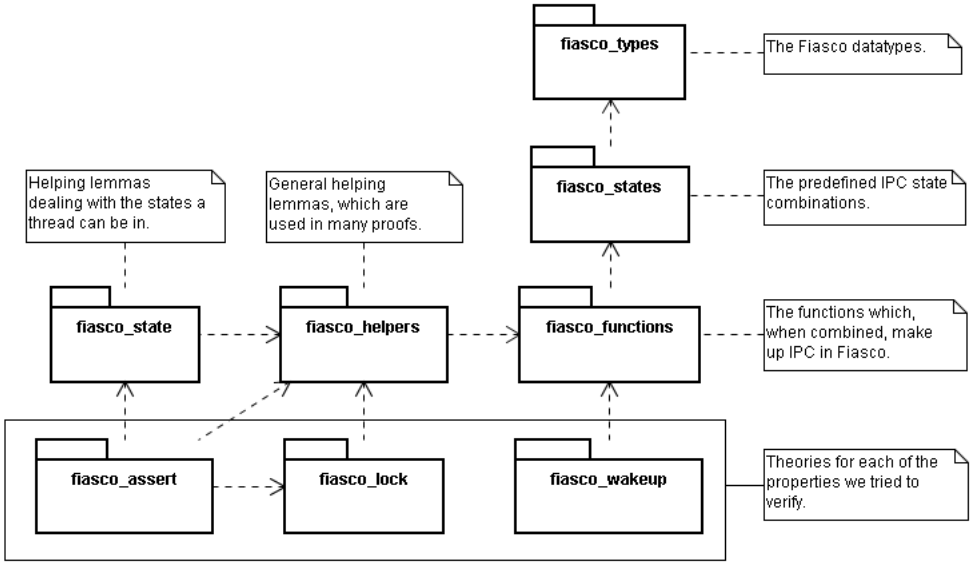

Figure 3.1: PVS theories structure.

The basis is formed by the <fiasco_types> theory, in which the Fiasco-related datatypes (such as threads and timeouts) are defined. These types are used by the <fiasco_functions>- and <fiasco_states> theories, where the IPC-related functions and predefined thread states are defined respectively. Having setup the system this way prevents the different theories from redefining these essential IPC components (thus creating a single point of definition). Besides the three theories for our proof properties (<fiasco_assert>, <fiasco_lock> and <fiasco_wakeup>), there are two theories which define lemmas that are required to verify the properties, but that are not directly related to these properties. The first of these theories is the <fiasco_helpers> theory in which general purpose lemmas are declared. The second assisting theory, <fiasco_state>, defines more specific helping lemmas that only involve the state a thread can be in.

3.2.4 Fiasco types

To model the IPC path we have to model the types involved. Because not all details of the types defined in Fiasco are relevant to the properties we want to prove, we do not have to create a one-on-one model. In short, we will only model those types that are directly relevant to the properties we want to prove and on a level sufficiently high enough to be suitable for a proof properties. This means that many Fiasco types will have quite abstract definitions in our models. We will now discuss the types in our PVS model.

Sender list

When a sender attempts to send a message to a receiver that is currently busy, that sender is added to the receiver's sender list (which contains all senders wanting to send a message to the receiver). Once the receiver has finished IPC with the current sender, it pops the first sender off the sender list and engages in IPC with that sender (if that sender is still willing to do so).

When modeling the sender list, one should note that we abstracted away the receive part as we focused on the send part. Therefore, we were only interested in the sender list from the viewpoint of the sender. Assuming this viewpoint, the status of a receiver's sender list can be one the following options:

- The sender is the first item in the sender list and will thus be the next sender with whom the receiver will engage in IPC.
- The sender is in the receiver's sender list but not as the first item; that means it will have to wait for the receiver to finish IPC with the preceding senders before IPC can be engaged upon.
- The sender is not enqueued in the sender list at all, which means that it is either currently engaged in IPC with the receiver or not interested in sending a message to the receiver.

These options make it clear that the original list-like implementation of the sender list is slightly more complex than it has to be, therefore we modelled the sender list as the following, simple enumeration:

```
Sender_list: TYPE = {
                      First,    %<this> is the first sender in the list.
                      Enqueued, %<this> is enqueued but not the first item.
                      Dequeued  %<this> is not in the non-empty list.
                    }
```

Source 3.1: PVS: sender list definition.

In our model thus every receiver's sender list is specifically targeted to the current active thread.

Thread state

Because IPC in Fiasco is very dependent on the states the sender- and receiver are in, the thread state enumeration is one of the key datatypes in our model. In Fiasco, the thread state is stored as a bit mask[1] of all possible states a thread can be in (such as *ready* or *dead*). In PVS we modelled the thread state as a record of boolean values, each one representing a specific state bit being set or not. We could have modelled the bit mask as a PVS bitvector [BMS+97], which would represent the actual implementation more faithfully, but this would have needlessly complicated the model and proof. As we are only interested in which states a thread is in, our simple record of boolean values suffices.

```
Thread_state: TYPE = [# thread_ready:    bool,
                        thread_cancel:   bool,
                        thread_dead:     bool,
                        thread_busy:     bool,
                        thread_invalid:  bool,
                        thread_polling:  bool,
                        thread_receiving: bool,
```

[1] A bit mask is an integer where individual bits are used to store data in.

```
                        thread_ipc_in_progress: bool,
                        thread_send_in_progress: bool,
                        thread_transfer_in_progress: bool #]
```

Source 3.2: PVS: thread state definition.

When one compares the definition above with that in the Fiasco source code, one may note that there are some state bits missing in our definition. This is due to the properties we tried to verify, our set of state bits forms the minimal subset of states necessary to create a model in which our properties could be verified. For example, the *thread_polling_long* bit is not included in our definition, as it is only used in long IPC which we have not modelled. The *thread_polling* bit is an example of the importance of the thread state in IPC, as it indicates if a sender is waiting for a receiver to become ready.

In Fiasco IPC, there are several predefined states that have specific bits toggled on or off. The main function of these states is to prevent the same states from being redefined each time they are used. An example of such a predefined state is the *Thread_ipc_sending_mask* state, which indicates the state a sender is in when it waits for a receiver to become ready; this state is defined as follows:

```
#define Thread_ipc_sending_mask         (Thread_send_in_progress        | \
                                         Thread_polling                 | \
                                         Thread_polling_long)
```

Source 3.3: C++: *Thread_ipc_sending_mask* state definition.

It is important to note that in the definition above only the three specified bits are sets, the rest is unset. To easily achieve the same effect in our model, we created a basic state, *TS_empty*, which has all bits unset. Extending this *empty* state allowed us to easily create predefined states such as the one defined above. Our definition of the *Thread_ipc_sending_mask* state is the following:

```
TS_ipc_sending_mask: Thread_state = TS_empty WITH [
                                      thread_send_in_progress := true,
                                      thread_polling := true
                                    ]
```

Source 3.4: PVS: *Thread_ipc_sending_mask* state definition.

We have used a slightly different naming convention, where the *TS*-prefix indicates a thread state definition, for brevity purposes. The *thread_polling_long* bit is not included in our predefined state, because it is not part of our model (as explained in more detail earlier).

Sender and receiver

When modeling threads, we note that a thread can act as both a sender and a receiver. As threads, senders and receivers are objects containing fields, we naturally defined them as records in our model. For threads to act as a receiver, in Fiasco a thread is an extension of a receiver (it inherits from it). For extension (or inheritance) of records, we used the extensible records mechanism.

In Fiasco, a receiver extends a context, which contains many scheduling-related fields (such as the scheduling context). However, as we have abstracted away scheduling, these fields have not made it in our model. The *state* field is not removed from the context though, as is is integral to IPC in Fiasco. A receiver extends a context with two fields of its own. The first of these fields is the *partner* field, which indicates the partner (or sender) thread the receiver is engaged in IPC with. The second field, named *sender_list*, contains the receiver's sender list, which was discussed earlier.

```
% The context contains the thread state, here a lot of details have been
% abstracted away which mostly concern scheduling.
Context: TYPE = [# state: Thread_state #]

% The receiver role inherits from <Context> and adds two fields: <partner> and
% <sender_list>. The first reflects the fact that a receiver always has a
% partner associated with it and the second reflects the threads that want to
```

```
% engage in IPC with the receiver.
Receiver: TYPE = Context WITH [# partner: Thread_pointer,
                                 sender_list: Sender_list #]

% The sender role in our model does not have any real function, but we have
% included it for use in possible later extensions of our model.
Sender: TYPE = [# #]
```

Source 3.5: PVS: context, sender and receiver definitions.

Although it might seem odd to include a definition of an empty sender record type, we have included it for use in possible later extensions of our model.

Thread

Having modelled the two roles which form the basis of threads, we are now ready to define the threads themselves. Besides the fields from the sender and receiver roles, threads have one additional field, named *thread_lock*. The function of this field is to model thread locks. This means that every thread can be locked by exactly one other thread. As long as a thread is locked, only the thread owning the lock can access the locked thread. The thread lock itself is defined in Fiasco as a pointer to the thread holding the lock; in our model we have followed this design.

```
% The actual definition of the thread type, which inherits from both the sender
% and the receiver types.
Thread: TYPE = Sender WITH Receiver WITH [# thread_lock: Thread_pointer #]
```

Source 3.6: PVS: thread definition.

To clarify what the result is of using extensible records, we have created an alternative definition of the thread type below, which does not rely on extensible records. The alternative definition results in the same record being created, as both have exactly the same fields.

```
% This alternative thread type definition includes the sender's and receiver's
% fields directly in the definition, it does not use the extensible records
% mechanism, but still has the exact same fields.
Thread: TYPE = [# state:    Thread_state,
                  partner:  Thread_pointer,
                  sender_list: Sender_list,
                  thread_lock: Thread_pointer #]
```

Source 3.7: PVS: thread alternative definition.

For the thread pointer type, we have followed the C++ implementation where pointers are just integers (referring to objects). As explained, the thread lock was modelled as a pointer to the thread owning the lock. However, it is also possible that there is no thread lock on a thread. In Fiasco, this is implemented by having the thread lock being equal to the NULL-pointer. We modelled the NULL-pointer as a thread pointer instance without a specific value. One might expect the zero thread to have the value zero (corresponding to the C++ implementation), but then changing the type of a thread pointer to a positive number would also require the definition of the zero thread to be changed (which would not be necessary with our definition).

```
% A pointer to a thread is just an integer, this reflects a C++ pointer also
% being an integer.
Thread_pointer: TYPE = int

% Define the zero thread pointer, which represents the C++ NULL pointer.
Zero_thread: Thread_pointer
```

Source 3.8: PVS: thread pointer definition.

Threads

In Fiasco, at any single time there can (and probably will be) more than one created thread. For each created thread, there exists a pointer to access that thread. There is also a subset of threads, called the ready-list, which contains all threads ready to be scheduled (with some exceptions, see the backgrounds on Fiasco). However, as scheduling has not been incorporated in our model, the ready-list has also been abstracted away. The only thing for us to model is thus the list of created threads, which we have defined as a function from thread pointers to threads.

```
% The thread list is a function which translates thread pointers to threads;
% this reflects the C++ concept of pointers to objects (in this case pointers
% to threads).
Thread_list: TYPE = [Thread_pointer -> Thread]
```

Source 3.9: PVS: definition of thread list.

Other types

Many functions involved in the IPC path return an IPC error code. Although there are many possible error codes that can be returned, the source code only checks *if* an error occured and is not interested in the actual error code itself[2]. We have therefore chosen to model the error codes as a boolean value which indicates whether an error occured or not. A similar reasoning can be applied to IPC timeouts. We are only interested *if* a timeout occured and not *how*. Timeouts are thus also a defined as a boolean value.

```
% Most functions in Fiasco return an error code, of which there are many.
% However, we are only interested in whether an error occured and thus the
% error code is modelled as a boolean value.
Ipc_err: TYPE = bool

% Indicates if a timeout occured, additional details of timeouts such as
% the exact time at which they expire are not of interest to us.
L4_timeout: TYPE = bool
```

Source 3.10: PVS: IPC error codes- and timeout definitions.

3.2.5 Fiasco functions

Converting from sequential- to functional code

One of the key design issues was how to model the execution of imperative C++ source code in the functional language used by PVS. Because our model should represent the Fiasco C++ source, we have tried to remain very close to the source code in our conversion to PVS. This resulted in our PVS model quite often directly reflecting the C++ source, which allows anyone familiar with the Fiasco sources to quickly grasp the inner workings of our model. Another benefit of this strict adherence to the source code, is an added confidence in our model as one can easily check if the model accurately reflects the source code. To create this fairly direct conversion, we often used the *LET* construction, which can be used to update data without transferring control to another function. Conveniently, most flow-control structures (such as *if*-statements) could be directly translated to PVS.

Functions defined in C++ can have *side-effects*, which means that functions can modify some global state besides modifying their return values. We therefore introduced the concept of a system state in our model, which each function should be able to access and modify. This is achieved by having each function take the system state as one of its input parameters and also have it return the system state. An example of this method is shown below, where the *sys_thread_ex_regs()* function takes a parameter of the type *System_state* (which definition will be discussed shortly) and returns the original system state, but with the *thread_cancel* bit of the thread pointed to by *tp* set to true.

[2]The reason why there are still many different error codes is that the caller of IPC might be interested in what went wrong.

```
sys_thread_ex_regs(tp: Thread_pointer,state_old: System_state): System_state =
  state_old WITH [(threads)(tp)`state`thread_cancel := true]
```

Source 3.11: PVS: an example of a system state-modifying function.

System state

Now that we have a way to simulate the updating of the system state, we have to look at what fields the system state should contain. As the list of created threads should be accessible at all times, it has to be included into the system state. Furthermore, in our proofs we will be looking at IPC from the viewpoint of a single thread, in our case the active, running thread[3], which we will refer to as the *this* thread (with a clear analogy to the corresponding C++ term). A better name for the *this* field would have been *this_pointer*, but we favored the shorter version for brevity purposes. This leads to the following, preliminary system state:

```
System_state: TYPE =
            [#
              this:    Thread_pointer, % Pointer to active thread.
              threads: Thread_list     % List of all threads.
            #]
```

Source 3.12: PVS: incomplete system state, focusing on threads.

It is now possible to access the *this* thread by providing it as a parameter to the *threads* field (which is actually a function from thread pointers to threads). In our model, most proofs will involve the *this* thread as that is the thread from which viewpoint we look at IPC.

```
% Return the this thread, which the <this> field of the system state points to.
get_this_thread(state: System_state): Thread =
  state`threads(state`this)
```

Source 3.13: PVS: example of accessing this thread.

There is a problem with the current system state definition though. As it is defined now, it is possible that the current, running thread is the zero thread (pointed to by the NULL-pointer), which is obviously incorrect. To correct this flaw, the type of the *this* field was changed to the *This_thread_pointer* type, which is defined as the subset of thread pointers not equal to the zero thread:

```
% The pointer to the current active thread (the <this> thread) must never be
% the <Zero_thread> (as it points to nothing) and thus we have defined
% a special type of thread pointer for the <this> thread which excludes the
% thread pointer being equal to the <Zero_thread>.
This_thread_pointer: TYPE = { tp: Thread_pointer | NOT tp = Zero_thread }
```

Source 3.14: PVS: definition of special this pointer type.

Errors and timeouts

One of the problems we encountered when converting from C++ to our model, were the varying return values of the functions involved in IPC. Each IPC-related function in Fiasco had one of the following three return types: *void*[4], *bool* or *Ipc_err*. A closer inspection revealed that all bool-returning functions did not have any side-effects (with one exception discussed later), whereas the other functions did. As we explained earlier, a state-modifying function (one with side-effects) must at least return the updated system state. As the functions returning an *Ipc_err* value also modified the system state, those functions should return the updated system state, but they also had to return the error value. One option was to have each function return a tuple of a system state and an error value. However, as there are only

[3] In Fiasco there can only be one thread running at any single moment in time, support for multi-processor systems has not yet been integrated into the kernel. Consequently, in a multi-processor system, Fiasco will only use one of those processors.

[4] Which indicates no value is returned.

a couple of functions modifying the error value, we opted for another solution: integration of the error code into the system state:

```
System_state: TYPE =
            [#
              this:    This_thread_pointer, % Pointer to active thread.
              threads: Thread_list,         % List of all threads.
              error:   Ipc_err              % Indicates if error occurred.
            #]
```

Source 3.15: PVS: incomplete system state, with integrated error field.

The initial system state has the *error* field set to false, which makes sense as no error can have occured *before* IPC has even been started. While engaged in IPC, this field can be set by functions to indicate if an error occured; this value can subsequently be checked by the caller of the function to see if an error was returned. Most often it was not necessary for the caller of the error-returning function to reset the *error* field to false after an error was returned, as an error often resulted in IPC being prematurely aborted. An example of this is shown in the code excerpt below, where the *do_ipc_send_part()* function calls *try_handshake_receiver()*. The latter sets the *error* field to true when the partner is invalid and then immediately returns; the *do_ipc_send_part()* function then checks if the *error* field has been set to true and acts accordingly (by immediately returning from the function):

```
% Try to have the sender and receiver agree upon engaging in IPC.
try_handshake_receiver(partner: Thread_pointer,
                       state_old: System_state): System_state =
  % Immediately return with an error if the partner is invalid.
  IF partner = Zero_thread OR [...] THEN
    state_old WITH [error := true]
  ELSE
    [...]
  ENDIF

% Handle the send part in the do_ipc() function.
do_ipc_send_part(partner: Thread_pointer, have_receive: bool,
                 state_old: System_state): System_state =
  % Start with a handshake and immediately return if an error occured.
  LET state_temp = try_handshake_receiver(partner,state_old) IN
  IF state_temp`error THEN
    state_temp
  ELSE
    [...]
  ENDIF
```

Source 3.16: PVS: example of error setting and checking.

Following a similar line of reasoning, we integrated a *timeout* field into the system state, which indicates if a timeout has occured. Once again, this field is initially set to false and only becomes true when a timeout occured. Our basic system state is thus defined as follows:

```
System_state: TYPE =
            [#
              this:    This_thread_pointer, % Pointer to active thread.
              threads: Thread_list,         % List of all threads.
              error:   Ipc_err,             % Indicates if error occurred.
              timeout: L4_timeout           % Indicates if timeout occurred.
            #]
```

Source 3.17: PVS: basic system state, with integrated *timeout* field.

To ensure that initially the *error*- and *timeout* fields are set to false, both fields are explicitly unset at the start of the IPC path (which is the *sys_ipc()* function). Where possible though, we have not made any assumptions about the initial values of fields or parameters to keep the model as generic as possible:

```
sys_ipc(have_send: bool, partner: Thread_pointer,
        have_receive: bool, sender: Thread_pointer,
        state_old: System_state): System_state =
```

```
LET state_temp = state_old WITH [(error)   := false,
                                 (timeout) := false]
IN
  % Make sure there is either a send- or receive part, if so start IPC.
  IF have_send OR have_receive THEN
    do_ipc(have_send,partner,have_receive,sender,state_temp)
  ELSE
    state_temp WITH [(error) := true]
  ENDIF
```

Source 3.18: PVS: system state initialization in the *sys_ipc()* function.

One can cleary see that IPC is only started when there is a send and/or receive part (which makes sense as otherwise there is nothing to do).

Increment model development

As said, we have developed our model incrementally. We will illustrate this with an example. Initially, we defined the *sender_ok()* function as follows:

```
% Indicates if a partner is ready for a sender.
sender_ok(sender: Thread_pointer, receiver: Thread_pointer,
          state_old: System_state): bool
```

Source 3.19: PVS: *sender_ok()* function without implementation.

This function is defined without an implementation, which means that PVS cannot determine the value returned by the function when it is called; any use of this function in a proof therefore requires all possible return values (in this case only two: *true* or *false*) to be checked. Using unimplemented functions, we were able to rapidly develop a very crude model. After we were satisfied with the general structure of the model, we added implementations to the unimplemented functions to have them better reflect the source code. The definition of the *sender_ok()* function *with* implementation is listed below.

```
% Indicates if a partner is ready for a sender.
sender_ok(sender: Thread_pointer, receiver: Thread_pointer,
          state_old: System_state): bool =
  % The receiver should at least be in a receiving IPC state.
  IF NOT state_old`threads(receiver)`state`thread_receiving OR
     NOT state_old`threads(receiver)`state`thread_ipc_in_progress
  THEN
    false
  % Open wait: no partner should be specified and the sender should be the
  % first in the receiver's sender list.
  ELSIF state_old`threads(receiver)`partner = Zero_thread AND
        state_old`threads(receiver)`sender_list = First THEN
    true
  % Closed wait: check if the sender is the same as the partner specified.
  ELSIF sender = state_old`threads(receiver)`partner THEN
    true
  ELSE
    false
  ENDIF
```

Source 3.20: PVS: *sender_ok()* function.

We have already discussed shortly how we converted from the Fiasco C++ source to PVS. To show an example of such a conversion, we have included the *sender_ok()* function as defined in Fiasco, which can be compared to our PVS definition listed above.

```
bool Receiver::sender_ok (const Sender *sender) const
{
  unsigned ipc_state = state() & (Thread_receiving |
                                  //                Thread_send_in_progress |
                                  Thread_ipc_in_progress);

  // If Thread_send_in_progress is still set, we're still in the send phase
```

```
  if (EXPECT_FALSE (ipc_state != (Thread_receiving | Thread_ipc_in_progress)))
    return false;

  // Check open wait; test if this sender is really the first in queue
  if (EXPECT_TRUE(!partner()
                  && (!_sender_first || sender == _sender_first)))
    return true;

  // Check closed wait; test if this sender is really who we specified
  if (EXPECT_TRUE (sender == partner()))
    return true;

  return false;
}
```

Source 3.21: C++: *sender_ok()* function.

3.2.6 Splitting functions

Three of Fiasco's functions have been split into several functions in our PVS model, namely the *do_ipc()*, *do_send_wait()* and *ipc_receiver_ready()* functions. Although the splitting is not in accordance with our aim to create a (mostly) one-on-one conversion of the source code, for each split there is a good reason. The reason why the *do_ipc()* function was split in two parts was to separate the send part from the receive part. This enabled us to do proofs for only one of the two parts, thus creating more compact and modular proofs (a change in the receive part would not affect proofs of the send part[5]). The main incentive to split the *do_send_wait()* function was its considerable size. Splitting once again resulted in more modular and compact proofs. The last function, *ipc_receiver_ready()*, was split for a totally different reason. In our model, we differentiate between functions that modify the system state and those that do not. The former always return the system state, whereas the latter return anything but the system state (for example boolean values). The problem with the *ipc_receiver_ready()* function was that it had to both modify the system state and return a boolean value. Therefore, the function was split into a state-modifying part (the *ipc_receiver_ready_change()* function) and a boolean-returning part (*ipc_receiver_ready()*). The state-modification only occurs when the *ipc_receiver_ready()* function returns true, so in our model a return value of true results in calling the *ipc_receiver_ready_change()*.

```
% Check if the sender is ready for the receiver.
IF ipc_receiver_ready(sender,receiver,state_old)
THEN
  % Apply the changes caused by the side-effects of the
  % ipc_receiver_ready() function.
  ipc_receiver_ready_change(sender,receiver,state_old)
ELSE
  state_old
ENDIF
```

Source 3.22: PVS: calling the split *ipc_receiver_ready()* function.

An overview of the functions split and the functions into which they were split is listed below:

Original function	**Split into**
do_ipc()	*do_ipc_send_part()*, *do_ipc_receive_part()*
do_send_wait()	*do_send_wait()*, *do_send_wait_loop()*, *do_send_wait_finish()*
ipc_receiver_ready()	*ipc_receiver_ready()*, *ipc_receiver_ready_change()*

Table 3.1: Model: split functions.

[5]This is due to the fact that the receive part is always executed *after* the send part and thus cannot influence it. The reverse does not automatically hold, as the receive part *can* be preceded by a send part which could influence its begin state.

3.2.7 Preemption points

As explained in detail earlier, in Fiasco a preemption point is a function that temporarily allows other threads to be scheduled and executed in favor of the current executing thread. When another thread indeed gets scheduled, its execution might change the system state. In our model a preemption point is thus a state-modifying function. Because we have abstracted away scheduling, we have to simulate the result of the scheduling on the system state. Our focus is thus on *what* the result of a preemption point on the system state is, not on *how* this result is achieved. In our model, we discern between five different actions that can occur in a preemption point:

- Nothing happens.
- The partner thread is killed (through the *kill()* function).
- An IPC timeout occurs.
- The ongoing IPC is cancelled (through the *sys_thread_ex_regs()* function).
- The receiver becomes ready to receive a message from the sender.

These five actions are not complete though, one of the most striking omissions is a concurrent sender wanting to send a message to the same receiver. We have modelled only the five actions listed above to keep the model compact, more actions would further complicate the model and proofs..

Even though the first of the five actions is the most likely to occur, we will model the actions as if they all have the same probability. This simplified version of the real situation does not have any negative effect on the validity of the model, all possible actions have to be examined in a proof anyway. The possible preemption actions are defined as the enumeration type *Preemption_action*. Although the enumeration might suggest otherwise, a single preemption point might in fact execute several of its possible actions. Therefore, we have also created the *Preemption_actions* type, which is just a list of preemption actions.

```
% Enumeration of the possible preemption actions.
Preemption_action: TYPE = {
                            Nothing,             % Nothing happens.
                            Kill,                % The partner is killed.
                            Timeout,             % A timeout occurs.
                            Sys_thread_ex_regs,  % IPC is cancelled.
                            Receiver_ready       % The receiver becomes ready.
                          }

Preemption_actions: TYPE = list[Preemption_action]
```

Source 3.23: PVS: *preemption_action* type.

Randomness

When executing a preemption point, one never knows beforehand what actions it will execute, its behaviour appears to be random. One of our main problems was how to model this randomness. Although PVS has the *choose()* function, which randomly chooses an element from a set, we could not use it in our model. The problem with this function is due to the very definition of a function: equal input results in equal output. In our case, its input would always be equal to the set of possible combinations of preemption actions (its powerset). Because there are several calls to the preemption point function, each of which has to use the same *choose()* function call, the returned actions would in each case be the same. To circumvent this, we created an uninterpreted function, named *preemption_action()*, which returns a list of preemption actions. As the function contains no implementation, the output is essentially random. To prevent the problem with equal inputs, the *preemption_action()* function accepts a single input parameter, which we make sure is different with each call.

```
% Return a (random) preemption action.
preemption_action(n: nat): Preemption_actions
```

Source 3.24: PVS: *preemption_action()* function.

To allow the seed input parameter[6] to be different with each call, a *seed* field was added to the system state:

```
% Modified system state with added <seed> field.
System_state: TYPE =
            [#
              this:    This_thread_pointer, % Pointer to active thread.
              threads: Thread_list,         % List of all threads.
              error:   Ipc_err,             % Indicates if error occurred.
              timeout: L4_timeout,          % Indicates if timeout occurred.
              seed:    nat                  % Seed used for randomness.
            #]
```

Source 3.25: PVS: expanded system state, with added *seed* field.

Before each call to the *preemption_action()* function, the seed is incremented to ensure its uniqueness. After a list of preemption actions has been retrieved, what remains is to execute those actions. Precisely that is the function of the *preemption_point_actions()* function, it executes each action and updates the system state correspondingly. The *preemption_point()* function is defined as follows:

```
preemption_point(partner: Thread_pointer, allow_timeout: bool,
                 state_old: System_state): System_state =
  LET
    % Increment the seed and use it to get a random list of preemption
    % actions.
    state_old = state_old WITH [seed := state_old`seed + 1],
    actions = preemption_action(state_old`seed)
  IN
    % Execute the preemption actions.
    preemption_point_actions(actions,partner,allow_timeout,state_old)
```

Source 3.26: PVS: *preemption_point()* function.

We see that the *preemption_point()* function not only takes the system state as one of its parameters, but also a boolean value named *allow_timeout*. The function of this value is to indicate if a timeout can occur in the preemption point, which is not always so because timeouts are only used in some functions (such as the *do_send_wait()* function).

Preemption actions

For execution of the randomly chosen preemption actions, the *preemption_point_actions()* function is used. The definition of this function is very simple. It takes the same parameters as the *preemption_point()* function calling it, but with one additional parameter that specifies the preemption actions to be executed. As a reminder, the *Preemption_actions* type is actually a list of *Preemption_action* instances. The *preemption_point_actions()* function first checks if the action list is empty, if so the system state is returned unmodified. If the list is not empty, the system checks which action it should execute and then updates the system state accordingly. It then continues executing the remaining actions until there are no actions left; the function is therefore (defined as) a recursive function[7].

```
% Execute a list of preemption point actions.
preemption_point_actions(actions: Preemption_actions, partner: Thread_pointer,
                         allow_timeout: bool,
                         state_old: System_state): RECURSIVE System_state =
  CASES actions OF
    null: state_old,
    cons(action,remaining_actions):
```

[6] A value which only function is to randomize behaviour is often referred to as a seed value.
[7] A recursive function calls itself.

```
    LET state_temp =
      % Check if a call was made to sys_thread_ex_regs().
      IF action = Sys_thread_ex_regs THEN
        sys_thread_ex_regs(state_old`this, state_old)
      % Only allow a timeout to occur when specified, most of the time a
      % timeout cannot occur.
      ELSIF action = Timeout AND allow_timeout THEN
        timeout(state_old`this, state_old)
      % Check if the partner is killed.
      ELSIF action = Kill THEN
        kill(partner, state_old)
      % A receiver can only become ready when it is not equal to the sender,
      % if so it would be waiting forever.
      ELSIF action = Receiver_ready THEN
        receiver_ready(state_old`this, partner, state_old)
      ELSE
        state_old
      ENDIF
    IN
    % Execute the remaining actions.
    preemption_point_actions(remaining_actions, partner, allow_timeout, state_temp)
  ENDCASES
  MEASURE length(actions)
```

Source 3.27: PVS: *preemption_point_actions()* function.

Most of these functions have very simple definitions, with the notable exception of the *receiver_ready()* function. As detailed earlier, the *receiver_ready()* function models a receiver becoming ready to receive a message from the sender; we are in fact modeling that the receiver is executing the receive part of IPC. A receiver becomes ready for a sender in its receiving loop, which is preceded by the necessary initialization. To prevent this initialization from occuring several times, we created a slightly modified version of the receive part in our *receiver_ready()* function, which uses a newly added *receiver_initialized* system state field; initially this field is set to false to guarantee that the receiver will always be initialized.

Another problem with the *receiver_ready()* lies in it calling *ipc_receiver_ready()*, which is a boolean-returning function with side-effects; the solution to this problem has been discussed earlier. We will now list the definition of the *receiver_ready()* function:

```
% This preemption action signifies a receiver becoming ready to receive a
% message from a sender.
receiver_ready(sender: Thread_pointer, receiver: Thread_pointer,
               state_old: System_state): System_state =
  LET
    state_old =
      % Check if the receiver has been initialized.
      IF NOT state_old`receiver_initialized THEN
        % Prepare the receiver using a 'random' partner.
        prepare_receive_dirty(Receiver_partner_thread, receiver, state_old)
          WITH [(receiver_initialized) := true]
      ELSE
        state_old
      ENDIF
  IN
  % Make sure the receiver is in the right state to receive from a sender.
  IF state_old`threads(receiver)`state`thread_receiving AND
     state_old`threads(receiver)`state`thread_ipc_in_progress AND NOT
     state_old`threads(receiver)`state`thread_cancel
  THEN
    % Closed wait: check if the sender is the same as the partner specified.
    IF sender = state_old`threads(receiver)`partner THEN
      % Check if the sender meets the conditions of the receiver
      IF in_sender_list(receiver, state_old) AND
         ipc_receiver_ready(sender, receiver, state_old)
      THEN
        % Apply the changes caused by the side-effects of the
```

```
        % ipc_receiver_ready() function.
        ipc_receiver_ready_change(sender,receiver,state_old)
      ELSE
        state_old
      ENDIF
    % Open wait: no partner should be specified and the sender should be
    % the first item.
    ELSE
      % Check if the sender is not ready in which case it should be removed
      % from the sender list.
      IF NOT ipc_receiver_ready(sender,receiver,state_old) THEN
        sender_dequeue_head(receiver,state_old)
      ELSE
        % Apply the changes caused by the side-effects of the
        % ipc_receiver_ready() function.
        ipc_receiver_ready_change(sender,receiver,state_old)
      ENDIF
    ENDIF
  ELSE
    state_old
  ENDIF
```

Source 3.28: PVS: *ipc_receiver_ready()* function.

Chapter 4

Model verification

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger W. Dijkstra

"The problem with wrong proofs to correct statements is that it is hard to give a counterexample."

Hendrik W. Lenstra

4.1 Verification attempts

In this chapter we will discuss our verification attempts. Earlier, we listed the properties we attempted to prove. Here we will give the formal definition of each property we tried to prove. As the model and the properties were directly tied to each other, most properties required alterations to the model. These will be discussed along with the problems encountered during the verification.

4.1.1 Property 1: removal of sender's thread lock on receiver

Definition

When a sender wants to send a message to a receiver, they first have to agree upon engaging in IPC; this is referred to as the *handshake*. If the handshake has been successful, the sender will try to send the message to the receiver. During the message sending, it is vital that the receiver thread is not locked by another thread, as otherwise the message could not be transferred. Therefore, when the receiver is locked by another thread, the sender acquires that lock before sending the message. After the message has been sent, there should not be any thread lock remaining on the receiver, as it then becomes unavailable to other threads until the lock is released.

Model details

In Fiasco, acquiring a thread lock is done by calling *lock_dirty()*, wherein the lock owner of the target thread is set to the current thread (which, in our model, corresponds to the sender thread pointed to by the *this* field in the system state). The *lock_dirty()* function in our model is a typical example of an abstracted function. In Fiasco, when the *lock_dirty()* function sees that the targeted thread is already locked by another thread, it enters a loop from which it only breaks if the lock has been released[1]. The

[1] To help the lock be released as soon as possible, the waiting thread donates its execution time to the lock owner; this mechanism is described in more detail in the Fiasco backgrounds section.

end result of the function is that the targeted thread is locked by the current thread. In our model, we only modelled this result and did not model the loop at all. We consider this a safe abstraction, as we are only interested in the result of the function, which is always the same.

```
% Set the thread lock of the partner thread to the <this> thread.
lock_dirty(partner: Thread_pointer, state_old: System_state): System_state =
  state_old WITH [(threads)(partner)`thread_lock := state_old`this]
```

Source 4.1: PVS: *lock_dirty()* function.

For the release of a thread lock, there are two similar functions, which only differ slightly in their functionality. The main difference is that in the first function, *clear_dirty()*, the release of the lock can involve a switch of execution context. However, as this was undesirable in some cases, the *clear_dirty_dont_switch()* function was created. As we have not modelled scheduling (which includes execution context switches), both functions are functionally equal in our model. As the implementation details have once again largely been abstracted (just as in the *lock_dirty()* function), the two functions have exactly the same definition in our model. We have still included both functions as it allows for a better reflection of the source code and it allows for an easier transition should we later decide to model scheduling.

```
% Remove the thread lock on the partner thread by setting the thread lock
% owner to the zero thread.
clear_dirty(partner: Thread_pointer, state_old: System_state): System_state =
  state_old WITH [(threads)(partner)`thread_lock := Zero_thread]

% Remove the thread lock on the partner thread by setting the thread lock
% owner to the zero thread.
clear_dirty_dont_switch(partner: Thread_pointer,
                        state_old: System_state): System_state =
  state_old WITH [(threads)(partner)`thread_lock := Zero_thread]
```

Source 4.2: PVS: *clear_dirty()* and *clear_dirty_dont_switch()* functions.

As we had already modelled thread locks, the only required alteration to our model (we consider the function mentioned above as additions, not alterations) was to incorporate the notion of a handshake, which takes place in the *try_handshake_receiver()* function. However, we needed to check the success of a handshake *after* the send part had finished (which is represented by the *do_send_part()* function) and therefore a field named *handshake_attempted* was added to the system state:

```
System_state: TYPE =
            [#
              this:    This_thread_pointer, % Pointer to active thread.
              threads: Thread_list,         % List of all threads.
              error:   Ipc_err,             % Indicates if error occurred.
              timeout: L4_timeout,          % Indicates if timeout occurred.
              seed:    nat,                 % Seed used for randomness.
              handshake_attempted: bool     % Indicates if handshake attempted.
            #]
```

Source 4.3: PVS: basic system state, with integrated *handshake_attempted* field.

We can now set the *handshake_attempted* field in the *try_handshake_receiver()* function; initially its value is set to false (in the *sys_ipc()* function), which reflects the fact that initially no handshake has been attempted.

```
try_handshake_receiver(partner: Thread_pointer,
                       state_old: System_state): System_state =
  % Immediately return with an error if the partner is the Zero- or Nil thread
  % or the partner is an invalid thread.
  IF partner = Zero_thread OR
     state_old`threads(partner)`state`thread_invalid OR
     partner = Nil_thread
  THEN
    state_old WITH [error := true]
  ELSE
  LET
```

```
    % Set the <handshake_attempted> field to true because the receiver is
    % valid.
    state_temp = state_old WITH [handshake_attempted := true]
  IN
    [...]
  ENDIF
```

Source 4.4: PVS: setting the *handshake* field.

Verification and problems

We are now ready to give the formal definition of our property in PVS:

```
% If a handshake has been attempted, that should imply that after finishing
% the send part, there should be no thread lock on the partner.
do_ipc_send_part_handshake_attempted_lock_free: LEMMA
  FORALL (partner: Thread_pointer, have_receive: bool, state_old: System_state):
    LET state_new = do_ipc_send_part(partner, have_receive, state_old) IN
      NOT state_old`handshake_attempted AND
      NOT state_old`error AND
      state_new`handshake_attempted IMPLIES
        state_new`threads(partner)`thread_lock = Zero_thread
```

Source 4.5: PVS: property 1, formal definition.

The lemma states that if a handshake has been attempted in the send part, the receiver (named *partner* for similarity to the Fiasco source code) has no thread locking it after handling the send part. Although the definition is pretty self-explanatory, there remains one odd requirement in the lemma: initially the *error* field of the system state must not be set. This requirement can be explained quite easily when we remember its semantics, which are that it signifies if an error occurred. When *do_ipc_send_part()* is called, the *error* field should be set to false as no error can have occurred at that time. This holds in our model as no error-returning functions are called before *do_ipc_send_part()*; combined with the initialization to false by *sys_ipc()*, this results in the *error* field being false before *do_ipc_send_part()* is called.

One of the problems we encountered was that, due to an initial lack of modularity, we only defined the lemma listed above and thus small changes in the model often required (almost) the whole lemma to be redone. To circumvent this, we split the proof into parts, where each function received its own lock-related lemma. In this situation, lemmas became dependent upon other lemmas, essentially creating a sort of hierarchy with the formal property lemma at the top. This modular approach was much more resistant against changes in the model, most often a change in the model only required a single proof to be redone. Therefore, this approach has also been taken in the verification of the other properties (where applicable).

4.1.2 Property 2: waking up receiver in combined send/receive

Definition

When a combined send- and receive IPC call is made, the sender and receiver both assume the sender- and receiver role. First, the sender tries to send a message to the receiver. Once that has been successful, the sender enters a state in which it waits for the receiver to send a message back. It is at this point that the roles are swapped: the sender becomes the receiver and vice versa. When the receiver assumes the sender role, it should be in a state where it is ready to be scheduled. Should this not be the case, it might never be scheduled, which in turn would result in the sender forever waiting for a message from the receiver. It is therefore imperative that when the receiver assumes the sender role, it is in a state ready to be scheduled.

Model details

Whether a thread is ready to be scheduled depends on a single state bit being set: the *thread_ready* bit. As we have already modelled threads and their states (including the *thread_ready* bit), we are only left with determining if the receiver will be assigned the sender role during IPC. For this purpose, we can use a function of Fiasco IPC which determines if a sender and receiver are engaged in IPC: *in_ipc()*. The definition of this function is listed below:

```
% Indicates if the receiver is engaged in IPC with the sender.
in_ipc(sender: Thread_pointer, receiver: Thread_pointer,
       state_old: System_state): bool =
  state_old`threads(receiver)`state`thread_transfer_in_progress AND
  state_old`threads(receiver)`state`thread_ipc_in_progress AND NOT
  state_old`threads(receiver)`state`thread_cancel AND
  state_old`threads(receiver)`partner = sender
```

Source 4.6: PVS: *in_ipc()* definition.

At any single moment it is thus possible to determine if a sender and receiver are engaged in IPC with each other. We can use this in our property to determine, after the send part, if the sender and receiver are still engaged in IPC, as this implies that the sender expects a message back from the receiver, which thus will assume the sender role.

Verification and problems

The formal definition in PVS of our wakeup property is as follows:

```
% When the sender- and receiver are still engaged in IPC after the send part
% has finished, the receiver should be in a state ready to be scheduled, which
% means that the <thread_ready> bit of the receiver should be set.
do_ipc_send_part_in_ipc_receiver_awoken: LEMMA
  FORALL (partner: Thread_pointer, have_receive: bool, state_old: System_state):
    LET state_new = do_ipc_send_part(partner, have_receive, state_old) IN
      NOT state_new`error AND in_ipc(state_new`this, partner, state_new) IMPLIES
        state_new`threads(partner)`state`thread_ready
```

Source 4.7: PVS: property 2, formal definition.

This lemma clearly states that if the sender (pointed to by *state_new`this*) and the receiver (pointed to by *partner*) are still engaged in IPC after finishing the send part, the *thread_ready* bit of the receiver should be set. Once again we require that the *error* field is unset before calling *do_ipc_send_part()*, the reasoning follows the one laid down in the previous property.

4.1.3 Property 3: validation of assertions in the code

Definition

As in many C++ programs, Fiasco (and its IPC path) contains many calls to the *assert()* function. When a call is made to the *assert()* function, one wants to test if a critical condition holds at that moment; the condition is an expression evaluating to a boolean value. If the expression evaluates to true, execution is continued as if nothing happened. However, if the expression evaluates to false, execution of the program is aborted completely. A call to assert should thus only be made when one wants to test a critical condition. For the correct functioning of the program, all assertions should evaluate to true.

Ensuring that not a single assertion fails is precisely the property we attempted to verify. As assertions should only evaluate critical conditions, our model should be able to prove these. If, however, our model fails to do so, we have either found a bug in the code or in our model. Being able to verify all assertions thus would also increase the confidence we have in the correctness of our model.

Model details

As the assertions are spread over many functions, we needed to modify the system state. For this purpose, we added the *assertions_held* field to the system state:

```
System_state: TYPE =
          [#
            this:     This_thread_pointer, % Pointer to active thread.
            threads:  Thread_list,         % List of all threads.
            error:    Ipc_err,             % Indicates if error occurred.
            timeout:  L4_timeout,          % Indicates if timeout occurred.
            seed:     nat,                 % Seed used for randomness.
            handshake_attempted: bool,     % Indicates if handshake attempted.
            assertions_held:     bool      % Indicates if assertions held.
          #]
```

Source 4.8: PVS: basic system state, with added *assertions_held* field.

The semantics of this field are that it signifies if all assertions have held, the failure of a single assertion will result in *assertions_held* being unset for the remainder of IPC. To achieve this, the status of the *assertions_held* field is always updated by combining the assertions with the previous status of the *assertions_held* field using the *and* operator. By assuring that *assertions_held* is initially set to true (as no assertions can possibly have failed at that time), the *assertions_held* field will only be true after handling IPC if all assertions held.

Consider the following, simple assertion, which is made at the beginning of the receive part:

```
assert(have_receive);
```

Source 4.9: C++: *have_receive* assertion.

Translated to our model, this statement results in the following definition:

```
% Assert that at this point the <have_receive> parameter is true.
state_old WITH [(assertions_held) :=
                  state_old`assertions_held AND
                  have_receive]
```

Source 4.10: PVS: *have_receive* assertion.

We should note though that not all assertions have been integrated into our model, most notably we have excluded assertions in the *ipc_receiver_ready()* function and in the receive part. The first omissions were due to us not modeling the *receiver* field and the second were due to the abstractions applied to the receive part (one assertion in the receive part remained though, namely the one listed above).

Verification and problems

The lemma corresponding to our property is defined as follows::

```
% The various assertions in the code should hold after engaging in IPC.
assertions_held: LEMMA
  FORALL (have_send: bool, partner: Thread_pointer, have_receive: bool,
          sender: Thread_pointer, state_old: System_state):
    % Engage in IPC with the supplied parameters.
    LET state_new = sys_ipc(have_send, partner,
                            have_receive, sender, state_old) IN
      NOT state_old`threads(state_old`this)`state`thread_polling IMPLIES
        state_new`assertions_held
```

Source 4.11: PVS: property 3, formal definition.

Contrary to the previous two properties, this lemma does not require that the *error* field is initially not set. This is due to the fact that this lemma uses the *sys_ipc()* function, which already initializes the *error* field to false. However, we *do* make another assumption, namely that initially the *thread_polling* bit is not set. The reason why we added this assumption will now be expanded upon.

Problem 1: *thread_polling* bit unset

Our first problem occured when we tried to verify the lemma above *without* the *thread_polling* assumption. The problem popped up when we tried to verify the following assertion in the *do_ipc_send_part()* function:

```
% Start with a handshake and immediately return if an error occured.
LET state_temp = try_handshake_receiver(partner,state_old) IN
IF state_temp`error THEN
  state_temp
ELSE
LET
  % Assert that after a successful handshake the sender thread is no
  % longer polling.
  state_temp = state_temp WITH [(assertions_held) :=
                 state_temp`assertions_held AND
               NOT state_temp`threads(state_temp`this)`state`thread_polling],
  [...]
IN
  [...]
ENDIF
```

Source 4.12: PVS: *thread_polling* assertion.

More verbosely, the above assertion tries to verify that after a successful handshake, the *thread_polling* bit of the *this* thread, which is the sender, is not set. The semantics of the *thread_polling* bit are that it is only set (on the sender) when the sender has to wait for the receiver to become ready; essentially the sender *polls* the receiver (hence the bit's name) at intervals to see if it has become ready. Once the receiver has become ready (the handshake has been successful), the *thread_polling* bit should be unset as the sender is no longer waiting. To verify this assertion, we created the following lemma:

```
% If no error has occured when calling try_handshake_receiver(), the
%<thread_polling> of the <this> thread will always be unset.
try_handshake_receiver_no_error_not_polling: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = try_handshake_receiver(partner,state_old) IN
      NOT state_old`error AND
      NOT state_new`error IMPLIES
        NOT state_new`threads(state_new`this)`state`thread_polling
```

Source 4.13: PVS: *try_handshake_receiver_no_error_not_polling* lemma.

If we were able to verify this lemma, we would have verified the *thread_polling* assertion. However, when trying to verify this lemma, we found that one specific path in the *try_handshake_receiver()* function rendered the lemma unprovable. We will list the lines involved in this path (where less relevant parts have been replaced by [...] for brevity purposes):

```
% Try to have the sender and receiver agree upon engaging in IPC.
try_handshake_receiver(partner: Thread_pointer,
                       state_old: System_state): System_state =
  % Immediately return with an error if the partner is the Zero- or Nil thread
  % or the partner is an invalid thread.
  IF partner = Zero_thread OR [...] THEN
    state_old WITH [error := true]
  ELSE
  LET
    [...]
  IN
    % IPC has not been cancelled, now check if the receiver is ready for
    % the sender. If not, wait for the receiver to become ready.
    IF NOT sender_ok(state_temp`this,partner,state_temp) THEN
    LET
      state_temp = do_send_wait(partner,state_temp)
    IN
      [...]
    ELSE
      state_temp WITH [error := false]
```

```
    ENDIF
  ENDIF
```

Source 4.14: PVS: *try_handshake_receiver()* problematic lines.

The problematic path arises when the partner is valid and the *sender_ok()* function returns true, which results in a return from the function with the *error* field set to false. As, in our model, the functions before the call to *sender_ok()* do not modify the *thread_polling* bit, the assertion only succeeds if the *thread_polling* bit is not set before *try_handshake_receiver()* is called.

Unfortunately, this assumption cannot be found anywhere in the source code. To test this assumption, we modified the Fiasco source code and added our assumption as an assertion:

```
assert(!(state() & Thread_polling));                                    //!

if (EXPECT_TRUE(have_send_part || have_receive_part))
  ret = do_ipc(have_send_part, partner,
         have_receive_part, sender,
         t, regs);
```

Source 4.15: C++: *thread_polling* bit, added assertion.

As can be seen, we assert that the *thread_polling* bit is not set before *do_ipc()* is called. After recompiling Fiasco (including our added assertion), we ran an IPC test application to check if the assertion failed. To make sure the system reached the assertion at all, we first tested a version where our assertion was preceded by an assertion guaranteed to fail[2]. After we verified that our assertion was indeed evaluated, we ran the test application again (with the obviously failing assertion removed of course) to see if the assertion would fail, which it did not. We now had some confidence that our assumption was correct, however the quotation at the beginning of the chapter by Lenstra can be applied to this situation: our assertion test can only prove its incorrectness, not its correctness. This is due to the fact that the IPC tester might not examine all possible states, a state might be missed in which the assertion would fail.

As an alternative way to increase confidence that our assumption holds, we tried to verify that the *thread_polling* bit being unset is an invariant of the Fiasco IPC path. Here we also have to assume that the *thread_polling* is not set initially, as otherwise we would have the exact same problem as before. The invariant corresponds to the following lemma:

```
% If we assume that initially, the <this> thread's <thread_polling> bit is
% not set, it will still be unset after calling sys_ipc().
sys_ipc_not_polling: LEMMA
  FORALL (have_send: bool, partner: Thread_pointer, have_receive: bool,
          sender: Thread_pointer, state_old: System_state):
    % Engage in IPC with the supplied parameters.
    LET state_new = sys_ipc(have_send,partner,
                            have_receive,sender,state_old) IN
      NOT state_old`threads(state_old`this)`state`thread_polling IMPLIES
        NOT state_new`threads(state_new`this)`state`thread_polling
```

Source 4.16: PVS: unset *thread_polling()* bit invariant.

Unfortunately, we failed to prove both this lemma and the lemma mentioned earlier which was required for proving our property. Both lemmas depended on the *thread_polling* bit always being unset after calling *do_send_wait()*. The semantics of this function are that the sender waits for the receiver to become ready and only returns when the sender is ready or an error occured. In both cases, the *thread_polling* bit should have been unset as the sender is no longer waiting. Therefore, we had to prove the following lemma to finish the proofs of both lemmas:

```
% After calling do_send_wait(), the <this> thread's <thread_polling> bit will
% always be unset.
do_send_wait_not_polling: LEMMA
```

[2]We used the simplest solution possible: assert(false);

```
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = do_send_wait(partner, state_old) IN
      NOT state_new`threads(state_new`this)`state`thread_polling
```

Source 4.17: PVS: unset *thread_polling()* bit invariant.

At the start of this function, the *thread_polling* bit is set to indicate that the sender is waiting for the receiver to become ready. When we tried to prove this lemma, we found a path in which the *thread_polling* bit was not unset at the return of the function.

```
% Create a thread state which bits to add to the <this> thread.
LET
  [...]
  add_bits = TS_empty WITH [thread_polling := true,
                            thread_send_in_progress := true,
                            thread_ipc_in_progress := true],
  state_temp = state_add_dirty(state_temp`this, add_bits, state_temp),
  [...]
IN
  [...]
          % Return if the receiver is ready to receive a message from the <this>
          % thread. If not enter the waiting loop.
          IF sender_ok(state_temp`this, partner, state_temp) THEN
            state_temp WITH [error := false]
          ELSE
            % Enter the waiting loop.
            [...]
          ENDIF
  [...]
```

Source 4.18: PVS: *do_send_wait()* function, problematic *thread_polling* path.

The erroneous situation occured, once again, when the *sender_ok()* function is called. All functions after *state_add_dirty()* and before *sender_ok()* do not modify the *thread_polling* bit. When *sender_ok()* returns true, the *do_send_wait()* function immediately returns and execution is continued in the *do_ipc_send_part()* function. As at this point the *thread_polling* bit will be set to true (due to the *state_add_dirty()* function call), the *thread_polling* assertion will fail. This time, there was no additional assumption which would solve our problem. Our only option was to define an axiom:

```
% The <thread_polling> bit is not set when sender_ok() returns true.
sender_ok_not_polling: AXIOM
  FORALL (partner: Thread_pointer, state_old: System_state):
    sender_ok(state_old`this, partner, state_old) IMPLIES
      NOT state_old`threads(state_old`this)`state`thread_polling
```

Source 4.19: PVS: the axiom used in the *do_send_wait()* function.

With this axiom, our problem was solved, however it indicated that either our model was incorrect or the source code. After careful consideration of our model, we contacted the author of the Fiasco IPC path, who confirmed that we had indeed found a bug in the source code. This bug could be fixed by unsetting the *thread_polling* bit after the *sender_ok()* function has returned true and before returning from the function. In our model, we resolved this problem by using our axiom, we could also have fixed the bug in our model but chose not to in order to keep the model consistent with the code.

Problem 2: sender not equal to receiver

At certain points in our proofs, we needed the assurance that certain bits remained unchanged after calling a specific function. For most functions this posed no problem, with the exception of the *preemption_point()* function. This function relied on the *preemption_point_actions()* function, which updates the system state according to, randomly generated, preemption actions. As the *receiver_ready* preemption action was quite complex, for each *preemption_point_actions()* lemma we created a corresponding lemma focusing only on that action, of which an example is shown below:

```
% The <thread_ipc_in_progress> bit of the <this> thread will not be changed
% after receiver_ready() was called.
receiver_ready_ipc_in_progress_unchanged: LEMMA
 FORALL(sender: Thread_pointer, receiver: Thread_pointer, state_old: System_state):
   LET state_new = receiver_ready(state_old`this, receiver, state_old) IN
       state_old`threads(state_old`this)`state`thread_ipc_in_progress =
         state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When no timeouts are allowed, the <thread_ipc_in_progress> bit of the
% <this> thread will not be changed after calling preemption_point_actions().
preemption_point_actions_no_timeout_ipc_in_progress_unchanged: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          state_old: System_state):
    LET state_new = preemption_point_actions(actions, partner, false, state_old) IN
      state_old`threads(state_old`this)`state`thread_ipc_in_progress =
        state_new`threads(state_new`this)`state`thread_ipc_in_progress
```

Source 4.20: PVS: *preemption_actions()*- and related *receiver_ready()* lemma.

Verifying the *preemption_point_actions()* lemma was very straightforward, as most preemption actions did not involve the *thread_ipc_in_progress* bit. However, verifying the corresponding *receiver_ready()* lemma proved troublesome. The problem occured when the sender was equal to the receiver and the receiver was to have its state initialized; this initialization modified the state in a way that prevented us from proving the lemma.

In essence, the problem originates in a situation that cannot occur in Fiasco, namely when a sender wants to send a message to itself. This would result in the sender having to wait forever for the receiver (itself) to become ready (at least with our modeling of timeouts). Although this exception is not handled in the Fiasco source code, we modified the model to reflect that a receiver cannot become ready for a sender when both are equal. We felt this was the right decision as it might not reflect the actual source code, but it did reflect its behaviour. The *preemption_point_actions()* function and the *receiver_ready_ipc_in_progress_unchanged* lemma were thus modified and the proof could be completed.

```
% Execute a list of preemption point actions.
preemption_point_actions(actions: Preemption_actions, partner: Thread_pointer,
                         allow_timeout: bool,
                         state_old: System_state): RECURSIVE System_state =
  CASES actions OF
    [...]
    % A receiver can only become ready when it is not equal to the sender,
    % if so it would be waiting forever.
    ELSIF action = Receiver_ready AND NOT state_old`this = partner THEN
      receiver_ready(state_old`this, partner, state_old)
    [...]
  ENDCASES
  MEASURE length(actions)

% The <thread_ipc_in_progress> bit of the <this> thread will not be changed
% after receiver_ready() was called and the sender (the <this> thread) was
% not equal to the receiver.
receiver_ready_ipc_in_progress_unchanged: LEMMA
 FORALL(sender: Thread_pointer, receiver: Thread_pointer, state_old: System_state):
   LET state_new = receiver_ready(state_old`this, receiver, state_old) IN
     NOT state_old`this = receiver IMPLIES
       state_old`threads(state_old`this)`state`thread_ipc_in_progress =
         state_new`threads(state_new`this)`state`thread_ipc_in_progress
```

Source 4.21: PVS: *receiver_ready()*- and dependent *preemption_actions()* lemma.

Problem 3: not engaged in IPC assertion

In the *do_send_wait_loop()* function, an assertion is made that the *in_ipc()* function should evaluate to false when the *thread_cancel* and *thread_transfer_in_progress* bits are not set. To put it another way:

only when the sender and receiver are engaged in IPC will the *thread_transfer_in_progress* bit be set. Unfortunately, we were unable to prove this assertion, which we expect is due to our simplified preemption point definition and lack of scheduling, as they result in the receiver state being less accurately modelled (which is precisely what is needed in the *in_ipc()* function). For our proof we thus had to resort to the following axiom:

```
% When the <thread_transfer_in_progress> bit of the <this> thread is not set,
% that implies that the partner and the <this> thread are not engaged in IPC.
not_transfer_in_progress_not_in_ipc: AXIOM
  FORALL (partner: Thread_pointer, state_old: System_state):
    NOT state_old`threads(state_old`this)`state`thread_transfer_in_progress
    IMPLIES
      NOT in_ipc(state_old`this, partner, state_old)
```

Source 4.22: PVS: the axiom used in the *do_send_wait_loop()* function.

Example proof

To conclude our discussion of our verification attempts, we will list an example of a typical proof of our model. We will look at the *assertions_held* lemma, which was the lemma corresponding to the property which verification attempt we just discussed. The proof will also show the modular structure of our proofs; one might expect the proof to be very large, but due to the use of the following sublemma it is actually very small (and therefore suitable for demonstration purposes).

```
% The various assertions in the code should hold after handling the send part
% when initially no error or timeout occurred.
do_ipc_send_part_assertions_held: LEMMA
  FORALL (partner: Thread_pointer, have_receive: bool, state_old: System_state):
    LET state_new = do_ipc_send_part(partner, have_receive, state_old) IN
      NOT state_old`error AND
      NOT state_old`timeout AND
      NOT state_old`threads(state_old`this)`state`thread_polling AND
       state_old`assertions_held
      IMPLIES
        state_new`assertions_held
```

Source 4.23: PVS: property 3, sublemma for the *do_ipc_send_part()* function.

We are now ready to proceed with the proof itself.

The proof of the `assertions_held` lemma starts with the formula as we have defined it earlier. As a precondition, it requires that initially the *thread_polling* bit of the *this* thread is not set. If this precondition is met, all assertions should hold (indicated by requiring the *assertions_held* field to be true) after calling *sys_ipc()*:

```
{1}  ∀ (have_send: bool, partner: Thread_pointer, have_receive: bool, sender: Thread_pointer,
          state_old: System_state):
       LET state_new = sys_ipc(have_send, partner, have_receive, sender, state_old) IN
         ¬ state_old`threads(state_old`this)`state`thread_polling ⟹ state_new`assertions_held
```

After some initializing commands (namely (skolem!)[3], (ground) and (flatten)), we end up with a basic version of the formula we want to prove:

```
{1}  state_old'`threads(state_old'`this)`state`thread_polling
{2}  sys_ipc(have_send', partner', have_receive', sender', state_old')`assertions_held
```

[3]When applying automatic skolemization, the ′ character is appended to universal quantifiers to indicate the constants they are replaced with. Unfortunately, the ′ character is very similar to the ‘ character that is used to access fields in a record. As an example of the possible confusion, accessing the this field of the state_old′ constant translates to: state_old′‘this.

We can see that the *thread_polling* requirement appears as consequent 1, which means that we can assume that it does not hold. If we could prove that it *does* hold, our proof would be completed. In our case, we will use consequent 1 to prove consequent 2. We now expand *sys_ipc()* to see its definition, where we see that it calls *do_ipc()* with the initialized system state as one of its parameters:

```
{1}  state_old'`threads(state_old'`this)`state`thread_polling
{2}  IF have_send' ∨ have_receive'
       THEN do_ipc(have_send', partner', have_receive', sender',
                   state_old'
                     WITH [(error) := FALSE,
                           (timeout) := FALSE,
                           (handshake_attempted) := FALSE,
                           (assertions_held) := TRUE,
                           (receiver_initialized) := FALSE])`assertions_held
     ELSE TRUE
     ENDIF
```

To prevent the same, initialized state from being displayed several times, we issue a (name-replace) command. In our proof, this results in all occurences of the initialized state being replaced with the name *state_old_initialized*:

```
{-1}  state_old'
        WITH [(error) := FALSE,
              (timeout) := FALSE,
              (handshake_attempted) := FALSE,
              (assertions_held) := TRUE,
              (receiver_initialized) := FALSE]
      = state_old_initialized
-------------------------------------------------------------------
{1}  state_old'`threads(state_old'`this)`state`thread_polling
{2}  IF have_send' ∨ have_receive'
       THEN do_ipc(have_send', partner', have_receive', sender',
                   state_old_initialized)`assertions_held
     ELSE TRUE
     ENDIF
```

Of course, the replaced state itself should still be known, so antecedent -1 is added to reflect the equality introduced by (name-replace). The result of the replace action itself can be seen in consequent 2. Please note that this command serves only cosmetic purposes, it does not change the state by any means. Although at this point the replace is only introducing overhead, our later sequents will benefit from it.

Once again, we have to expand a function to continue, in this case we expand *do_ipc*:

```
{-1}  state_old′
        WITH [(error) := FALSE,
              (timeout) := FALSE,
              (handshake_attempted) := FALSE,
              (assertions_held) := TRUE,
              (receiver_initialized) := FALSE]
       = state_old_initialized
─────────────────────────────────────────────────────────────────────────────────────────────
{1}   state_old′‘threads(state_old′‘this)‘state‘thread_polling
{2}   IF have_send′ ∨ have_receive′
        THEN IF have_receive′ ∧
                ¬ IF have_send′
                    THEN do_ipc_send_part(partner′, have_receive′, state_old_initialized)‘error
                  ELSE state_old_initialized‘error
                  ENDIF
             THEN do_ipc_receive_part(sender′, TRUE,
                                      IF have_send′
                                        THEN do_ipc_send_part(partner′,
                                                              TRUE,
                                                              state_old_initialized)
                                      ELSE state_old_initialized
                                      ENDIF)‘assertions_held
           ELSE IF have_send′
                  THEN do_ipc_send_part(partner′, have_receive′,
                                        state_old_initialized)‘assertions_held
                ELSE state_old_initialized‘assertions_held
                ENDIF
           ENDIF
      ELSE TRUE
      ENDIF
```

The benefit of the (name-replace) command can now clearly be seen, had not we not issued this command there would have been six (identical) initialized states being displayed. When one looks at how many lines the initialized state definition occupies, the replacement is a big improvement in readability.

At this point, one might expect that we would begin to instantiate the *have_send* or *have_receive* constants, which would introduce branching into our proof. Although this is possible, we first expand *do_ipc_receive_part()*, as by definition it does not change the *assertions_held* field. It thus has no influence on consequent 2, which leads to a simplified version of the consequent. This will prove beneficial later in the proof:

```
{-1}  state_old'
        WITH [(error) := FALSE,
              (timeout) := FALSE,
              (handshake_attempted) := FALSE,
              (assertions_held) := TRUE,
              (receiver_initialized) := FALSE]
       = state_old_initialized
------------------------------------------------------------------------------
{1}   state_old'`threads(state_old'`this)`state`thread_polling
{2}   IF have_send' ∨ have_receive'
        THEN IF have_receive' ∧
                ¬ IF have_send'
                    THEN do_ipc_send_part(partner', have_receive', state_old_initialized)`error
                  ELSE state_old_initialized`error
                  ENDIF
              THEN IF have_send'
                    THEN do_ipc_send_part(partner', TRUE, state_old_initialized)`assertions_held
                  ELSE state_old_initialized`assertions_held
                  ENDIF
             ELSE IF have_send'
                    THEN do_ipc_send_part(partner', have_receive',
                                          state_old_initialized)`assertions_held
                  ELSE state_old_initialized`assertions_held
                  ENDIF
             ENDIF
        ELSE TRUE
        ENDIF
```

Once again, one might expect us to introduce branching in our proof at this point. However, as we are only left with the *do_ipc_send_part()* function, we can use the *do_ipc_send_part_assertions_held* lemma listed earlier. This lemma states that calling *do_ipc_send_part()* results in the *assertions_held* field being true, which is precisely what we want to prove. If we add the lemma to our proof (with the (lemma) command) and automatically instantiate it (using the (inst?) command), we get the following result:

```
{-1}  LET state_new = do_ipc_send_part(partner', have_receive', state_old_initialized) IN
        ¬ state_old_initialized`error ∧
         ¬ state_old_initialized`timeout ∧
          ¬ state_old_initialized`threads(state_old_initialized`this)`state`thread_polling ∧
           state_old_initialized`assertions_held
          ⟹ state_new`assertions_held
{-2}  state_old'
        WITH [(error) := FALSE,
              (timeout) := FALSE,
              (handshake_attempted) := FALSE,
              (assertions_held) := TRUE,
              (receiver_initialized) := FALSE]
       = state_old_initialized
-------------------------------------------------------------------------------
{1}   state_old'`threads(state_old'`this)`state`thread_polling
{2}   IF have_send' ∨ have_receive'
        THEN IF have_receive' ∧
                ¬ IF have_send'
                    THEN do_ipc_send_part(partner', have_receive', state_old_initialized)`error
                  ELSE state_old_initialized`error
                  ENDIF
              THEN IF have_send'
                    THEN do_ipc_send_part(partner', TRUE, state_old_initialized)`assertions_held
                  ELSE state_old_initialized`assertions_held
                  ENDIF
             ELSE IF have_send'
                    THEN do_ipc_send_part(partner', have_receive',
                                          state_old_initialized)`assertions_held
                  ELSE state_old_initialized`assertions_held
                  ENDIF
             ENDIF
        ELSE TRUE
        ENDIF
```

We see that the *do_ipc_send_part_assertions_held* lemma has been added as antecent -1, with filled in values due to the automatic instantiation. A closer look at those values reveals that they correspond to the values in consequent 2, which is precisely what we want as we want to prove that consequent 2 holds. However, there are some additional requirements that have to be met before antecent -1 can be said to hold. Among these requirements we find that the *thread_polling* bit should initially not be set, which is precisely what consequent 1 states. The other requirements correspond directly to the initialization done by *sys_ipc()*, and can therefore be directly found in antecedent -2. We thus have all requirements to prove antecent -1, which allows us to prove that consequent 2 holds. Please note that by expanding *do_ipc_receive_part()* and using the *do_ipc_send_part_assertions_held* lemma, we did not have to introduce any branching in our proof. Most often though, this will not be possible.

To finish the proof, two calls to (assert) suffice; the first is necessary for antecent -1 to be removed of its *LET* construction and the second completes the proof:

This completes the proof of `assertions_held`.

Q.E.D.

Chapter 5

Discussion

"The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them."

Sir William Bragg

5.1 General discussion

When one wants to attempt verification of a program, that program has to be specified in a language that supports theorem proving. As most programs are written in languages that do not support theorem proving (such as Java and C++), a verification of such a program requires a conversion to a theorem prover (such as PVS). This conversion has to retain the exact semantics of the program, otherwise proofs in the theorem prover do not necessarily apply to the source code.

If the semantics of both the program's language and the theorem prover are known, an automatic conversion between these two becomes possible. For larger programs this is an absolute must, as a full, manual conversion would take a huge amount of time. Even small programs can benefit from an automatic conversion as it is not unlikely that an error is introduced in the manual conversion. Unfortunately, for many languages a fully functional conversion is not yet available; most notably this list of languages includes the widely used C++ language. In practice therefore, automatic conversion is not doable for many programs. Even if an automatic conversion would be possible, full verification of programs is (currently) not a realistic option because of the involved complexity.

If no automated- or full, manual conversion is possible, the situation is not hopeless though. An alternative solution is to create a model of the program and apply verification to that model. The disadvantage of this method is that it involves a manual conversion and that the proofs cannot be said to directly apply to the program, they are only guaranteed to apply to the model. The proofs are therefore only useful insofar as one trusts the model to be an accurate representation of the source code. However, such a model *can* lead to the detection of flaws in the program design, which we will discuss later. A model also allows for abstraction, which can help to focus on a specific property of the program and usually also lessens the amount of time spent on creating the model and proofs. This approach also opens up the possibility for larger programs to undergo verification, albeit only partial.

Even if one manages to formally verify source code, there is often still one vulnerable step remaining: compilation of the source code to machine code. This conversion has to retain the source code's semantics, as otherwise the verified properties of the source code do not necessarily apply to the compiled application.

5.2 Research discussion

Because we worked on C++ code, our options were (at the time) limited to creating a model of the code. However, even if a fully functional C++ to PVS converter had been available, we would probably still have opted for creating a model of the code because of the complexity of the IPC subsystem, which would probably have resulted in very large and complex proofs. This complexity arises because the IPC subsystem is not an isolated subsystem, it has in fact many interdependencies with other parts of the kernel (such as the scheduling subsystem). Therefore, creation of the model was quite tedious, even though the code was fairly well-documented and there was a thesis by René Reusner describing the general IPC outline. However, one should take into account though that we had no prior knowledge of the system; had the model been created by someone (intimately) familiar with the source code it would probably have sped up the creation significantly. As creating the model required us to gain a lot of insight in the code, the conversion itself greatly helped in increasing our understanding of the system. We consider this to be an advantage of model creation over automatic conversion, which requires no understanding at all.

While creating the model, we experimented with an approach in which we defined a very basic model that could be used and overridden in other theories. Should properties (which are defined in separate theories) require a more detailed model, they could override parts of the basic model and replace them with more detailed versions. The main benefit of this approach is that it results in very minimalistic models: a theory would only expand those parts relevant to its property. This approach would likely have resulted in more compact proofs. Although this approach worked fairly well, there were some disadvantages. As we wanted our model to be a faithful representation of the code, a single, detailed model, which was used in all properties, better reflected the real situation. Another disadvantage was that there was no single point of definition; it was possible to have two theories override parts in conflicting ways. The obvious downside of a single, detailed model was that the proofs became more involved, but the benefit of having a more faithful representation of the code outweighed this disadvantage in our opinion.

Creating the model itself was an iterative process, one in which we frequently revisited our design because of slight discrepancies to the code and gained insights into the inner workings of the system. To prevent a single change resulting in a single, large proof having to be redone, we opted for a very modular structure in which lemmas were broken down into smaller lemmas. To better enable the breakdown of lemmas into smaller lemmas, some of the larger functions were split into smaller functions. The original lemmas for the larger functions could now be broken into smaller lemmas specifically targeting the smaller functions. It is important to note that the semantics of the original function needed to be retained when it was split, however this posed no real problems. Through this modular approach we created proofs that were much more resistant to model changes, although some parts of the proof (particularly automatic lemma instantiation) still rendered proofs quite vulnerable to changes.

One of the main issues when creating the model was how to model the interruptible nature of Fiasco's IPC implementation. Our solution made use of the fact that the IPC path was only interruptible at specific points in the code (referred to as preemption points). After each preemption point, several actions might have influenced the ongoing IPC (of which we were trying to prove a property). We therefore modelled a preemption point as a function which randomly executed a list of actions that could influence IPC (which included an action where nothing happens). Although this was of great use in our model, it was also a quite risky solution. The main problem is that you have to known *all* actions that can occur in a preemption point along with their effects on IPC. Besides using the source code, we also used the thesis by René Reusner [Reu05] as a reference, but there is no guarantee that we did not miss any subtle actions (which is not unlikely given the concurrent structure of Fiasco). The obvious solution is to actually model the concurrency, but that would have greatly increased the complexity of the model and the proofs and has therefore been abstracted away.

Two of the three lemmas we tried to verify could be proven without resorting to additional assumptions.

Although the two verified properties might seem simple, they were both important to the functioning of the system in (very) specific situations. In both cases, we verified that specific threads would not be waiting endlessly in a specific situation, an important property for a real-time kernel. The third property, verification of the assertions in the code, was mainly used to increase confidence in the correctness of our model. It is with this property that we encountered several problems. One problem was the result of our simplified modeling of preemption points, more specifically the modeling of a receiver becoming ready for a sender. When the sender was equal to the receiver, it could become ready for itself which is not possible in the Fiasco IPC implementation we modelled (which included a simplified timeout representation). This however, was not directly stated in the source code but was inferred from it.

The second problem we encountered was verifying that at a certain point the sender and receiver were not engaged in IPC. Unfortunately, our verification attempts failed and to continue verification we created a (hopefully temporary) axiom dealing with this assertion. We believe our inability to verify this assumption is in large part due to our simplified model of preemption points, as the assertion mostly deals with the receiver becoming ready which is modelled in the preemption point.

The most important problem we found dealt with the *thread_polling* state bit assertion. In our attempts to verify this specific assertion, we found that we had to assume that initially (before IPC was started) the *thread_polling* bit was not set. We tried to verify this through adding the assumption as an assertion in the Fiasco source code, which was then recompiled and an IPC test suite was run on it. Unfortunately, this method cannot verify our assumption as we are not sure that it tests the assertion in all possible states. It could however provide a counter-proof to our assumption if the assertion had failed, but it did not fail. As an alternative to this method, we tried to verify that the *thread_polling* bit being unset was an invariant of IPC; this verification also required the assumption that the *thread_polling* bit is initially not set. Unfortunately, we failed to verify both the invariant and the original assertion for the exact same reason. The problem was with the *do_send_wait()* function, which ought to unset the *thread_polling* bit before returning. We found that there was a path though in which this was not done. To rule out the most obvious cause of this problem, we checked if the model did not correctly reflect the source code. However, we failed to see any discrepancies between our model and the source code. Not even our simplified preemption point could have been the cause, as the source code made it clear that the *thread_polling* bit was only changed at very specific functions, all of which were not involved in a preemption point. Our next step was to check with the designer and implementer of the IPC path, René Reusner, who verified that we had indeed found a bug in the source code (which could be easily fixed). Only when we assumed this bug fixed and used our initial *thread_polling* state assumption were we able to prove the assertion.

We consider the finding of an actual bug in Fiasco's IPC path an important finding. Should the bug occur, it would crash the whole kernel (and thus everything running on top of it). Removal of even one such bug can therefore be considered quite important. It is not odd that the bug had not been found earlier, as it is highly unlikely to occur[1]. Although one bug might not seem like much, this can be due to several reasons. One option is that the properties chosen were too simple, they were perhaps not likely to contain any errors at all. Another option is that our model is too abstract or simply incorrect, which might lead to some (lower-level) errors not being found. The last option is that there simply were not many errors in the modelled part of the code. We consider this a very plausible option, as the version of Fiasco we modelled had been extensively tested over a long period of time.

Even more important besides finding the bug is the fact that our abstract model was able to find it. This shows that it is not necessary to have a full, one-on-one model in order to find bugs. Please note once again that we have applied a lot of abstractions to our model, an essential component as scheduling has even been completely left out. For larger pieces of software, (partial) verification can thus be a feasible solution when one uses an abstract model. Verification by creating an abstract model can also benefit smaller programs as a more compact model is likely to result in easier, smaller proofs, which can save

[1] Due to the very short time in which the receiver has time to become ready.

valuable time. As an abstract model by definition is not completely equal to the source code, there may be subtle errors that are missed though.

The choice of abstractions is of the utmost importance as a wrongly chosen abstraction might simplify the model too much and a wrongly defined abstraction can easily lead to incorrect proofs. Defining the abstractions should thus preferrably be done by someone very familiar with the inner workings of the source code. Although the abstractions require a lot of thought, we suggest that the rest of the model is created by keeping the conversion as close to the source code as possible. The main advantage of this method, having a model very faithful to the source code, is increased confidence in the model accurately reflecting the source code. This is important for a model as it inheritly does not directly reflect the source code and thus any proofs of the model are only valid insofar as one trusts the model to be a good reflection of the source code.

Chapter 6

Conclusions and future work

"If you don't fail at least 90 percent of the time, you're not aiming high enough."

Alan Kay

6.1 Conclusions

For extremely critical programs (such as the software of the space shuttle), relatively small programs or algorithms (such as an encryption algorithm), the effort to apply full verification might be worthwhile. Unfortunately, for most other programs full verification is not a valid option because it requires too much effort. One of the reasons why full verification often requires so much time is the inability to automatically convert from source code to the proof system's language. This conversion is often necessary as most programs are written in languages that do not support proof creation. The only alternative to automated conversion is manual conversion, which requires (far) more effort than automated conversion. A manual conversion has the advantage though that it actively involves the converter in the conversion process, which is likely to lead to a better understanding of the system.

An alternative to full verification is partial verification, in which only certain parts of the code are verified. The main benefit of this approach is that it can use a more compact model. In order to create a compact model, one abstracts away parts that are not relevant to the properties one wants to prove. The danger of applying abstractions is that the model might not accurately reflect the source code, parts may have been abstracted away incorrectly. Another possible pitfall is that the model has become too abstract, which might prevent lower-level bugs from being found. Besides all these disadvantages, this approach also has its benefits. Its main benefit is its use of a more compact model, which likely results in smaller and simpler proofs. Furthermore, one does not need to know all the implementation details of parts that have been abstracted away, a functional description of these parts often suffices.

In our research, we created a model of a subsystem of the Fiasco microkernel, namely IPC. Even though the subsystem is relatively small, it was still quite complex due to its many inter-relations with other parts of the kernel. We therefore decided to create an abstract model of Fiasco IPC. Of all abstractions made, the most important were the preemption points, which are the points in the IPC path where execution might be (temporarily) switched to another thread. Using our abstract model, we tried to verify three different properties. The first property ensured that after sending an IPC message, there should not be any thread lock on the receiver. Our second property ensured that in a combined send and receive IPC call, the receiver is ready to send a message back to the sender when it is required to do so. These two properties were verified without requiring any additional assumptions. Both verified

properties ensured that in certain situations, threads would not be waiting infinitely, which is obviously important to a real-time system. The third property, in which we verified the various assertions in the IPC path, proved more troublesome. In the end, for one assertion we had to resort to using an axiom. We believe our inability to verify this property was due to our simplified modeling of preemption points. Our second problem was with another assertion, which required an additional assumption about the initial state. Even with this assumption we failed to verify the assertion, which turned out to be due to an error in the source code.

We were thus able to find a bug in the Fiasco IPC implementation, even with our greatly abstracted model. The bug we found was not likely to occur, but when it did it would have crashed the whole kernel and therefore everything running on top of it. The bug finding thus had a direct, practical use in that the Fiasco kernel had one bug removed from its code. Although useful, we consider the fact that our abstract model was able to find the bug more important. It shows that it is not necessary to create a strict, one-on-one conversion of the source code to find bugs. This opens up the possiblity for verification on larger programs, where one only focuses on those aspects of particular interest for verification and abstracts away the rest.

When one attempts to create an abstract model, choosing those abstractions wisely is a key step that requires a lot of insight into the inner workings of the system. We recommend that everything that is not abstracted away is converted as faithful to the code as possible, as this greatly increases the confidence one has in the correctness of the model. This is important as proofs of a model are only useful insofar one trusts the model to be a correct representation of the source code.

6.2 Future work

As has become clear in this thesis, there are still many open problems and points of improvement. In this section we will discuss the most important.

6.2.1 Further verification

As we have only verified three properties of IPC in Fiasco, there are many other properties open for verification. It might be more interesting though to verify our three properties in an expanded version of our model to see if they can still be proven. Every abstraction applied is a candidate for expansion, however we consider preemption points the most interesting candidate for expanding upon for the following reason. As preemption points form an integral part of IPC, they should preferrably be modelled as faithful to the source code as possible. Conversely, in our model preemption points are the most abstracted component. As preemption points themselves depend on other abstractions, such as the complete abstraction of scheduling and the abstraction of the receive part (corresponding to a receiver becoming ready). Therefore an expansion of preemption points in the model would most likely also include integration respectively expansion of these abstractions into the model.

Not only is inclusion of scheduling a required condition for more accurate preemption point modeling, it is also a very important aspect of Fiasco (IPC) itself. Modeling scheduling is therefore not only necessary for a better preemption point model, but it also opens up several other possible verification properties and improves the model's similarity to the source code. An interesting scheduling-related property would be verifying the absence of priority inversion in IPC.

If scheduling gets included into the model and preemption points get a more faithful representation, it makes sense to move the receive part functionality from the preemption points back to its original location. A receiver becoming ready can now be modelled by actually scheduling the receiver and following its IPC path. Having done this, it would be interesting to check if the sender being equal to the receiver once again leads to problematic proofs.

Another important aspect of Fiasco IPC that has been abstracted is the long IPC path. Not only does inclusion of the long IPC path make the path itself longer, and thus proofs larger, it also greatly increases the complexity of proofs dealing with the IPC path. This is in most part due to the full preemptability of the long IPC path. In the short IPC path, there are only specific points in which execution could be interrupted (namely the preemption points), but execution in the long IPC path can be interrupted after every instruction. For proofs, this leads to an enormous increase in the state space needed to be examined, as after every executed statement a number of branches is introduced that is equal to all possible actions a preemption might result in (of which there are many). The main problem is thus how to deal with this preemptability (and resulting state space explosion) efficiently in proofs.

As we have seen, there are many abstractions that can be expanded upon in future work. However, there is one open issue with the current proofs that certainly deserves looking into: our inability to verify the assertion stating that, at a certain point, the sender and receiver must not be engaged in IPC with each other. We believe that this property could not be proven due to our simplified version of a receiver becoming ready and possibly some additional assumptions might be necessary. If this simplified version is expanded upon, as we just discussed, it is interesting to see if our suspicions were correct or if there is a flaw in our model.

6.2.2 Modular proofs

Currently, one of the problems with proofs is their fragility: a small modification in the model often requires whole proofs to be redone. A possible solution to this problem is to create a more modular model, where a small modification results in only some or parts of the proof having to be redone. In our research, we successfully created a very modular model where small changes only required small changes in the proofs. To create a more modular model, we split some large functions into smaller functions, whilst retaining the original semantics. Although more modular proofs can thus in part be achieved in the current situation, certain aspects of the proof (particularly instantiations of lemmas) are often still bound to a specific model and are thus still susceptive to small changes in that model. For proofs to become less fragile, it is important that these fragile aspects of proofs can be handled better.

6.2.3 Automation of verification

Often even relatively basic proofs need some form of user-guidance. When a larger proof is being constructed, the amount of work involved becomes very high. For verification to become more successful it is therefore essential that as much as possible of the proof construction can be automated. This is particularly important for larger programs to be verified, but it is also important for quick verification of smaller programs or models.

6.2.4 Improved conversion

As said, the conversion from program to theorem prover is currently of vital importance to the verification of programs. To be able to apply full verification to larger programs, the conversion to a theorem prover should be done (almost) fully automated, as manual conversion can be very time-intensive. Although there are some converters available at the moment, most only support a subset of the features of the source language. For (much) wider applicability to real-life applications, it is important that all features of a language are supported. As an example, the LOOP tool [vdBJ01] (which converts Java to PVS or Isabelle) currently does not support the concept of threads. This is a clear limitation, especially when one considers that future applications will focus more and more on multi-threading because of the advent of multi-processor systems.

Appendix A

PVS: fiasco_types.pvs

```
%-----------------------------------------------------------------------------
% The datatypes in Fiasco that we use in our model are defined here, examples
% of those datatypes include threads, timeouts and thread states.
%-----------------------------------------------------------------------------
fiasco_types : THEORY
EXPORTING ALL WITH ALL
BEGIN
  % A pointer to a thread is just an integer, this reflects a C++ pointer also
  % being an integer.
  Thread_pointer: TYPE = int

  % Define the zero thread pointer, which represents the C++ NULL pointer.
  Zero_thread: Thread_pointer

  % The nil thread pointer points to a special thread (the nil thread) that
  % executes when no other threads are being executed.
  Nil_thread: Thread_pointer

  % This will be the thread used by the receiver as its partner, it is defined
  % as an uninterpreted constant.
  Receiver_partner_thread: Thread_pointer

  % The pointer to the current active thread (the <this> thread) must never be
  % the <Zero_thread> (as it points to nothing) and thus we have defined
  % a special type of thread pointer for the <this> thread which excludes the
  % thread pointer being equal to the <Zero_thread>.
  This_thread_pointer: TYPE = { tp: Thread_pointer | NOT tp = Zero_thread }

  % Most functions in Fiasco return an error code, of which there are many.
  % However, we are only interested in whether an error occured and thus the
  % error code is modelled as a boolean value.
  Ipc_err: TYPE = bool

  % Indicates if a timeout occured, additional details of timeouts such as
  % the exact time at which they expire are not of interest to us.
  L4_timeout: TYPE = bool

  % Enumeration of the possible preemption actions. Normally, a preemption
  % point might result in a thread switch which can then influence the ongoing
  % IPC we are modeling. However, we have abstracted away thread scheduling
  % (and thus the thread switch) and are thus only interested in the result
  % such a thread switch might have on the ongoing IPC. In our model, there
  % are five different effects a preemption point can have; these actions
  % are defined (and described briefly) in the enumeration below.
  %
  % NOTE: these actions are by no means complete, one of the actions we have
  % not modelled is a concurrent sender which also tries to send a message to
```

```
% the receiver.
Preemption_action: TYPE = {
                            Nothing,             % Nothing happens.
                            Kill,                % The partner is killed.
                            Timeout,             % A timeout occurs.
                            Sys_thread_ex_regs, % IPC is cancelled.
                            Receiver_ready       % The receiver becomes ready.
                          }

% When a preemption point is called, it can have five possible effects on the
% ongoing IPC (as described in the <Preemption_action> enumeration). However,
% it is not said that only a single <Preemption_action> occurs in a preemption
% point, it is possible that combinations of preemption actions take place.
% Therefore when a preemption point is called, it executes a (random) list of
% preemption actions which we define here.
Preemption_actions: TYPE = list[Preemption_action]

% Enumeration of the status the receiver's sender list can be in. We are not
% interested in the actual implementation of the sender list but only in the
% status it results in.
Sender_list: TYPE = {
                      First,     % <this> is the first sender in the list.
                      Enqueued, % <this> is enqueued but not the first item.
                      Dequeued  % <this> is not in the non-empty list.
                    }

% Enumeration of the states a sender list can be in, which is either an empty
% or non-empty state.
Sender_list_status: TYPE = {
                             Empty,
                             Non_empty
                           }

% In Fiasco the thread state is defined as a bit mask, but for clarity and
% simplicity we have chosen to model it as a record of boolean values
% where a field's boolean value reflects the corresponding bit being set.
%
% NOTE: our thread state does not contain all state bits that are defined
% in Fiasco, they form the minimal subset of state bits that is necessary
% for the properties we have modelled. One example of a state bit that we
% have not included is the <thread_polling_long> bit, as it is only used in
% long IPC and we have not modelled long IPC.
Thread_state: TYPE = [# thread_ready:    bool,
                        thread_cancel:   bool,
                        thread_dead:     bool,
                        thread_busy:     bool,
                        thread_invalid:  bool,
                        thread_polling:  bool,
                        thread_receiving: bool,
                        thread_ipc_in_progress: bool,
                        thread_send_in_progress: bool,
                        thread_transfer_in_progress: bool #]

% The context contains the thread state, here a lot of details have been
% abstracted away which mostly concern scheduling.
Context: TYPE = [# state: Thread_state #]

% The receiver role inherits from <Context> and adds two fields: <partner> and
% <sender_list>. The first reflects the fact that a receiver always has a
% partner associated with it and the second reflects the threads that want to
% engage in IPC with the receiver.
Receiver: TYPE = Context WITH [# partner: Thread_pointer,
                                 sender_list: Sender_list #]

% The sender role in our model does not have any real function, but we have
% included it for use in possible later extensions of our model.
```

```
  Sender: TYPE = [# #]

  % A Fiasco thread extends both the <Receiver>- and <Sender> types. The thread
  % also has a field <thread_lock>, which points to the locking thread. If a
  % thread is not locked, the <thread_lock> field is equal to the <Zero_thread>.
  Thread: TYPE = Sender WITH Receiver WITH [# thread_lock: Thread_pointer #]

  % The thread list is a function which translates thread pointers to threads;
  % this reflects the C++ concept of pointers to objects (in this case pointers
  % to threads).
  Thread_list: TYPE = [Thread_pointer -> Thread]

  % The system state represents the global state of the system. We had to
  % introduce a global state as C++ functions can have side-effects, which is not
  % supported by the the functional language PVS uses. Therefore, we had to
  % devise a solution which enabled the global state to be modified by each
  % function without using side-effects. Our solution was to have each function,
  % that has side-effects, receive- and return the global system state. The
  % global system state can then be accessed and modified by all those functions.
  System_state: TYPE =
            [#
              this:     This_thread_pointer, % Pointer to active thread.
              threads:  Thread_list,         % List of all threads.
              error:    Ipc_err,             % Indicates if error occurred.
              timeout:  L4_timeout,          % Indicates if timeout occurred.
              seed:     nat,                 % Seed used for randomness.
              handshake_attempted:   bool,   % Indicates if handshake attempted.
              assertions_held:       bool,   % Indicates if assertions held.
              receiver_initialized:  bool    % Receiver state initialized.
            #]

  % We use an uninterpreted thread state constant to define other thread states
  % with, for example the empty- and full thread states are derived of it.
  TS: Thread_state

END fiasco_types
```

Appendix B

PVS: fiasco_functions.pvs

```
%-----------------------------------------------------------------------------
% The Fiasco IPC implementation is basically a set of inter-related functions.
% Here we define our PVS representation of those functions. Although most
% functions can be mapped almost one-on-one to their C++ counterparts, some
% functions are actually simplified versions, which describe the behaviour of
% its Fiasco implementations at a more abstract level. There have also been
% functions added that are not in the C++ source, in all cases this has been
% done to split up larger functions into more compact parts, the functionality
% of course remains intact.
%-----------------------------------------------------------------------------
fiasco_functions : THEORY
EXPORTING ALL WITH ALL
BEGIN
  %---------------------------------------------------------------------------
  % Import the theories which will be used in our lemmas.
  %---------------------------------------------------------------------------
  IMPORTING fiasco_types
  IMPORTING fiasco_states

  %---------------------------------------------------------------------------
  % Functional specifications which will describe IPC in Fiasco.
  %---------------------------------------------------------------------------

  % Add bits to the state of a thread, all bits set in the <bits> thread state
  % will be set in the state of the thread pointed to by <tp>.
  state_add_dirty(tp: Thread_pointer, bits: Thread_state,
                  state_old: System_state): System_state =
    LET
      % Determine which state to work with.
      state = state_old`threads(tp)`state,
      state =  state WITH
        [
         thread_ready      := state`thread_ready      OR bits`thread_ready,
         thread_cancel     := state`thread_cancel     OR bits`thread_cancel,
         thread_dead       := state`thread_dead       OR bits`thread_dead,
         thread_busy       := state`thread_busy       OR bits`thread_busy,
         thread_invalid    := state`thread_invalid    OR bits`thread_invalid,
         thread_polling    := state`thread_polling    OR bits`thread_polling,
         thread_receiving  := state`thread_receiving  OR bits`thread_receiving,
         thread_ipc_in_progress      := state`thread_ipc_in_progress OR
                                        bits`thread_ipc_in_progress,
         thread_send_in_progress     := state`thread_send_in_progress OR
                                        bits`thread_send_in_progress,
         thread_transfer_in_progress := state`thread_transfer_in_progress OR
                                        bits`thread_transfer_in_progress
        ]
    IN
```

```
    state_old WITH [(threads)(tp)`state := state]

% Delete bits from the state of a thread, all bits set in the <bits> thread
% state will be unset in the state of the thread pointed to by <tp>.
state_del_dirty(tp: Thread_pointer, bits: Thread_state,
                state_old: System_state): System_state =
  LET
    % Determine which state to work with.
    state = state_old`threads(tp)`state,
    state = state WITH
    [
     thread_ready      := state`thread_ready      AND NOT bits`thread_ready,
     thread_cancel     := state`thread_cancel     AND NOT bits`thread_cancel,
     thread_dead       := state`thread_dead       AND NOT bits`thread_dead,
     thread_busy       := state`thread_busy       AND NOT bits`thread_busy,
     thread_invalid    := state`thread_invalid    AND NOT bits`thread_invalid,
     thread_polling    := state`thread_polling    AND NOT bits`thread_polling,
     thread_receiving  := state`thread_receiving  AND NOT bits`thread_receiving,
     thread_ipc_in_progress      := state`thread_ipc_in_progress AND NOT
                                    bits`thread_ipc_in_progress,
     thread_send_in_progress     := state`thread_send_in_progress AND NOT
                                    bits`thread_send_in_progress,
     thread_transfer_in_progress := state`thread_transfer_in_progress AND NOT
                                    bits`thread_transfer_in_progress
    ]
  IN
    state_old WITH [(threads)(tp)`state := state]

% Delete- and add bits to the state of the thread pointed to by <tp>; all bits
% not set in the <mask> thread state will first be unset and then all bits
% set in <bits> will be set.
state_change_dirty(tp: Thread_pointer, mask: Thread_state,
                   bits: Thread_state,state_old: System_state): System_state =
  LET
    % Determine which state to work with.
    state = state_old`threads(tp)`state,
    state = state WITH
        [
         thread_ready      := state`thread_ready      AND mask`thread_ready,
         thread_cancel     := state`thread_cancel     AND mask`thread_cancel,
         thread_dead       := state`thread_dead       AND mask`thread_dead,
         thread_busy       := state`thread_busy       AND mask`thread_busy,
         thread_invalid    := state`thread_invalid    AND mask`thread_invalid,
         thread_polling    := state`thread_polling    AND mask`thread_polling,
         thread_receiving  := state`thread_receiving  AND mask`thread_receiving,
         thread_ipc_in_progress      := state`thread_ipc_in_progress AND
                                        mask`thread_ipc_in_progress,
         thread_send_in_progress     := state`thread_send_in_progress AND
                                        mask`thread_send_in_progress,
         thread_transfer_in_progress := state`thread_transfer_in_progress AND
                                        mask`thread_transfer_in_progress
        ],

    state = state WITH
      [
       thread_ready      := state`thread_ready      OR bits`thread_ready,
       thread_cancel     := state`thread_cancel     OR bits`thread_cancel,
       thread_dead       := state`thread_dead       OR bits`thread_dead,
       thread_busy       := state`thread_busy       OR bits`thread_busy,
       thread_invalid    := state`thread_invalid    OR bits`thread_invalid,
       thread_polling    := state`thread_polling    OR bits`thread_polling,
       thread_receiving  := state`thread_receiving  OR bits`thread_receiving,
       thread_ipc_in_progress      := state`thread_ipc_in_progress OR
                                      bits`thread_ipc_in_progress,
       thread_send_in_progress     := state`thread_send_in_progress OR
                                      bits`thread_send_in_progress,
```

```
        thread_transfer_in_progress := state`thread_transfer_in_progress OR
                                       bits`thread_transfer_in_progress
      ]
  IN
    state_old WITH [(threads)(tp)`state := state]

% Delete bits from the state of a thread, all bits set in the <bits> thread
% state will be unset in the state of the thread pointed to by <tp>.
%
% NOTE: in our model this function is just an interface for the
% state_del_dirty() function. However, in Fiasco there is a real difference
% between the two functions as the dirty version assumes that the CPU lock
% is held, which allows for faster updating. As we have not modelled the CPU
% lock, the two functions are the same in our model. We have chosen to still
% include the state_del() function because it allows for a better reflection
% of the source code and the model might later be changed to allow CPU locks.
state_del(tp: Thread_pointer, bits: Thread_state,
          state_old: System_state): System_state =
  state_del_dirty(tp,bits,state_old)

% Delete- and add bits to the state of the thread pointed to by <tp>; all bits
% not set in the <mask> thread state will first be unset and then all bits
% set in <bits> will be set.
%
% NOTE: here we have a similar situation to the one described in the notes of
% the state_del() function, please check its notes above for more information.
state_change(tp: Thread_pointer, mask: Thread_state, bits: Thread_state,
             state_old: System_state): System_state =
  state_change_dirty(tp,mask,bits,state_old)

% Remove the thread lock on the partner thread by setting the thread lock
% owner to the zero thread.
clear_dirty(partner: Thread_pointer, state_old: System_state): System_state =
  state_old WITH [(threads)(partner)`thread_lock := Zero_thread]

% Remove the thread lock on the partner thread by setting the thread lock
% owner to the zero thread.
%
% NOTE: although this specification is equal to that of clear_dirty(), once
% again this is due to our model being a simplified version of the source
% code. In the source code, the clear_dirty() function possibly switches the
% execution context, whereas the clear_dirty_dont_switch() function obviously
% does not. As we do not model execution context switches, the result of the
% two functions is the same. We have still included both functions because of
% the reasons also specified in the notes of the state_del() function.
clear_dirty_dont_switch(partner: Thread_pointer,
                        state_old: System_state): System_state =
  state_old WITH [(threads)(partner)`thread_lock := Zero_thread]

% Set the thread lock of the partner thread to the <this> thread.
%
% NOTE: this is a typical function in which a lot of implementation details
% have been abstracted away. In the C++ implementation, the function first
% checks if another thread has locked the partner and if so, it enters a loop
% that waits for the lock to be released. The end result is always that the
% lock is acquired by the <this> thread so this lead to our simplified version.
lock_dirty(partner: Thread_pointer,state_old: System_state): System_state =
  state_old WITH [(threads)(partner)`thread_lock := state_old`this]

% Indicates if a partner is ready for a sender.
sender_ok(sender: Thread_pointer, receiver: Thread_pointer,
          state_old: System_state): bool =
  % The receiver should at least be in a receiving IPC state.
  IF NOT state_old`threads(receiver)`state`thread_receiving OR
     NOT state_old`threads(receiver)`state`thread_ipc_in_progress
  THEN
```

```
    false
  % Open wait: no partner should be specified and the sender should be the
  % first in the receiver's sender list.
  ELSIF state_old`threads(receiver)`partner = Zero_thread AND
        state_old`threads(receiver)`sender_list = First THEN
    true
  % Closed wait: check if the sender is the same as the partner specified.
  ELSIF sender = state_old`threads(receiver)`partner THEN
    true
  ELSE
    false
  ENDIF

% Set the partner of the receiver to the specified sender.
set_partner(sender: Thread_pointer, receiver: Thread_pointer,
            state_old: System_state): System_state =
  state_old WITH [(threads)(receiver)`partner := sender]

% Initialize IPC between the sender and receiver.
ipc_init(sender: Thread_pointer, receiver: Thread_pointer,
         state_old: System_state): System_state =
  % Set the <thread_transfer_in_progress> bit on the receiver state.
  LET
    % Set the partner.
    state_temp = set_partner(sender,receiver,state_old),
    bits = TS_empty WITH [thread_transfer_in_progress := true]
  IN
    state_add_dirty(receiver,bits,state_temp)

% Transfer a message between the sender and receiver.
%
% NOTE: this function completely ignores the actual copying of the message,
% we assume that the copying will always succeed. What's more, this function
% does not do anything at all because we totally abstracted away the sending
% of the message.
transfer_msg(receiver: Thread_pointer,state_old: System_state): System_state =
  state_old

% Indicates if the receiver is engaged in IPC with the sender.
in_ipc(sender: Thread_pointer, receiver: Thread_pointer,
       state_old: System_state): bool =
  state_old`threads(receiver)`state`thread_transfer_in_progress AND
  state_old`threads(receiver)`state`thread_ipc_in_progress AND NOT
  state_old`threads(receiver)`state`thread_cancel AND
  state_old`threads(receiver)`partner = sender

% Wakeup the receiver, which ensures that the state is changed to reflect that
% it is no longer involved in IPC and is ready to continue with other tasks.
wake_receiver(receiver:Thread_pointer, state_old:System_state): System_state =
  % Define the mask and bits to change the state of the receiver.
  LET mask = TS_full WITH [thread_busy := false,
                           thread_receiving := false,
                           thread_transfer_in_progress := false,
                           thread_ipc_in_progress := false],
      bits = TS_empty WITH [thread_ready := true]
  IN
    state_change_dirty(receiver,mask,bits,state_old)

% Return a (random) sender list status.
%
% NOTE: the function is defined without an implementation, that means that we
% cannot predict the return value of the function. However, as it still is a
% function that means that equal input parameters result in the same return
% values. Therefore, to really achieve the randomness we are looking for we
% need to make sure that the input parameter never has the same value twice.
sender_list_status(n: nat): Sender_list_status
```

```
% Enqueue the <this> thread in the receiver's sender list.
sender_enqueue(receiver:Thread_pointer,state_old:System_state): System_state =
  LET
    % Update the seed and get a random sender list status.
    state_old = state_old WITH [seed := state_old`seed + 1],
    status = sender_list_status(state_old`seed)
  IN
  % Check if the sender list is empty, if so the <this> thread will become
  % the first item in the sender list and otherwise it will be set to an
  % enqueued status (indicating that it is in the sender list but is not
  % the first item).
  IF status = Empty THEN
    state_old WITH [(threads)(receiver)`sender_list := First]
  ELSE
    state_old WITH [(threads)(receiver)`sender_list := Enqueued]
  ENDIF

% Dequeue the <this> thread from the receiver's sender list.
sender_dequeue(receiver:Thread_pointer,state_old:System_state): System_state =
  state_old WITH [(threads)(receiver)`sender_list := Dequeued]

% Dequeue the <this> thread from the receiver's sender list.
sender_dequeue_head(receiver: Thread_pointer,
                    state_old: System_state): System_state =
  state_old WITH [(threads)(receiver)`sender_list := Dequeued]

% Indicates if the sender is in the receiver's sender list.
in_sender_list(receiver: Thread_pointer, state_old: System_state): bool =
  NOT state_old`threads(receiver)`sender_list = Dequeued

% Prepare a thread for receiving a message from the specified sender.
prepare_receive_dirty(sender: Thread_pointer, receiver: Thread_pointer,
                      state_old: System_state): System_state =
  LET
    % Set the partner.
    state_temp = set_partner(sender,receiver,state_old),

    mask = TS_full WITH [thread_send_in_progress := false,
                         thread_polling := false,
                         thread_transfer_in_progress := false],
    bits = TS_empty WITH [thread_receiving := true,
                          thread_ipc_in_progress := true]
  IN
    state_change_dirty(receiver,mask,bits,state_temp)

% Kill the partner thread.
kill(partner: Thread_pointer, state_old: System_state): System_state =
  % Both locked- and invalid threads cannot be killed
  IF NOT state_old`threads(partner)`thread_lock = state_old`this AND
     NOT state_old`threads(partner)`state`thread_invalid
  THEN
    state_old WITH[(threads)(partner)`state`thread_dead := true]
  ELSE
    state_old
  ENDIF

% Update the current thread's registers, which can be used to create a new
% thread out of an existing one. This results in the <thread_cancel> bit
% of the <this> thread being set, signifying that ongoing IPC should be
% cancelled.
sys_thread_ex_regs(tp: Thread_pointer,state_old: System_state): System_state =
  state_old WITH [(threads)(tp)`state`thread_cancel := true]

% An IPC timeout has occured, which means that the specified timeout time has
% passed. We simulate this by setting the <timeout> member to true.
```

```
timeout(tp: Thread_pointer, state_old: System_state): System_state =
 LET
   % Define the state changed by the timeout.
   mask = TS_full WITH [thread_ipc_in_progress := false],
   bits = TS_empty WITH [thread_ready := true]
 IN
   state_change(tp,mask,bits,state_old) WITH [timeout := true]

% Called by the receiver on a sender to determine if both are ready to engage
% in IPC.
ipc_receiver_ready(sender: Thread_pointer, receiver: Thread_pointer,
                   state_old: System_state): bool =
  state_old`threads(sender)`state`thread_ipc_in_progress AND
  sender_ok(sender,receiver,state_old)

% In this function the side-effects of the ipc_receiver_ready() function are
% handled.
%
% NOTE: in the Fiasco source code the receiver calls the ipc_receiver_ready()
% function, which returns a boolean value indicating if the sender was ready
% for the receiver. However, this function also has side-effects if the
% sender was indeed ready so we had to split up the function in one in which
% a boolean value was returned and one in which the side-effects are handled.
ipc_receiver_ready_change(sender: Thread_pointer, receiver: Thread_pointer,
                          state_old: System_state): System_state =
  LET
    % Initialize IPC.
    state_temp = ipc_init(sender,receiver,state_old),

    % Set the sender to ready and indicate that the transfer is in progress.
    bits = TS_empty WITH [thread_ready := true,
                          thread_transfer_in_progress := true],
    state_temp = state_add_dirty(sender,bits,state_temp),

    % Remove the ready state from the receiver.
    bits = TS_empty WITH [thread_ready := true]
  IN
    state_del_dirty(receiver,bits,state_temp)

% This preemption action signifies a receiver becoming ready to receive a
% message from a sender.
receiver_ready(sender: Thread_pointer, receiver: Thread_pointer,
               state_old: System_state): System_state =
  LET
    state_old =
      % Check if the receiver has been initialized.
      IF NOT state_old`receiver_initialized THEN
        % Prepare the receiver using a 'random' partner.
        prepare_receive_dirty(Receiver_partner_thread,receiver,state_old)
          WITH [(receiver_initialized) := true]
      ELSE
        state_old
      ENDIF
  IN
  % Make sure the receiver is in the right state to receive from a sender.
  IF state_old`threads(receiver)`state`thread_receiving AND
     state_old`threads(receiver)`state`thread_ipc_in_progress AND NOT
     state_old`threads(receiver)`state`thread_cancel
  THEN
    % Closed wait: check if the sender is the same as the partner specified.
    IF sender = state_old`threads(receiver)`partner THEN
      % Check if the sender meets the conditions of the receiver
      IF in_sender_list(receiver,state_old) AND
         ipc_receiver_ready(sender,receiver,state_old)
      THEN
        % Apply the changes caused by the side-effects of the
```

```
        % ipc_receiver_ready() function.
        ipc_receiver_ready_change(sender,receiver,state_old)
      ELSE
        state_old
      ENDIF
    % Open wait: no partner should be specified and the sender should be
    % the first item.
    ELSE
      % Check if the sender is not ready in which case it should be removed
      % from the sender list.
      IF NOT ipc_receiver_ready(sender,receiver,state_old) THEN
        sender_dequeue_head(receiver,state_old)
      ELSE
        % Apply the changes caused by the side-effects of the
        % ipc_receiver_ready() function.
        ipc_receiver_ready_change(sender,receiver,state_old)
      ENDIF
    ENDIF
  ELSE
    state_old
  ENDIF

% Return a (random) preemption action.
%
% NOTE: the randomness achieved in the same way as described in the notes of
% the sender_list_status() function.
preemption_action(n: nat): Preemption_actions

% Execute a list of preemption point actions.
preemption_point_actions(actions: Preemption_actions, partner: Thread_pointer,
                         allow_timeout: bool,
                         state_old: System_state): RECURSIVE System_state =
  CASES actions OF
    null: state_old,
    cons(action,remaining_actions):
    LET state_temp =
      % Check if a call was made to sys_thread_ex_regs().
      IF action = Sys_thread_ex_regs THEN
        sys_thread_ex_regs(state_old`this,state_old)
      % Only allow a timeout to occur when specified, most of the time a
      % timeout cannot occur.
      ELSIF action = Timeout AND allow_timeout THEN
        timeout(state_old`this,state_old)
      % Check if the partner is killed.
      ELSIF action = Kill THEN
        kill(partner,state_old)
      % A receiver can only become ready when it is not equal to the sender,
      % if so it would be waiting forever.
      ELSIF action = Receiver_ready AND NOT state_old`this = partner THEN
        receiver_ready(state_old`this,partner,state_old)
      ELSE
        state_old
      ENDIF
    IN
    % Execute the remaining actions.
    preemption_point_actions(remaining_actions,partner,allow_timeout,state_temp)
  ENDCASES
  MEASURE length(actions)

% Simulate a preemption point by ignoring the actual scheduling occuring and
% focus only on the effects that scheduling might have on the system state.
% To achieve this, a list of random preemption actions is created, which are
% then executed and each executed action updates the system state.
preemption_point(partner: Thread_pointer, allow_timeout: bool,
                 state_old: System_state): System_state =
  LET
```

```
    % Increment the seed and use it to get a random list of preemption
    % actions.
    state_old = state_old WITH [seed := state_old`seed + 1],
    actions = preemption_action(state_old`seed)
 IN
    % Execute the preemption actions.
    preemption_point_actions(actions,partner,allow_timeout,state_old)

% Abort the IPC send operation.
abort_send(partner: Thread_pointer, state_old: System_state): System_state =
  LET
    % Unset the bits that indicate that the sender was busy in IPC.
    bits = TS_empty WITH [thread_send_in_progress := true,
                          thread_polling := true,
                          thread_ipc_in_progress := true,
                          thread_transfer_in_progress := true],
    state_temp = state_del_dirty(state_old`this,bits,state_old),

    state_temp = preemption_point(partner,false,state_temp),

    state_temp =
      % Check if the sender needs to be dequeued from the sender list.
      IF in_sender_list(partner,state_temp) THEN
        sender_dequeue(partner,state_temp)
      ELSE
        state_temp
      ENDIF,

    state_temp = preemption_point(partner,true,state_temp)
  IN
    % Remove the thread lock on the receiver.
    clear_dirty(partner,state_temp) WITH [error := true]

% Handle the finishing part of the do_send_wait() function.
do_send_wait_finish(partner: Thread_pointer,
                    state_old: System_state): System_state =
  LET
    state_temp =
      % If a timeout has hit at this point, it can be safely ignored as the
      % handshake has already taken place. However, make sure that the
      %<thread_ipc_in_progress> bit of the <this> thread is set once again
      % as the timeout will have unset it.
      IF state_old`timeout THEN
      LET
        bits_add = TS_empty WITH [thread_ipc_in_progress := true]
      IN
        state_add_dirty(state_old`this,bits_add,state_old)
      ELSE
        state_old
      ENDIF,

    % Add preemption points and dequeue the sender from the sender list.
    state_temp = preemption_point(partner,false,state_temp),
    state_temp = sender_dequeue(partner,state_temp),
    state_temp = preemption_point(partner,false,state_temp),

    % Remove the polling bit as the handshake has ended.
    bits_del = TS_empty WITH [thread_polling := true],
    state_temp = state_del_dirty(state_temp`this,bits_del,state_temp)
  IN
    % Check if IPC was cancelled.
    IF state_temp`threads(state_temp`this)`state`thread_cancel THEN
      % Check if the sender- and receiver are still engaged in IPC, if so the
      % state of the receiver needs to be changed.
      IF NOT in_ipc(state_temp`this,partner,state_temp) THEN
        abort_send(partner,state_temp)
```

```
      ELSE
      LET
        mask = TS_full WITH [thread_ipc_in_progress := false],
        bits = TS_empty WITH [thread_cancel := true, thread_ready := true],
        state_temp = state_change_dirty(partner,mask,bits,state_temp)
      IN
        abort_send(partner,state_temp)
      ENDIF
    % Abort the send if the sender is not in IPC with the receiver or the
    % receiver has been killed.
    ELSIF NOT in_ipc(state_temp`this,partner,state_temp) OR
          state_temp`threads(partner)`state`thread_dead
    THEN
      abort_send(partner,state_temp)
    ELSE
      % Assert that the sender is not engaged in IPC with the receiver and
      % that the handshake bit is not set.
      state_temp WITH [(assertions_held) :=
                    state_temp`assertions_held AND
      state_temp`threads(state_temp`this)`state`thread_ipc_in_progress AND NOT
                    in_sender_list(partner,state_temp),
                    error := false]
    ENDIF

% Handle the waiting loop part of the do_send_wait() function, which waits for
% the receiver to become ready for the sender.
do_send_wait_loop(partner: Thread_pointer,
                  state_old: System_state): System_state =
  LET
    % Remove the thread lock on the partner as we have to give it a chance in
    % the preemption point to get in the state in which it is ready to receive
    % a message from the <this> thread. Afterwards acquire the thread lock
    % again.
    state_temp = clear_dirty(partner,state_old),
    state_temp = preemption_point(partner,true,state_temp),
    state_temp = lock_dirty(partner,state_temp)
  IN
  % Check if IPC was cancelled or the handshake bit was set, if so finish the
  % do_send_wait() call.
  IF state_temp`threads(state_temp`this)`state`thread_cancel OR
     state_temp`threads(state_temp`this)`state`thread_transfer_in_progress
  THEN
    do_send_wait_finish(partner,state_temp)
  ELSE
  LET
    % Assert that the sender is not engaged in IPC with the receiver and that
    % the handshake bit is not set.
    state_temp = state_temp WITH [(assertions_held) :=
                    state_temp`assertions_held AND NOT
                    in_ipc(state_temp`this,partner,state_temp) AND NOT
      state_temp`threads(state_temp`this)`state`thread_transfer_in_progress]
  IN
    % Check if a timeout has occured, if so abort the sending.
    IF state_temp`timeout THEN
    LET
      % Assert that the sender is not engaged in IPC with the receiver and
      % that the handshake bit is not set.
      state_temp = state_temp WITH [(assertions_held) :=
             state_temp`assertions_held AND NOT
             state_temp`threads(state_temp`this)`state`thread_ipc_in_progress]
    IN
      abort_send(partner,state_temp)
    % Abort the send if the partner was killed.
    ELSIF state_temp`threads(partner)`state`thread_dead THEN
      abort_send(partner,state_temp)
    ELSE
```

```
    LET
      % Assert that the sender is still engaged in the handshake with the
      % receiver.
      state_temp = state_temp WITH [(assertions_held) :=
            state_temp`assertions_held AND
            state_temp`threads(state_temp`this)`state`thread_ipc_in_progress],

      % Set the polling state
      bits = TS_empty WITH [thread_polling := true],
      state_temp = state_add_dirty(state_temp`this,bits,state_temp)
    IN
      do_send_wait_finish(partner,state_temp)
    ENDIF
  ENDIF

% Wait for the receiver to become ready for the sender.
do_send_wait(partner: Thread_pointer,state_old: System_state): System_state =
  LET
    % Assert that the no other thread but the zero- or <this> thread hold the
    % thread lock on the receiver.
    state_temp = state_old WITH [(assertions_held) :=
                   state_old`assertions_held AND
                  (state_old`threads(partner)`thread_lock = Zero_thread OR
                   state_old`threads(partner)`thread_lock = state_old`this)],

    % Create a thread state which bits to add to the <this> thread.
    add_bits = TS_empty WITH [thread_polling := true,
                              thread_send_in_progress := true,
                              thread_ipc_in_progress := true],
    state_temp = state_add_dirty(state_temp`this,add_bits,state_temp),

    % Lock the partner before continuing.
    state_temp = lock_dirty(partner,state_temp),

    % Add a preemption point
    state_temp = preemption_point(partner,false,state_temp)
  IN
    % Check if IPC was cancelled.
    IF state_temp`threads(state_temp`this)`state`thread_cancel THEN
    LET
      % Remove the thread lock on the receiver.
      state_temp = clear_dirty(partner,state_temp)
    IN
      % Delete the bits from the <this> thread's state.
      state_del_dirty(state_temp`this,TS_ipc_end_mask,
                      state_temp) WITH [error := true]
    ELSE
      % Return if the receiver is ready to receive a message from the <this>
      % thread. If not enter the waiting loop.
      IF sender_ok(state_temp`this,partner,state_temp) THEN
        state_temp WITH [error := false]
      ELSE
      LET
        % Enqueue the sender in the sender list of the receiver.
        state_temp = sender_enqueue(partner,state_temp),

         % Give the receiver time to become ready for the sender.
        state_temp = preemption_point(partner,true,state_temp)
      IN
        % Abort the sending if a timeout occured.
        IF state_temp`timeout THEN
          abort_send(partner,state_temp)
        ELSE
          do_send_wait_loop(partner,state_temp)
        ENDIF
      ENDIF
```

```
    ENDIF

% Try to have the sender and receiver agree upon engaging in IPC.
try_handshake_receiver(partner: Thread_pointer,
                       state_old: System_state): System_state =
  % Immediately return with an error if the partner is the Zero- or Nil thread
  % or the partner is an invalid thread.
  IF partner = Zero_thread OR
     state_old`threads(partner)`state`thread_invalid OR
     partner = Nil_thread
  THEN
    state_old WITH [error := true]
  ELSE
  LET
    % Set the <handshake_attempted> field to true because the receiver is
    % valid.
    state_temp = state_old WITH [handshake_attempted := true]
  IN
    LET state_temp =
      % If a thread lock is already held on the receiver, acquire the thread
      % lock (after a preemption point has been set).
      IF NOT state_temp`threads(partner)`thread_lock = Zero_thread THEN
      LET state_temp = preemption_point(partner,false,state_temp) IN
        lock_dirty(partner,state_temp)
      ELSE
        state_temp
      ENDIF
    IN
      % Check if the result of the preemption point was that IPC has been
      % canceled, if so remove the thread lock and return an error.
      IF state_temp`threads(state_temp`this)`state`thread_cancel THEN
        clear_dirty(partner,state_temp) WITH [error := true]
      ELSE
        % IPC has not been cancelled, now check if the receiver is ready for
        % the sender. If not, wait for the receiver to become ready.
        IF NOT sender_ok(state_temp`this,partner,state_temp) THEN
        LET
          state_temp = do_send_wait(partner,state_temp)
        IN
          % Assert that the sender does not hold a thread lock on the receiver
          % when an error occured.
          state_temp WITH [(assertions_held) :=
             state_temp`assertions_held AND
            (NOT state_temp`error OR
             NOT state_temp`threads(partner)`thread_lock = state_temp`this)]
        ELSE
          state_temp WITH [error := false]
        ENDIF
      ENDIF
  ENDIF

% Handle the send part in the do_ipc() function.
do_ipc_send_part(partner: Thread_pointer, have_receive: bool,
                 state_old: System_state): System_state =
  % Start with a handshake and immediately return if an error occured.
  LET state_temp = try_handshake_receiver(partner,state_old) IN
  IF state_temp`error THEN
    state_temp
  ELSE
  LET
    % Assert that after a successful handshake the sender thread is no
    % longer polling.
    state_temp = state_temp WITH [(assertions_held) :=
                   state_temp`assertions_held AND
                NOT state_temp`threads(state_temp`this)`state`thread_polling],
```

```
    % Initialize IPC.
    state_temp = ipc_init(state_temp`this,partner,state_temp),

    % Transfer the IPC message.
    state_temp = transfer_msg(partner,state_temp),

    % Assert that the thread lock on the receiver is held by the <this> thread
    % or that there is no thread lock.
    state_temp = state_temp WITH [(assertions_held) :=
                    state_temp`assertions_held AND
                   (state_temp`threads(partner)`thread_lock = Zero_thread OR
                    state_temp`threads(partner)`thread_lock = state_temp`this)]
  IN
    % Check if an error occured or that there is no receive part.
    IF state_temp`error OR NOT have_receive THEN
    LET
      state_new =
        % If the partner is still engaged in IPC, wake him up.
        IF in_ipc(state_temp`this,partner,state_temp) THEN
          wake_receiver(partner,state_temp)
        ELSE
          state_temp
        ENDIF,
      % Remove the thread lock on the partner, adjust the state of the <this>
      % thread and return the error state of the message transfer.
      state_new = clear_dirty(partner,state_new)
    IN
      state_del(state_new`this,TS_ipc_end_mask,
                state_new) WITH [error := state_temp`error]
    ELSE
    % Remove the thread lock on the partner.
    LET state_temp = clear_dirty_dont_switch(partner,state_temp) IN
      % Return an error if the partner is not engaged in IPC with us. This is
      % because we only arrive here if no error occured during message
      % transfer and there is a receive part, which means that the partner
      % has yet to send a message to us and should therefore still be engaged
      % in IPC with us. If the partner is still in IPC with us, wake him up
      % to enable him to send his message.
      IF NOT in_ipc(state_temp`this,partner,state_temp) THEN
        state_del(state_temp`this,TS_ipc_end_mask,
                  state_temp) WITH [error := true]
      ELSE
        wake_receiver(partner,state_temp)
      ENDIF
    ENDIF
  ENDIF

% Handle the receive part in the do_ipc() function.
do_ipc_receive_part(sender: Thread_pointer, have_receive: bool,
                    state_old: System_state): System_state =
  state_old WITH [(assertions_held) :=
                  state_old`assertions_held AND
                  have_receive]

% Handle IPC between the sender- and receiver.
do_ipc(have_send: bool, partner: Thread_pointer, have_receive: bool,
       sender: Thread_pointer, state_old: System_state): System_state =
  LET state_temp =
    % Check if there is a send part.
    IF have_send THEN
      do_ipc_send_part(partner,have_receive,state_old)
    ELSE
      state_old
    ENDIF
  IN
    % If no error occured in the send part and there is a receive part,
```

```
      % enter the receive part and otherwise return.
      IF have_receive AND NOT state_temp`error THEN
        do_ipc_receive_part(sender,have_receive,state_temp)
      ELSE
        state_temp
      ENDIF

  % Execute an IPC call.
  %
  % NOTE: we tried to make as few assumptions as possible on the system state
  % but there are some fields that have to be initialized before engaging in
  % IPC. Naturally, before an IPC call is made there has not been any error,
  % timeout, handshake attempted. Furthermore initially the assertions all hold.
  sys_ipc(have_send: bool, partner: Thread_pointer,
          have_receive: bool, sender: Thread_pointer,
          state_old: System_state): System_state =
    LET state_temp = state_old WITH [(error)    := false,
                                     (timeout)  := false,
                                     (handshake_attempted)  := false,
                                     (assertions_held)      := true,
                                     (receiver_initialized) := false]
    IN
      % Make sure there is either a send- or receive part, if so start IPC.
      IF have_send OR have_receive THEN
        do_ipc(have_send,partner,have_receive,sender,state_temp)
      ELSE
        state_temp WITH [(error) := true]
      ENDIF

END fiasco_functions
```

Appendix C

PVS: fiasco_states.pvs

```
%-----------------------------------------------------------------------------
% In Fiasco there are several predefined thread states, here we model those
% states. As our representation of the thread state as a record of boolean
% values does not allow for easy bitwise combinining, we also created an empty-
% and full thread state, in which none- respectively all bits are set, to allow
% for easy creation of thread state masks.
%-----------------------------------------------------------------------------
fiasco_states: THEORY
EXPORTING ALL WITH ALL
BEGIN
  %---------------------------------------------------------------------------
  % Import the theories which will be used in our lemmas.
  %---------------------------------------------------------------------------
  IMPORTING fiasco_types

  %---------------------------------------------------------------------------
  % Declare the state combinations that are used in Fiasco, which allow for easy
  % updating of the state of a thread.
  %---------------------------------------------------------------------------

  % The empty thread state has all state bits set to false.
  %
  % NOTE: this declaration cannot be found in the Fiasco source code, but has
  % been defined by us to allow for easy creation of thread states with
  % specific bits set.
  TS_empty: Thread_state = TS WITH[thread_ready     := false,
                                   thread_cancel    := false,
                                   thread_dead      := false,
                                   thread_busy      := false,
                                   thread_invalid   := false,
                                   thread_polling   := false,
                                   thread_receiving := false,
                                   thread_ipc_in_progress := false,
                                   thread_send_in_progress := false,
                                   thread_transfer_in_progress := false]

  % The full thread state has all state bits set to true.
  %
  % NOTE: this declaration cannot be found in the Fiasco source code, but has
  % been defined by us to allow for easy creation of thread states with
  % specific bits unset.
  TS_full: Thread_state = TS WITH[thread_ready     := true,
                                  thread_cancel    := true,
                                  thread_dead      := true,
                                  thread_busy      := true,
                                  thread_invalid   := true,
                                  thread_polling   := true,
```

```
                                thread_receiving := true,
                                thread_ipc_in_progress := true,
                                thread_send_in_progress := true,
                                thread_transfer_in_progress := true]

  % This state mask indicates that a thread is sending an IPC message.
  TS_ipc_sending_mask: Thread_state = TS_empty WITH [
                                        thread_send_in_progress := true,
                                        thread_polling := true
                                      ]

  % This state mask indicates that a thread is receiving an IPC message.
  TS_ipc_receiving_mask: Thread_state = TS_empty WITH [
                                          thread_busy := true,
                                          thread_receiving := true,
                                          thread_transfer_in_progress := true
                                        ]

  % This state mask is a combination of the <TS_ipc_sending_mask> and
  % <TS_ipc_receiving_mask> masks with the addition of the
  % <thread_ipc_in_progress> bit being set.
  TS_ipc_mask: Thread_state = TS_ipc_receiving_mask WITH [
                                thread_ipc_in_progress := true,
                                thread_send_in_progress := true,
                                thread_polling := true
                              ]

  % This mask is the sending mask with bits set that indicate that a
  % transfer- and IPC is in progress.
  %
  % NOTE: this mask is not one that is actually defined in the Fiasco source
  % code, but as it is used in several locations we decided to create a
  % single-point-of-definition which has become this thread state mask.
  TS_ipc_end_mask: Thread_state = TS_ipc_sending_mask WITH [
                                    thread_transfer_in_progress := true,
                                    thread_ipc_in_progress := true
                                  ]

END fiasco_states
```

Appendix D

PVS: fiasco_helpers.pvs

```
%-----------------------------------------------------------------------------
% There are many lemmas that are used in more than one of our property theories,
% for these lemmas we have created this separate theory, which other theories
% can easily import and use in their proofs.
%-----------------------------------------------------------------------------
fiasco_helpers: THEORY
BEGIN
  %---------------------------------------------------------------------------
  % Import the theories which will be used in our lemmas.
  %---------------------------------------------------------------------------
  IMPORTING fiasco_functions

  %---------------------------------------------------------------------------
  % Legend:
  %   LEMMA %- : the lemma depends on an assumption.
  %   LEMMA %/ : the lemma depends on an axiom.
  %---------------------------------------------------------------------------

  %---------------------------------------------------------------------------
  % Lemmas which involve the <sender_list> field of the receiver.
  %---------------------------------------------------------------------------

  % When the <this> thread is initially dequeued from the sender list, that will
  % also be the case after calling timeout().
  timeout_dequeued_unchanged: LEMMA
    FORALL(receiver: Thread_pointer, state_old: System_state):
      LET state_new = timeout(state_old`this, state_old) IN
        state_old`threads(receiver)`sender_list = Dequeued IMPLIES
          state_new`threads(receiver)`sender_list = Dequeued

  % When the <this> thread is initially dequeued from the sender list, that will
  % also be the case after calling kill().
  kill_dequeued_unchanged: LEMMA
    FORALL(receiver: Thread_pointer, state_old: System_state):
      LET state_new = kill(receiver, state_old) IN
        state_old`threads(receiver)`sender_list = Dequeued IMPLIES
          state_new`threads(receiver)`sender_list = Dequeued

  % When the <this> thread is initially dequeued from the sender list, that will
  % also be the case after calling receiver_ready().
  receiver_ready_dequeued_unchanged: LEMMA
   FORALL(sender: Thread_pointer, receiver: Thread_pointer, state_old: System_state):
     LET state_new = receiver_ready(state_old`this, receiver, state_old) IN
       state_old`threads(receiver)`sender_list = Dequeued IMPLIES
          state_new`threads(receiver)`sender_list = Dequeued

  % When the <this> thread is initially dequeued from the sender list, that will
```

```
% also be the case after calling preemption_point_actions().
preemption_point_actions_dequeued_unchanged: LEMMA
  FORALL (actions: Preemption_actions, allow_timeout: bool,
          partner: Thread_pointer, state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,
                                             allow_timeout,state_old) IN
      state_old`threads(partner)`sender_list = Dequeued IMPLIES
        state_new`threads(partner)`sender_list = Dequeued

% When the <this> thread is initially dequeued from the sender list, that will
% also be the case after calling preemption_point().
preemption_point_dequeued_unchanged: LEMMA
  FORALL (partner:Thread_pointer,allow_timeout:bool,state_old:System_state):
    LET state_new = preemption_point(partner,allow_timeout,state_old) IN
      state_old`threads(partner)`sender_list = Dequeued IMPLIES
        state_new`threads(partner)`sender_list = Dequeued

%-------------------------------------------------------------------------------
% Lemmas which involve the status of the <timeout> system state field.
%-------------------------------------------------------------------------------

% The <timeout> field is not changed by calling sender_enqueue().
sender_enqueue_timeout_unchanged: LEMMA
  FORALL (receiver:Thread_pointer,state_old: System_state):
    LET state_new = sender_enqueue(receiver,state_old) IN
      state_old`timeout = state_new`timeout

% The <timeout> field is not changed by calling kill().
kill_timeout_unchanged: LEMMA
  FORALL(receiver:Thread_pointer,state_old:System_state):
    LET state_new = kill(receiver,state_old) IN
      state_old`timeout = state_new`timeout

% The <timeout> field is not changed by calling receiver_ready().
receiver_ready_timeout_unchanged: LEMMA
 FORALL(sender:Thread_pointer,receiver:Thread_pointer,state_old:System_state):
   LET state_new = receiver_ready(state_old`this,receiver,state_old) IN
     state_old`timeout = state_new`timeout

% When no timeouts are allowed, the <timeout> value will not be changed by
% calling preemption_point_actions().
preemption_point_actions_timeout_unchanged: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,
                                             false,state_old) IN
      state_old`timeout = state_new`timeout

% When no timeouts are allowed, the <timeout> value will not be changed by
% calling preemption_point().
preemption_point_timeout_unchanged: LEMMA
  FORALL (partner:Thread_pointer,state_old: System_state):
    LET state_new = preemption_point(partner,false,state_old) IN
      state_old`timeout = state_new`timeout

% When initially the <timeout> field is set, that will still be the case after
% calling preemption_point_actions().
preemption_point_actions_timeout: LEMMA
  FORALL (actions: Preemption_actions, allow_timeout: bool,
          partner: Thread_pointer, state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,
                                             allow_timeout,state_old) IN
      state_old`timeout IMPLIES state_new`timeout

% When initially the <timeout> field is set, that will still be the case after
% calling preemption_point().
```

```
preemption_point_timeout: LEMMA
  FORALL (partner: Thread_pointer, allow_timeout: bool, state_old: System_state):
    LET state_new = preemption_point(partner, allow_timeout, state_old) IN
      state_old`timeout IMPLIES state_new`timeout

%-----------------------------------------------------------------------------
% Lemmas which involve the value of the <error> system state field.
%-----------------------------------------------------------------------------

% The sender_enqueue() function leaves the <error> field unmodified.
sender_enqueue_error_unchanged: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = sender_enqueue(partner, state_old) IN
      state_old`error = state_new`error

% The timeout() function leaves the <error> field unmodified.
timeout_error_unchanged: LEMMA
  FORALL(state_old: System_state):
    LET state_new = timeout(state_old`this, state_old) IN
      state_old`error = state_new`error

% The kill() function leaves the <error> field unmodified.
kill_error_unchanged: LEMMA
  FORALL(receiver: Thread_pointer, state_old: System_state):
    LET state_new = kill(receiver, state_old) IN
      state_old`error = state_new`error

% The ipc_receiver_ready_change() function leaves the <error> field unmodified.
ipc_receiver_ready_change_error_unchanged: LEMMA
 FORALL (partner: Thread_pointer, state_old: System_state):
  LET state_new = ipc_receiver_ready_change(state_old`this, partner, state_old)IN
     state_old`error = state_new`error

% The receiver_ready() function leaves the <error> field unmodified.
receiver_ready_error_unchanged: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = receiver_ready(state_old`this, partner, state_old) IN
      state_old`error = state_new`error

% The preemption_point_actions() function leaves the <error> field unmodified.
preemption_point_actions_error_unchanged: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          have_timeout: bool, state_old: System_state):
    LET state_new = preemption_point_actions(actions, partner,
                                             have_timeout, state_old) IN
      state_old`error = state_new`error

% The preemption_point() function leaves the <error> field unmodified.
preemption_point_error_unchanged: LEMMA
  FORALL (partner: Thread_pointer, have_timeout: bool, state_old: System_state):
    LET state_new = preemption_point(partner, have_timeout, state_old) IN
      state_old`error = state_new`error

%-----------------------------------------------------------------------------
% Lemmas which involve the value of the <this> system state field.
%-----------------------------------------------------------------------------

% Indicate that the <this> field is not changed by calling sender_dequeue().
sender_dequeue_this_unchanged: LEMMA
  FORALL(partner: Thread_pointer, state_old: System_state):
    LET state_new = sender_dequeue(partner, state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling sender_enqueue().
sender_enqueue_this_unchanged: LEMMA
  FORALL(partner: Thread_pointer, state_old: System_state):
```

```
    LET state_new = sender_enqueue(partner,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling
% prepare_receive_dirty().
prepare_receive_dirty_this_unchanged: LEMMA
  FORALL(sender:Thread_pointer,receiver:Thread_pointer,state_old:System_state):
    LET state_new = prepare_receive_dirty(sender,receiver,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling timeout().
timeout_this_unchanged: LEMMA
  FORALL(state_old:System_state):
    LET state_new = timeout(state_old`this,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling kill().
kill_this_unchanged: LEMMA
  FORALL(receiver:Thread_pointer,state_old:System_state):
    LET state_new = kill(receiver,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling
% ipc_receiver_ready_change().
ipc_receiver_ready_change_this_unchanged: LEMMA
  FORALL(receiver:Thread_pointer,state_old:System_state):
    LET state_new = ipc_receiver_ready_change(state_old`this,receiver,
                                              state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling receiver_ready().
receiver_ready_this_unchanged: LEMMA
  FORALL(receiver:Thread_pointer,state_old:System_state):
    LET state_new = receiver_ready(state_old`this,receiver,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling
% preemption_point_actions().
preemption_point_actions_this_unchanged: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          have_timeout: bool,state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,
                                             have_timeout,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling preemption_point().
preemption_point_this_unchanged: LEMMA
  FORALL (partner: Thread_pointer,have_timeout: bool,state_old: System_state):
    LET state_new = preemption_point(partner,have_timeout,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling abort_send().
abort_send_this_unchanged: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = abort_send(partner,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling
% do_send_wait_finish().
do_send_wait_finish_this_unchanged: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = do_send_wait_finish(partner,state_old) IN
      state_old`this = state_new`this

% Indicate that the <this> field is not changed by calling
% do_send_wait_loop().
```

```
  do_send_wait_loop_this_unchanged: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
      LET state_new = do_send_wait_loop(partner,state_old) IN
        state_old`this = state_new`this

  % Indicate that the <this> field is not changed by calling do_send_wait().
  do_send_wait_this_unchanged: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
      LET state_new = do_send_wait(partner,state_old) IN
        state_old`this = state_new`this

  % Indicate that the <this> field is not changed by calling
  % try_handshake_receiver().
  try_handshake_receiver_this_unchanged: LEMMA
    FORALL (partner: Thread_pointer,state_old: System_state):
      LET state_new = try_handshake_receiver(partner,state_old) IN
        state_old`this = state_new`this

  % Indicate that the <this> field is not changed by calling do_ipc_send_part().
  do_ipc_send_part_this_unchanged: LEMMA
    FORALL (partner: Thread_pointer,have_receive: bool,state_old: System_state):
      LET state_new = do_ipc_send_part(partner,have_receive,state_old) IN
        state_old`this = state_new`this

  % Indicate that the <this> field is not changed by calling
  % do_ipc_receive_part().
  do_ipc_receive_part_this_unchanged: LEMMA
    FORALL (sender: Thread_pointer,have_receive: bool,state_old: System_state):
      LET state_new = do_ipc_receive_part(sender,have_receive,state_old) IN
        state_old`this = state_new`this

  % Indicate that the <this> field is not changed by calling sys_ipc().
  sys_ipc_this_unchanged: LEMMA
    FORALL (have_send: bool, partner: Thread_pointer, have_receive: bool,
            sender: Thread_pointer, state_old: System_state):
      % Engage in IPC with the supplied parameters.
      LET state_new = sys_ipc(have_send,partner,
                              have_receive,sender,state_old) IN
        state_old`this = state_new`this

END fiasco_helpers
```

Appendix E

PVS: fiasco_state.pvs

```
%-----------------------------------------------------------------------------
% Many lemmas depend on specific bits of a thread state to be set or unset after
% calling a function. For all those lemmas dealing with state bits that are not
% directly tied to a property, we have created this separate theory, which other
% theories can easily import and use.
%-----------------------------------------------------------------------------
fiasco_state: THEORY
BEGIN
  %---------------------------------------------------------------------------
  % Import the theories which will be used in our lemmas.
  %---------------------------------------------------------------------------
  IMPORTING fiasco_functions
  IMPORTING fiasco_lock
  IMPORTING fiasco_helpers

  %---------------------------------------------------------------------------
  % Legend:
  %   LEMMA %- : the lemma depends on an assumption.
  %   LEMMA %/ : the lemma depends on an axiom.
  %---------------------------------------------------------------------------

  %---------------------------------------------------------------------------
  % Lemmas dealing with the status of the <thread_ipc_in_progress> bit.
  %---------------------------------------------------------------------------

  % The <thread_ipc_in_progress> bit of the <this> thread will not be changed
  % after sender_enqueue() was called.
  sender_enqueue_ipc_in_progress_unchanged: LEMMA
    FORALL (receiver: Thread_pointer, state_old: System_state):
      LET state_new = sender_enqueue(receiver, state_old) IN
        state_old`threads(state_old`this)`state`thread_ipc_in_progress =
          state_new`threads(state_new`this)`state`thread_ipc_in_progress

  % The <thread_ipc_in_progress> bit of the <this> thread will not be changed
  % after kill() was called.
  kill_ipc_in_progress_unchanged: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
      LET state_new = kill(partner, state_old) IN
        state_old`threads(state_old`this)`state`thread_ipc_in_progress =
          state_new`threads(state_new`this)`state`thread_ipc_in_progress

  % The <thread_ipc_in_progress> bit of the <this> thread will not be changed
  % after receiver_ready() was called and the sender (the <this> thread) was
  % not equal to the receiver.
  %
  % NOTE: depends on the additional assumption that the sender is not equal
  % to the receiver.
```

```
receiver_ready_ipc_in_progress_unchanged: LEMMA %-
 FORALL(sender:Thread_pointer,receiver:Thread_pointer,state_old:System_state):
   LET state_new = receiver_ready(state_old`this,receiver,state_old) IN
     NOT state_old`this = receiver IMPLIES
       state_old`threads(state_old`this)`state`thread_ipc_in_progress =
         state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When no timeouts are allowed, the <thread_ipc_in_progress> bit of the
% <this> thread will not be changed after calling preemption_point_actions().
preemption_point_actions_no_timeout_ipc_in_progress_unchanged: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,false,state_old) IN
      state_old`threads(state_old`this)`state`thread_ipc_in_progress =
        state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When no timeouts are allowed, the <thread_ipc_in_progress> bit of the
% <this> thread will not be changed after calling preemption_point().
preemption_point_no_timeout_ipc_in_progress_unchanged: LEMMA
  FORALL (partner:Thread_pointer,state_old: System_state):
    LET state_new = preemption_point(partner,false,state_old) IN
      state_old`threads(state_old`this)`state`thread_ipc_in_progress =
        state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When no timeout has occured, the <thread_ipc_in_progress> bit of the
% <this> thread will not be changed after calling preemption_point_actions().
preemption_point_actions_no_timeout_ipc_in_progress_unchanged2: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          allow_timeout: bool, state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,
                                             allow_timeout,state_old) IN
      NOT state_old`timeout AND
      NOT state_new`timeout IMPLIES
        state_old`threads(state_old`this)`state`thread_ipc_in_progress =
          state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When no timeout has occured, the <thread_ipc_in_progress> bit of the
% <this> thread will not be changed after calling preemption_point().
preemption_point_no_timeout_ipc_in_progress_unchanged2: LEMMA
  FORALL (partner:Thread_pointer,allow_timeout:bool,state_old: System_state):
    LET state_new = preemption_point(partner,allow_timeout,state_old) IN
      NOT state_old`timeout AND
      NOT state_new`timeout IMPLIES
        state_old`threads(state_old`this)`state`thread_ipc_in_progress =
          state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When the <thread_ipc_in_progress_bit> of the <this> thread is initially
% not set, after calling preemption_point_actions() it will still be unset.
preemption_point_actions_not_thread_ipc_in_progress: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          allow_timeout: bool, state_old: System_state):
    LET state_new = preemption_point_actions(actions, partner,
                                             allow_timeout,state_old) IN
      NOT state_old`threads(state_old`this)`state`thread_ipc_in_progress
      IMPLIES
        NOT state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When the <thread_ipc_in_progress_bit> of the <this> thread is initially
% not set, after calling preemption_point() it will still be unset.
preemption_point_not_thread_ipc_in_progress: LEMMA
  FORALL (partner:Thread_pointer,allow_timeout:bool,state_old: System_state):
    LET state_new = preemption_point(partner,allow_timeout,state_old) IN
      NOT state_old`threads(state_old`this)`state`thread_ipc_in_progress
      IMPLIES
        NOT state_new`threads(state_new`this)`state`thread_ipc_in_progress
```

```
% When the preemption_point_actions() function has set the timeout to true,
% then that implies that the <thread_ipc_in_progress> bit of the <this>
% thread is unset.
preemption_point_actions_timeout_not_thread_ipc_in_progress: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          allow_timeout: bool, state_old: System_state):
    LET state_new = preemption_point_actions(actions, partner,
                                             allow_timeout,state_old) IN
      NOT state_old`timeout AND
       state_new`timeout IMPLIES
        NOT state_new`threads(state_new`this)`state`thread_ipc_in_progress

% When the preemption_point() function has set the timeout to true, then that
% implies that the <thread_ipc_in_progress> bit of the <this> thread is unset.
preemption_point_timeout_not_thread_ipc_in_progress: LEMMA
  FORALL (partner:Thread_pointer,allow_timeout:bool,state_old: System_state):
    LET state_new = preemption_point(partner,allow_timeout,state_old) IN
      NOT state_old`timeout AND
       state_new`timeout IMPLIES
        NOT state_new`threads(state_new`this)`state`thread_ipc_in_progress

%-------------------------------------------------------------------------
% Lemmas which involve the status of the <thread_transfer_in_progress> bit.
%-------------------------------------------------------------------------

% When the <thread_transfer_in_progress> bit of the <this> thread is not set,
% that implies that the partner and the <this> thread are not engaged in IPC.
%
% NOTE: unfortunately this is still an axiom, it should be a lemma.
not_transfer_in_progress_not_in_ipc: AXIOM
  FORALL (partner:Thread_pointer,state_old:System_state):
    NOT state_old`threads(state_old`this)`state`thread_transfer_in_progress
    IMPLIES
      NOT in_ipc(state_old`this,partner,state_old)

%-------------------------------------------------------------------------
% Lemmas which involve the status of the <thread_polling> bit.
%-------------------------------------------------------------------------

% The <thread_polling> bit is not set when sender_ok() returns true.
%
% NOTE: unfortunately we need this axiom because of a bug in the code.
% Should the bug be fixed, than this axiom can be removed.
sender_ok_not_polling: AXIOM
  FORALL (partner: Thread_pointer, state_old: System_state):
     sender_ok(state_old`this, partner, state_old) IMPLIES
      NOT state_old`threads(state_old`this)`state`thread_polling

% The <thread_polling> bit of the <this> thread is not modified by the kill()
% function.
kill_polling_unchanged: LEMMA
  FORALL (partner:Thread_pointer,state_old:System_state):
    LET state_new = kill(partner,state_old) IN
      state_old`threads(state_old`this)`state`thread_polling =
        state_new`threads(state_new`this)`state`thread_polling

% The <thread_polling> bit of the <this> thread is not modified by the
% timeout() function.
timeout_polling_unchanged: LEMMA
  FORALL (state_old:System_state):
    LET state_new = timeout(state_old`this,state_old) IN
      state_old`threads(state_old`this)`state`thread_polling =
        state_new`threads(state_new`this)`state`thread_polling

% The <thread_polling> bit of the <this> thread is not modified by the
% receiver_ready() function.
```

```
%
% NOTE: depends on the additional assumption that the sender is not equal
% to the receiver.
receiver_ready_polling_unchanged: LEMMA %-
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = receiver_ready(state_old`this, partner, state_old) IN
      NOT state_old`this = partner IMPLIES
       state_old`threads(state_old`this)`state`thread_polling =
        state_new`threads(state_new`this)`state`thread_polling

% The <thread_polling> bit of the <this> thread is not modified by the
% preemption_point_actions() function.
preemption_point_actions_polling_unchanged: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          allow_timeout: bool, state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,
                                             allow_timeout,state_old) IN
      state_old`threads(state_old`this)`state`thread_polling =
        state_new`threads(state_new`this)`state`thread_polling

% The <thread_polling> bit of the <this> thread is not modified by the
% preemption_point() function.
preemption_point_polling_unchanged: LEMMA
  FORALL (partner:Thread_pointer,allow_timeout:bool,state_old:System_state):
    LET state_new = preemption_point(partner,allow_timeout,state_old) IN
      state_old`threads(state_old`this)`state`thread_polling =
        state_new`threads(state_new`this)`state`thread_polling

% After calling abort_send(), the <this> thread's <thread_polling> bit will
% always be unset.
abort_send_not_polling: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = abort_send(partner,state_old) IN
      NOT state_new`threads(state_new`this)`state`thread_polling

% After calling do_send_wait_finish(), the <this> thread's <thread_polling>
% bit will always be unset.
do_send_wait_finish_not_polling: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = do_send_wait_finish(partner,state_old) IN
      NOT state_new`threads(state_new`this)`state`thread_polling

% After calling do_send_wait_loop(), the <this> thread's <thread_polling> bit
% will always be unset.
do_send_wait_loop_not_polling: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = do_send_wait_loop(partner,state_old) IN
      NOT state_new`threads(state_new`this)`state`thread_polling

% After calling do_send_wait(), the <this> thread's <thread_polling> bit will
% always be unset.
%
% NOTE: depends on the [sender_ok_not_polling] axiom.
do_send_wait_not_polling: LEMMA %/
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = do_send_wait(partner,state_old) IN
      NOT state_new`threads(state_new`this)`state`thread_polling

% If no error has occured when calling try_handshake_receiver(), the
% <thread_polling> of the <this> thread will always be unset.
%
% NOTE: depends on the additional assumption that initially the
% <thread_polling> bit of the <this> thread is not set.
try_handshake_receiver_no_error_not_polling: LEMMA %-
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = try_handshake_receiver(partner,state_old) IN
```

```
      NOT state_old`error AND
      NOT state_new`error AND
      NOT state_old`threads(state_old`this)`state`thread_polling IMPLIES
        NOT state_new`threads(state_new`this)`state`thread_polling

  % If we assume that initially the <this> thread's <thread_polling> bit is not
  % set, it will still be unset after calling try_handshake_receiver().
  %
  % NOTE: depends on the additional assumption that initially the
  % <thread_polling> bit of the <this> thread is not set.
  try_handshake_receiver_not_polling: LEMMA %-
    FORALL (partner: Thread_pointer,state_old: System_state):
      LET state_new = try_handshake_receiver(partner,state_old) IN
        NOT state_old`error AND
        NOT state_old`threads(state_old`this)`state`thread_polling IMPLIES
          NOT state_new`threads(state_new`this)`state`thread_polling

  % If we assume that initially the <this> thread's <thread_polling> bit is not
  % set, it will still be unset after calling do_ipc_send_part().
  %
  % NOTE: depends on the additional assumption that initially the
  % <thread_polling> bit of the <this> thread is not set.
  do_ipc_send_part_not_polling: LEMMA %-
    FORALL (partner: Thread_pointer,have_receive: bool,state_old: System_state):
      LET state_new = do_ipc_send_part(partner,have_receive,state_old) IN
        NOT state_old`error AND
        NOT state_old`threads(state_old`this)`state`thread_polling IMPLIES
          NOT state_new`threads(state_new`this)`state`thread_polling

  % If we assume that initially, the <this> thread's <thread_polling> bit is
  % not set, it will still be unset after calling sys_ipc().
  %
  % NOTE: depends on the additional assumption that initially the
  % <thread_polling> bit of the <this> thread is not set.
  %
  % NOTE: basically we have proved here that the <thread_polling> bit being
  % unset is an invariant.
  sys_ipc_not_polling: LEMMA %-
    FORALL (have_send: bool, partner: Thread_pointer, have_receive: bool,
            sender: Thread_pointer, state_old: System_state):
      % Engage in IPC with the supplied parameters.
      LET state_new = sys_ipc(have_send,partner,
                              have_receive,sender,state_old) IN
        NOT state_old`threads(state_old`this)`state`thread_polling IMPLIES
          NOT state_new`threads(state_new`this)`state`thread_polling

END fiasco_state
```

Appendix F

PVS: fiasco_lock.pvs

```
%-------------------------------------------------------------------------------
% When the sender and receiver engage in IPC it is required that the receiver
% is not locked by another thread, because otherwise the sender does not have
% complete access to the receiver (which it needs to send the message).
% Therefore, when the receiver is a valid receiver (because otherwise IPC is
% prematurely aborted), the sender attempts a handshake with the receiver.
% Part of that handshake is a check if the receiver is locked by another thread.
% If so, the sender acquires the thread lock. There are thus two possibilities
% for the receiver's thread lock: it is held by the sender (the <this> thread)or
% it is not held. After IPC has finished, the thread lock on the receiver should
% be released, because otherwise other threads cannot access the receiver thread.
%-------------------------------------------------------------------------------
fiasco_lock: THEORY
BEGIN
  %-----------------------------------------------------------------------------
  % Import the theories which will be used in our lemmas.
  %-----------------------------------------------------------------------------
  IMPORTING fiasco_functions
  IMPORTING fiasco_helpers

  %-----------------------------------------------------------------------------
  % Legend:
  %   LEMMA %- : the lemma depends on an assumption.
  %   LEMMA %/ : the lemma depends on an axiom.
  %-----------------------------------------------------------------------------

  %-----------------------------------------------------------------------------
  % Lemmas which involve the status of the <thread lock> field of the receiver.
  %-----------------------------------------------------------------------------

  % The thread lock on the partner is always free after calling abort_send().
  abort_send_lock_free: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
      LET state_new = abort_send(partner,state_old) IN
        state_new`threads(partner)`thread_lock = Zero_thread

  % The thread lock on the partner is not changed by sender_enqueue().
  sender_enqueue_lock_unchanged: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
      LET state_new = sender_enqueue(partner,state_old) IN
        state_old`threads(partner)`thread_lock =
          state_new`threads(partner)`thread_lock

  % The thread lock on the partner is not changed by timeout().
  timeout_lock_unchanged: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
      LET state_new = timeout(state_old`this,state_old) IN
```

```
      state_old`threads(partner)`thread_lock =
        state_new`threads(partner)`thread_lock

% The thread lock on the partner is not changed by kill().
kill_lock_unchanged: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = kill(partner,state_old) IN
      state_old`threads(partner)`thread_lock =
        state_new`threads(partner)`thread_lock

% The thread lock on the partner is not changed by ipc_receiver_ready_change().
ipc_receiver_ready_change_lock_unchanged: LEMMA
 FORALL (partner:Thread_pointer,state_old:System_state):
  LET state_new = ipc_receiver_ready_change(state_old`this,partner,state_old)IN
     state_old`threads(partner)`thread_lock =
      state_new`threads(partner)`thread_lock

% The thread lock on the partner is not changed by receiver_ready().
receiver_ready_lock_unchanged: LEMMA
  FORALL(partner:Thread_pointer,state_old:System_state):
    LET state_new = receiver_ready(state_old`this,partner,state_old) IN
      state_old`threads(partner)`thread_lock =
        state_new`threads(partner)`thread_lock

% The thread lock on the partner is not changed by preemption_point_actions().
preemption_point_actions_lock_unchanged: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          have_timeout: bool, state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,have_timeout,
                                             state_old) IN
      state_old`threads(partner)`thread_lock =
        state_new`threads(partner)`thread_lock

% The thread lock on the partner is not changed by preemption_point().
preemption_point_lock_unchanged: LEMMA
  FORALL (partner: Thread_pointer,have_timeout: bool,state_old: System_state):
    LET state_new = preemption_point(partner,have_timeout,state_old) IN
      state_old`threads(partner)`thread_lock =
        state_new`threads(partner)`thread_lock

% The thread lock on the partner is always free or held by the <this> thread
% after calling do_send_wait_finish() if initially the <this> thread is the
% owner of the thread lock on the receiver and if no error occurred while
% executing the function function.
do_send_wait_finish_no_error_lock_free_or_held: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = do_send_wait_finish(partner,state_old) IN
      NOT state_old`error AND
      NOT state_new`error AND
      state_old`threads(partner)`thread_lock = state_old`this IMPLIES
        state_new`threads(partner)`thread_lock = Zero_thread OR
        state_new`threads(partner)`thread_lock = state_new`this

% The thread lock on the partner is always free or held by the <this> thread
% after calling do_send_wait_loop(), if initially the <this> thread is the
% owner of the thread lock on the receiver and if no error occurred while
% executing the function function.
do_send_wait_loop_no_error_lock_free_or_held: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = do_send_wait_loop(partner,state_old) IN
      NOT state_old`error AND
      NOT state_new`error AND
      state_old`threads(partner)`thread_lock = state_old`this IMPLIES
        state_new`threads(partner)`thread_lock = Zero_thread OR
        state_new`threads(partner)`thread_lock = state_new`this
```

```
% The thread lock on the partner is always free or held by the <this> thread
% if no error occured in the executing of do_send_wait().
do_send_wait_no_error_lock_free_or_held: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = do_send_wait(partner,state_old) IN
      NOT state_old`error AND
      NOT state_new`error IMPLIES
        state_new`threads(partner)`thread_lock = Zero_thread OR
        state_new`threads(partner)`thread_lock = state_new`this

% The thread lock on the partner is always free or held by the <this> thread
% if no error occured in the executing of try_handshake_receiver().
try_handshake_receiver_no_error_lock_free_or_held: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = try_handshake_receiver(partner,state_old) IN
      NOT state_old`error AND
      NOT state_new`error IMPLIES
        state_new`threads(partner)`thread_lock = Zero_thread OR
        state_new`threads(partner)`thread_lock = state_new`this

% The thread lock on the partner is always free if an error occured in the
% execution of do_send_wait_finish().
do_send_wait_finish_error_lock_free: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
   LET state_new = do_send_wait_finish(partner,state_old) IN
    state_new`error IMPLIES state_new`threads(partner)`thread_lock = Zero_thread

% The thread lock on the partner is always free if an error occured in the
% execution of do_send_wait_loop().
do_send_wait_loop_error_lock_free: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
   LET state_new = do_send_wait_loop(partner,state_old) IN
    state_new`error IMPLIES state_new`threads(partner)`thread_lock = Zero_thread

% The thread lock on the partner is always free if an error occured in the
% execution of do_send_wait().
do_send_wait_error_lock_free: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
   LET state_new = do_send_wait(partner,state_old) IN
    state_new`error IMPLIES state_new`threads(partner)`thread_lock = Zero_thread

% The thread lock on the partner is always free if an error occured when
% executing try_handshake_receiver() and a handshake has been attempted.
try_handshake_receiver_handshake_error_lock_free: LEMMA
  FORALL (partner: Thread_pointer,state_old: System_state):
    LET state_new = try_handshake_receiver(partner,state_old) IN
      NOT state_old`handshake_attempted AND
      NOT state_old`error AND
      state_new`handshake_attempted AND
      state_new`error IMPLIES
        state_new`threads(partner)`thread_lock = Zero_thread

% The thread lock on the partner is always free after calling
% do_ipc_send_part() if a handshake has been attempted.
do_ipc_send_part_handshake_attempted_lock_free: LEMMA
  FORALL (partner: Thread_pointer,have_receive: bool,state_old: System_state):
    LET state_new = do_ipc_send_part(partner,have_receive,state_old) IN
      NOT state_old`handshake_attempted AND
      NOT state_old`error AND
      state_new`handshake_attempted IMPLIES
        state_new`threads(partner)`thread_lock = Zero_thread

% If an IPC handshake has been attempted on a valid IPC partner, this should
% imply that there is no thread lock on the partner after IPC has finished.
handshake_attempted_lock_free: LEMMA
  FORALL (have_send: bool, partner: Thread_pointer, have_receive: bool,
```

```
            sender: Thread_pointer, state_old: System_state):
      % Engage in IPC with the supplied parameters.
      LET state_new = sys_ipc(have_send,partner,
                              have_receive,sender,state_old) IN
        state_new`handshake_attempted IMPLIES
          state_new`threads(partner)`thread_lock = Zero_thread

END fiasco_lock
```

Appendix G

PVS: fiasco_wakeup.pvs

```
%-----------------------------------------------------------------------------
% In Fiasco IPC it is possible to have a combined send- and receive call, which
% means that after a sender has sent its message to the receiver. It then waits
% for that receiver to send a message back to him. After the sender has sent
% its message, it thus enters a receiving state. Simultaneously the receiver
% enters a sending state. Now the sender has to wait for the receiver to
% send its message. It is therefore important that the receiver is ready to be
% scheduled, otherwise the sender might be waiting forever for the receiver to
% send a message back. The receiver being ready to be scheduled is controlled
% by a single bit, namely the <thread_ready> bit that thus has to be set when
% the receiver switches to the sending status.
%-----------------------------------------------------------------------------
fiasco_wakeup: THEORY
BEGIN
  %---------------------------------------------------------------------------
  % Import the theories which will be used in our lemmas.
  %---------------------------------------------------------------------------
  IMPORTING fiasco_functions

  %---------------------------------------------------------------------------
  % Legend:
  %   LEMMA %- : the lemma depends on an assumption.
  %   LEMMA %/ : the lemma depends on an axiom.
  %---------------------------------------------------------------------------

  %---------------------------------------------------------------------------
  % Lemmas involving the <thread_ready> bit of the partner thread.
  %---------------------------------------------------------------------------

  % When the sender- and receiver are still engaged in IPC after the send part
  % has finished, the receiver should be in a state ready to be scheduled which
  % means that the <thread_ready> bit of the receiver should be set.
  do_ipc_send_part_in_ipc_receiver_awoken: LEMMA
    FORALL (partner: Thread_pointer, have_receive: bool, state_old: System_state):
      LET state_new = do_ipc_send_part(partner, have_receive, state_old) IN
        NOT state_new`error AND in_ipc(state_new`this, partner, state_new) IMPLIES
          state_new`threads(partner)`state`thread_ready

END fiasco_wakeup
```

Appendix H

PVS: fiasco_assert.pvs

```
%-----------------------------------------------------------------------------
% This theory focuses on the verification of the assert() calls that are made in
% the Fiasco IPC source code. It is important that our model can verify that
% these assertions hold, as a failure of a single assertion will cause Fiasco to
% abort execution completely.
%-----------------------------------------------------------------------------
fiasco_assert: THEORY
BEGIN
  %---------------------------------------------------------------------------
  % Import the theories which will be used in our lemmas.
  %---------------------------------------------------------------------------
  IMPORTING fiasco_functions
  IMPORTING fiasco_lock
  IMPORTING fiasco_state
  IMPORTING fiasco_helpers

  %---------------------------------------------------------------------------
  % Legend:
  %   LEMMA %- : the lemma depends on an assumption.
  %   LEMMA %/ : the lemma depends on an axiom.
  %---------------------------------------------------------------------------

  %---------------------------------------------------------------------------
  % Declaration of lemmas related to the system state's <assertions_held> field.
  %---------------------------------------------------------------------------

  % The assertions hold after sender_dequeue() is called.
  sender_dequeue_assertions_held: LEMMA
    FORALL (receiver:Thread_pointer,state_old: System_state):
      LET state_new = sender_dequeue(receiver,state_old) IN
        state_old`assertions_held IMPLIES state_new`assertions_held

  % The assertions hold after sender_enqueue() is called.
  sender_enqueue_assertions_held: LEMMA
    FORALL (receiver:Thread_pointer,state_old: System_state):
      LET state_new = sender_enqueue(receiver,state_old) IN
        state_old`assertions_held IMPLIES state_new`assertions_held

  % The assertions hold after timeout() is called.
  timeout_assertions_held: LEMMA
    FORALL (state_old: System_state):
      LET state_new = timeout(state_old`this,state_old) IN
        state_old`assertions_held IMPLIES state_new`assertions_held

  % The assertions hold after kill() is called.
  kill_assertions_held: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
```

```
    LET state_new = kill(partner,state_old) IN
      state_old`assertions_held IMPLIES state_new`assertions_held

% The assertions hold after receiver_ready() is called.
receiver_ready_assertions_held: LEMMA
 FORALL(sender:Thread_pointer,receiver:Thread_pointer,state_old:System_state):
   LET state_new = receiver_ready(state_old`this,receiver,state_old) IN
     state_old`assertions_held IMPLIES state_new`assertions_held

% The assertions hold after preemption_point_actions() is called.
preemption_point_actions_assertions_held: LEMMA
  FORALL (actions: Preemption_actions, partner: Thread_pointer,
          allow_timeout: bool, state_old: System_state):
    LET state_new = preemption_point_actions(actions,partner,
                                             allow_timeout,state_old) IN
      state_old`assertions_held IMPLIES state_new`assertions_held

% The assertions hold after preemption_point() is called.
preemption_point_assertions_held: LEMMA
  FORALL (partner:Thread_pointer,allow_timeout:bool,state_old: System_state):
    LET state_new = preemption_point(partner,allow_timeout,state_old) IN
      state_old`assertions_held IMPLIES state_new`assertions_held

% The assertions hold after abort_send() is called.
abort_send_assertions_held: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = abort_send(partner,state_old) IN
      state_old`assertions_held IMPLIES state_new`assertions_held

% The assertions hold after do_send_wait_finish() is called and initially
% either a timeout occurred or the <thread_ipc_in_progress> bit of the
% <this> thread is set.
do_send_wait_finish_assertions_held: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = do_send_wait_finish(partner,state_old) IN
      state_old`assertions_held AND
     (state_old`timeout OR
      state_old`threads(state_old`this)`state`thread_ipc_in_progress) IMPLIES
        state_new`assertions_held

% The assertions hold after do_send_wait_loop() is called and initially
% no timeout occurred and the <thread_ipc_in_progress> bit of the
% <this> thread is set.
%
% NOTE: depends on the [not_transfer_in_progress_not_in_ipc] axiom.
do_send_wait_loop_assertions_held: LEMMA %/
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = do_send_wait_loop(partner,state_old) IN
      state_old`assertions_held AND
      NOT state_old`timeout AND
      state_old`threads(state_old`this)`state`thread_ipc_in_progress IMPLIES
        state_new`assertions_held

% The assertions hold after do_send_wait() is called when initially
% no timeout occured and the partner thread is not locked or locked
% by the <this> thread.
do_send_wait_assertions_held: LEMMA
  FORALL (partner: Thread_pointer, state_old: System_state):
    LET state_new = do_send_wait(partner,state_old) IN
      NOT state_old`timeout AND
      state_old`assertions_held  AND
      (state_old`threads(partner)`thread_lock = Zero_thread OR
       state_old`threads(partner)`thread_lock = state_old`this)
      IMPLIES
        state_new`assertions_held
```

```
  % The various assertions in the code hold after calling
  % try_handshake_receiver() when initially no error or timeout occured.
  try_handshake_receiver_assertions_held: LEMMA
    FORALL (partner: Thread_pointer, state_old: System_state):
      % Engage in IPC with the supplied parameters.
      LET state_new = try_handshake_receiver(partner,state_old) IN
        NOT state_old`timeout AND
        NOT state_old`error AND
        state_old`assertions_held
        IMPLIES
          state_new`assertions_held

  % The various assertions in the code should hold after handling the send part
  % when initially no error or timeout occurred.
  %
  % NOTE: depends on the additional assumption that initially the
  % <thread_polling> bit of the <this> thread is not set.
  do_ipc_send_part_assertions_held: LEMMA %-
    FORALL (partner: Thread_pointer,have_receive: bool,state_old: System_state):
      LET state_new = do_ipc_send_part(partner,have_receive,state_old) IN
        NOT state_old`error AND
        NOT state_old`timeout AND
        NOT state_old`threads(state_old`this)`state`thread_polling AND
        state_old`assertions_held
        IMPLIES
          state_new`assertions_held

  % The various assertions in the code should hold after engaging in IPC.
  %
  % NOTE: depends on the additional assumption that initially the
  % <thread_polling> bit of the <this> thread is not set.
  assertions_held: LEMMA %-
    FORALL (have_send: bool, partner: Thread_pointer, have_receive: bool,
            sender: Thread_pointer,state_old: System_state):
      % Engage in IPC with the supplied parameters.
      LET state_new = sys_ipc(have_send,partner,
                              have_receive,sender,state_old) IN
        NOT state_old`threads(state_old`this)`state`thread_polling IMPLIES
          state_new`assertions_held

END fiasco_assert
```

Appendix I

C++: thread-ipc.cpp

```
//-----------------------------------------------------------------------------
// LEGEND:
//   //% : comments added by us.
//   //# : a line that has not been modelled.
//   //! : a line that has been added by us.
//-----------------------------------------------------------------------------

/** Receiver-ready callback.
    Receivers make sure to call this function on waiting senders when
    they get ready to receive a message from that sender. Senders need
    to overwrite this interface.

    Class Thread's implementation wakes up the sender if it is still in
    sender-wait state.
 */
PUBLIC virtual
bool
Thread::ipc_receiver_ready(Receiver *recv)
{
  assert(receiver());                                                    //#
  assert(receiver() == recv);                                            //#
  assert(receiver() == current());                                       //#

  if(!(state() & Thread_ipc_in_progress))
    return false;

  if(!recv->sender_ok(this))
    return false;

  recv->ipc_init(this);

  state_add_dirty (Thread_ready | Thread_transfer_in_progress);

  ready_enqueue();                                                       //#

  // put receiver into sleep
  receiver()->state_del_dirty(Thread_ready);

  return true;
}

/** L4 IPC system call.
    This is the 'normal'' version of the IPC system call.  It usually only
    gets called if ipc_short_cut() has failed.
    @param regs system-call arguments.
    @return value to be returned in %eax register.
 */
```

```
IMPLEMENT inline NOEXPORT ALWAYS_INLINE
void
Thread::sys_ipc()
{
  Sys_ipc_frame *regs = sys_frame_cast<Sys_ipc_frame>(this->regs());       //#

  Ipc_err ret(0);                                                          //#
  L4_timeout t = regs->timeout();                                          //#

  // find the ipc partner thread belonging to the destination id
  L4_uid id              = regs->snd_dst();                                //#
  bool have_receive_part = regs->rcv_desc().has_receive();                 //#
  bool have_send_part    = regs->snd_desc().has_snd();                     //#
  bool have_sender       = false;                                          //#
  bool lookup_done       = false;                                          //#
  Thread *partner        = 0;                                              //#
  Sender *sender         = 0;                                              //#
  Irq_alloc *interrupt   = 0;                                              //#

  // Add Thread_delayed_* flags if this a "next-period" IPC.
  // The flags must be cleared again on all exit paths from this function.
  if (EXPECT_FALSE (id.next_period()))                                     //#
    {
      state_add (Thread_delayed_deadline);                                 //#

      if (id.is_nil() && t.rcv_exp() && !t.rcv_man() &&                    //#
          !regs->has_abs_rcv_timeout())         // next period, 0 timeout  //#
        goto success;                                                      //#

      if (mode() & Nonstrict)                                              //#
        state_add (Thread_delayed_ipc);                                    //#
    }

  // do we do a closed receive operation?
  if (EXPECT_TRUE(have_receive_part))                                      //#
    {
      if (regs->rcv_desc().open_wait())                                    //#
        have_sender = true;                                                //#
      else if (EXPECT_FALSE(id.is_nil()))                                  //#
        {
          // only prepare IPC for timeout != 0                             //#
          if (!t.rcv_exp() || t.rcv_man() || regs->has_abs_rcv_timeout())  //#
            {
              sender = nil_thread;                                         //#
              have_sender = true;                                          //#
            }
        }
      else if (EXPECT_FALSE(id.is_irq()))                                  //#
        {
          interrupt = Irq_alloc::lookup (regs->irq());                     //#

          if (EXPECT_FALSE (!interrupt))                                   //#
            {
              commit_ipc_failure (regs, Ipc_err::Enot_existent);           //#
              return;                                                      //#
            }

          if (interrupt->owner() != this)                                  //#
            {
              // a receive with a timeout != 0 from a non-associated
              // interrupt is illegal
              if (!t.rcv_exp() || t.rcv_man()                              //#
                  || regs->has_abs_rcv_timeout())                          //#
                {
                  commit_ipc_failure (regs, Ipc_err::Enot_existent);       //#
                  return;                                                  //#
```

```
            }
        }

      if (interrupt->owner() == this)                                    //#
        {
          // we always try to receive from the
          // assoc'd irq first, not from the spec. one
          sender = nonull_static_cast<Irq*>(interrupt);                  //#
          have_sender = true;                                            //#
        }
    }
  else                                        // regular thread id       //#
    {
      partner = lookup (id, space());                                    //#
      lookup_done = true;                                                //#

      if (EXPECT_FALSE (!partner || partner->state() == Thread_invalid))//#
        {
          commit_ipc_failure (regs, Ipc_err::Enot_existent);             //#
          return;                                                        //#
        }

      if (EXPECT_FALSE (id.is_preemption()))                             //#
        sender = partner->preemption();                                  //#
      else                                                               //#
        sender = partner;                                                //#

      have_sender = true;                                                //#
    }
}

if (EXPECT_TRUE(have_send_part))                                         //#
{
  // irq quirks
  if (EXPECT_FALSE(id.is_irq()))                                         //#
    {
      Irq_alloc *irq = Irq_alloc::lookup(regs->irq());                   //#
      if (EXPECT_FALSE(!irq || irq->owner() != this))                    //#
        {
          // failed -- could not associate new irq
          commit_ipc_failure (regs, Ipc_err::Enot_existent);             //#
          return;                                                        //#
        }

      if (regs->msg_word(0) == 0) // ack IRQ                             //#
        irq->acknowledge();                                              //#
      else if (regs->msg_word(0) == 1) // disassociate IRQ               //#
        disassociate_irq(irq);                                           //#
      else                                                               //#
        {
          // failed -- could not associate new irq
          commit_ipc_failure (regs, Ipc_err::Enot_existent);             //#
          return;                                                        //#
        }
      have_send_part = false;                                            //#
    }
  else if (! lookup_done)                                                //#
    partner = lookup (id, space());                                      //#
}

if (EXPECT_FALSE(have_receive_part && !have_sender))                     //#
{
  if (!interrupt) // is disassoc from IRQ                                //#
    {
      assert(t.rcv_exp() && !t.rcv_man()); // timeout 0                  //#
      //
```

```
          // disassociate from all IRQs
          //
          if (_irq)                                                    //#
            {
              Irq_alloc::free_all(this);                               //#
              _irq = 0;                                                //#
            }
          commit_ipc_failure (regs, Ipc_err::Retimeout);               //#
          return;                                                      //#
        }
      else                                                             //#
        {
          //
          // associate with an interrupt
          //
          if (associate_irq(interrupt))                                //#
            {
              // success
              commit_ipc_failure (regs, Ipc_err::Retimeout);           //#
              return;                                                  //#
            }
        }
      have_receive_part = false;                                       //#
    }

  if (EXPECT_TRUE(have_send_part || have_receive_part))
    ret = do_ipc(have_send_part, partner,
                 have_receive_part, sender,
                 t, regs);

success:                                                               //#
  if (EXPECT_TRUE(!ret.has_error()))                                   //#
    commit_ipc_success (regs, ret);                                    //#
  else                                                                 //#
    commit_ipc_failure (regs, ret);                                    //#

  // New period doesn't begin as long as Thread_delayed_deadline is still set
  while (state_change_safely (~(Thread_delayed_deadline | Thread_ready),   //#
                              Thread_delayed_deadline))                //#
    schedule();                                                        //#
}

PRIVATE
Ipc_err Thread::do_send_wait (Thread *partner, L4_timeout t, Sys_ipc_frame *regs)
{

  // test if we have locked the partner
  // in the best case the partner is unlocked
  assert(!partner->thread_lock()->test()
         || partner->thread_lock()->lock_owner() == this);

  state_add_dirty(Thread_polling|Thread_send_in_progress|Thread_ipc_in_progress);
  // register partner so that we can be dequeued by kill()
  set_receiver (partner);                                              //#

  // lock here for the preemption points
  partner->thread_lock()->lock_dirty();
  Proc::preemption_point();

  if(state() & Thread_cancel)
    {
      partner->thread_lock()->clear_dirty();

      state_del (Thread_ipc_sending_mask
                 |Thread_transfer_in_progress
                 |Thread_ipc_in_progress);
```

```
    return Ipc_err(Ipc_err::Secanceled);
  }

if(partner->sender_ok(this))
  return 0;

sender_enqueue (partner->sender_list(), sched_context()->prio());

assert(partner->partner() != this);                                    //!
assert(!(state() & Thread_transfer_in_progress));                      //!

Proc::preemption_point();

IPC_timeout timeout;                                                   //#

if (EXPECT_FALSE(t.snd_exp()))
  {
    Unsigned64 tval = snd_timeout (t, regs);                           //#
    // Zero timeout or timeout expired already -- give up
    if (tval == 0)                                                     //#
        return abort_send(Ipc_err::Setimeout, partner);

    // enqueue timeout
    Proc::preemption_point();                                          //#

    set_timeout (&timeout);                                            //#
    timeout.set (tval);                                                //#
  }

do
  {

    partner->thread_lock()->clear_dirty();
    // possible PREEMPTION POINT

    Proc::preemption_point();

    //% We have not modelled the call to the <schedule()> function as
    //% its purpose is to give the receiver time to become ready to
    //% receive a message from this thread. In our model this can
    //% happen when the <preemption_point()> function is called.
    //% As we do not actually model the time a receiver needs to
    //% have to become ready this waiting is uncessary.
    if((state() & (Thread_ipc_in_progress | Thread_polling             //#
                   | Thread_cancel | Thread_transfer_in_progress))     //#
       == (Thread_ipc_in_progress|Thread_polling))                     //#
      {
        state_del_dirty(Thread_ready);                                 //#
        schedule();                                                    //#
      }

    //% This preemption point is superfluous in our model as we
    //% abstracted away the code directly above and thus would end
    //% up with two consecutive preemption points.
    Proc::preemption_point();                                          //#

    // if we are here 5 cases are  possible
    // - timeout has hit
    // - ipc was canceled
    // - the receiver had awoken us
    // - the receiver has been killed
    // - or someone has simply awake us, then we go to sleep again

    partner->thread_lock()->lock_dirty();
```

```
    if (EXPECT_FALSE(state() & Thread_cancel))
      break;

    // ipc handshake bit is set
    if(state() & Thread_transfer_in_progress)
      break;

    assert(!partner->in_ipc(this));

    //% This assertion is redundant due to the break specified above.
    assert(!(state() & Thread_transfer_in_progress));

    // detect if we have timed out
    if (timeout.has_hit())
      {
        // ipc timeout always clear this flag
        assert(!(state() & Thread_ipc_in_progress));
        return abort_send(Ipc_err::Setimeout, partner);
      }

    if(partner->state() == Thread_dead)
      return abort_send(Ipc_err::Enot_existent, partner);

    // huh? someone has simply awake us, goto sleep again

    // Make sure we're really still in IPC
    assert(state() &  Thread_ipc_in_progress);

    // adding again, so that ipc_receiver_ready find the correct state
    state_add_dirty(Thread_polling);

  } while(true);                                                    //#

// because the handshake has already taken place
// an triggered timeout can be ignored
if (timeout.has_hit())
  state_add_dirty(Thread_ipc_in_progress);

// reset is only an simple dequeing operation from an double
// linked list, so we dont need an extra preemption point for this
timeout.reset();                                                    //#
set_timeout(0);                                                     //#

Proc::preemption_point();
sender_dequeue (partner->sender_list());
Proc::preemption_point();

state_del_dirty(Thread_polling);

if (EXPECT_FALSE(state() & Thread_cancel))
  {
    // this test catch partner-cancel and kill
    if(!partner->in_ipc(this))
      return abort_send(Ipc_err::Secanceled, partner);

    // the partner still waits for us, cancel them too
    partner->state_change(~Thread_ipc_in_progress,
                          Thread_cancel | Thread_ready);
    partner->ready_enqueue();                                       //#
    Proc::preemption_point();                                       //#
    return abort_send(Ipc_err::Seaborted, partner);
  }

// partner canceled?, handles kill too
if(EXPECT_FALSE(!partner->in_ipc(this)))
  return abort_send(Ipc_err::Seaborted, partner);
```

```
  if(partner->state() == Thread_dead)
    return abort_send(Ipc_err::Enot_existent, partner);

  assert(state() & Thread_ipc_in_progress);
  assert (!in_sender_list());

  return 0;
}

PRIVATE
Ipc_err Thread::abort_send(Ipc_err err, Receiver *partner)
{
  state_del_dirty (Thread_send_in_progress | Thread_polling |
                   Thread_ipc_in_progress | Thread_transfer_in_progress);

  assert(partner);                                                    //#

  Proc::preemption_point();
  if(in_sender_list())
    sender_dequeue (partner->sender_list());

  Proc::preemption_point();
  partner->thread_lock()->clear_dirty();

  //% Another preemption point is not useful in our model as after this
  //% function has been called the termination will be terminated and
  //% thus no other statements will be executed which might have been
  //% influenced by the preemption point.
  Proc::preemption_point();
  if (_timeout && _timeout->is_set())                                 //#
    _timeout->reset();                                                //#

  set_timeout(0);                                                     //#

  return err;
}

PRIVATE inline NEEDS["logdefs.h"]
Ipc_err Thread::try_handshake_receiver(Thread *partner, L4_timeout t,
                                       Sys_ipc_frame *regs)
{

  // By touching the partner tcb we can raise an pagefault.
  // The Pf handler might be enable the interrupts, if no mapping in
  // the master kernel directory exists.
  // Because the partner can created in between,
  // and partner->state() == Thread_invalid is insufficient
  // we need a cancel test.
  //

  if(EXPECT_FALSE (partner == 0
                   // must not send to L4_NIL_ID
                   || partner->state() == Thread_invalid
                   || partner == nil_thread))
    return Ipc_err (Ipc_err::Enot_existent);

  assert(cpu_lock.test());                                            //#

  if(EXPECT_FALSE(partner->thread_lock()->test())) // lock is not free
    {
      Proc::preemption_point();
      partner->thread_lock()->lock_dirty();
    }

  if(EXPECT_FALSE(state() & Thread_cancel))
```

```
    {
      // clear_dirty() handle the not locked case too
      partner->thread_lock()->clear_dirty();
      return Ipc_err(Ipc_err::Secanceled);
    }

  Ipc_err err(0);

  if (EXPECT_FALSE (!partner->sender_ok
                    (nonull_static_cast<Sender*>(current_thread()))))

    {
      err = do_send_wait(partner, t, regs);

      // if an error occured, we should not hold the lock anymore
      assert(!err.has_error() || partner->thread_lock()->lock_owner() != this);
    }

  return err;
}

PRIVATE inline
void
Thread::wake_receiver (Thread *receiver)
{
  // If neither IPC partner is delayed, just update the receiver's state
  if (EXPECT_TRUE (!((state() | receiver->state()) & Thread_delayed_ipc))) //#
    {
      receiver->state_change_dirty(~(Thread_ipc_receiving_mask
                                     | Thread_ipc_in_progress),
                                   Thread_ready);
      return;
    }

  //% As we do not model periodic threads, we assume that the
  //% if-statement above will always be true. The section below is thus
  //% completely ignored.

  // Critical section if either IPC partner is delayed until its next period
  assert(cpu_lock.test());                                                  //#

  // Sender has no receive phase and deadline timeout already hit
  if ( (state() & (Thread_receiving |                                       //#
                   Thread_delayed_deadline | Thread_delayed_ipc)) ==        //#
      Thread_delayed_ipc)                                                   //#
    {
      state_change_dirty (~Thread_delayed_ipc, 0);                          //#
      switch_sched (sched_context()->next());                               //#
      _deadline_timeout.set (Timer::system_clock() + period());             //#
    }

  // Receiver's deadline timeout already hit
  if ( (receiver->state() & (Thread_delayed_deadline |                      //#
                             Thread_delayed_ipc) ==                         //#
                             Thread_delayed_ipc))                           //#
    {
      receiver->state_change_dirty (~Thread_delayed_ipc, 0);                //#
      receiver->switch_sched (receiver->sched_context()->next());           //#
      receiver->_deadline_timeout.set (Timer::system_clock() +              //#
                                       receiver->period());                 //#
    }

  receiver->state_change_dirty (~(Thread_ipc_mask|Thread_delayed_ipc),      //#
                                Thread_ready);                              //#
}
```

```
/**
 * Send an IPC message.
 *        Block until we can send the message or the timeout hits.
 * @param partner the receiver of our message
 * @param t a timeout specifier
 * @param regs sender's IPC registers
 * @return sender's IPC error code
 */
PRIVATE
Ipc_err Thread::do_ipc (bool have_send, Thread *partner,
                        bool have_receive, Sender *sender,
                        L4_timeout t, Sys_ipc_frame *regs)
{

  bool dont_switch = false;                                              //#

  if(have_send)
    {
      Ipc_err err = try_handshake_receiver(partner, t, regs);

      if(EXPECT_FALSE(err.has_error()))
        return err;

      assert(!(state() & Thread_polling));

      partner->ipc_init(this);

      // mmh, we can reset the receivers timeout
      // ping pong with timeouts will profit from it, because
      // it will require much less sorting overhead
      // if we dont reset the timeout, the possibility is very high
      // that the receiver timeout is in the timeout queue
      if (partner->_timeout && partner->_timeout->is_set())              //#
        {
          partner->_timeout->reset();                                    //#
          partner->set_timeout(0);                                       //#
        }

      Ipc_err ret = transfer_msg(partner, regs);

      if (Config::deceit_bit_disables_switch &&                          //#
          regs->snd_desc().deceite())                                    //#
        dont_switch = true;                                              //#

      // partner locked, i.e. lazy locking (not locked) or we own the lock
      assert(!partner->thread_lock()->test()
             || partner->thread_lock()->lock_owner() == this);


      if(EXPECT_FALSE(ret.has_error() || !have_receive))
        {
          // make the ipc partner ready if still engaged in ipc with us
          if(partner->in_ipc(this))
            {
              wake_receiver(partner);
              if(!dont_switch)                                           //#
  partner->thread_lock()->set_switch_hint(SWITCH_ACTIVATE_LOCKEE);//#
            }

          partner->thread_lock()->clear_dirty();

          state_del (Thread_ipc_sending_mask
                     |Thread_transfer_in_progress
                     |Thread_ipc_in_progress);
```

```
        return ret;
      }

    partner->thread_lock()->clear_dirty_dont_switch();
    // possible preemption point

    if(EXPECT_TRUE(!partner->in_ipc(this)))
      {
        state_del (Thread_ipc_sending_mask
                   |Thread_transfer_in_progress
                   |Thread_ipc_in_progress);

        return Ipc_err::Secanceled;
      }

    wake_receiver(partner);

  }
else
  regs->msg_dope(0);                                                    //#

assert(have_receive);
if (state() & Thread_cancel)                                            //#
  {
    state_del(Thread_ipc_mask);                                         //#
    return Ipc_err::Recanceled;                                         //#
  }

//% The lines below can be found in our model in a slightly altered version,
//% namely in the becoming
prepare_receive_dirty (sender, regs);

while (EXPECT_TRUE
       ((state() & (Thread_receiving | Thread_ipc_in_progress | Thread_cancel))
        == (Thread_receiving | Thread_ipc_in_progress)) )
  {

    Sender *next = 0;                                                   //#

    if(EXPECT_FALSE((long)*sender_list()))                              //#
      {

        if(sender) // closed wait                                       //#
          {

            if(sender->in_sender_list()                                 //#
               && this == sender->receiver()                            //#
               && sender->ipc_receiver_ready(this))                     //#
              next = sender;                                            //#
          }
        else // open wait                                               //#
          {

            next = *sender_list();                                      //#
            if(!next->ipc_receiver_ready(this))                         //#
              {
                next->sender_dequeue_head(sender_list());               //#
                Proc::preemption_point();                               //#
                continue;                                               //#
              }
          }
      }

    assert(cpu_lock.test());                                            //#

    //% We will skip the following conditional statement as it only
```

```
    //% deals with the scheduling of the partner and we have not
    //% modelled scheduling.
    if(EXPECT_FALSE((long) next))                                          //#
      {

        assert(!(state() & Thread_ipc_in_progress)                         //#
               || !(state() & Thread_ready));                              //#

        // maybe switch_exec should return an bool to avoid testing the
        // state twice
        if(have_send) {                                                    //#

          assert(partner);                                                 //#
          assert(partner->sched());                                        //#
        }

        if(EXPECT_TRUE(have_send && !dont_switch                           //#
                       && (partner->state() & Thread_ready)                //#
                       && (next->sender_prio() <= partner->sched()->prio()))) //#
          switch_exec_locked(partner,  Context::Not_Helping);              //#
        else                                                               //#
          {
            Proc::preemption_point();                                      //#
            assert(cpu_lock.test());                                       //#
            if(have_send && (partner->state() & Thread_ready))             //#
              partner->ready_enqueue();                                    //#
            schedule();                                                    //#
          }

        assert(state() & Thread_ready);                                    //#
      }

    else if(EXPECT_TRUE(have_send && (partner->state() & Thread_ready)))
      {
        if(!dont_switch)                                                   //#
          switch_exec_locked(partner,  Context::Not_Helping);              //#
        else                                                               //#
          partner->ready_enqueue();                                        //#
      }
    else                                                                   //#
      goto_sleep(t, regs);                                                 //#

    have_send = false;                                                     //#
  }

assert(!(state() & Thread_ipc_sending_mask));                              //#

if (EXPECT_FALSE((long)_timeout)) {                                        //#
  _timeout->reset();                                                       //#
  set_timeout(0);                                                          //#
}

// if the receive operation was canceled/finished before we
// switched to the old receiver, finish the send
if(have_send && (partner->state() & Thread_ready))                         //#
  {
    if(!dont_switch)                                                       //#
      switch_exec_locked(partner,  Context::Not_Helping);                  //#
    else                                                                   //#
      partner->ready_enqueue();                                            //#
  }

// fast out if ipc is already finished
if(EXPECT_TRUE((state() &                                                  //#
                (~(Thread_fpu_owner|Thread_cancel))) == Thread_ready))     //#
  return  get_ipc_err (regs);
```

```
  //% We will never reach this point as we do not model long IPC, we
  //% therefore assume that the if-statement directly above this
  //% statement is always true.
  handle_long_ipc();                                                    //#

  // abnormal termination?
  if (EXPECT_FALSE (state() & Thread_ipc_receiving_mask))               //#
    {
      // the IPC has not been finished.  could be timeout or cancel
      // XXX should only modify the error-code part of the status code

      if (state() & Thread_busy)        // we've presumably been reset!   //#
        commit_ipc_success (regs, Ipc_err::Reaborted);                  //#

      //else
      if (state() & Thread_cancel)      // we've presumably been reset!   //#
        {
#if 0                                                                   //#
          if (state() & Thread_transfer_in_progress)                    //#
            {
          LOG_MSG(this,"REAB2");                                        //#
            commit_ipc_success (regs, Ipc_err::Reaborted);              //#
            }
          else                                                          //#
            {
          LOG_MSG(this,"RECA1");                                        //#
#endif                                                                  //#
            commit_ipc_success (regs, Ipc_err::Recanceled);             //#
#if 0                                                                   //#
            }
#endif                                                                  //#
        }

      else                                                              //#
        commit_ipc_success (regs, Ipc_err::Retimeout);                  //#
    }

  state_del (Thread_ipc_mask);                                          //#

  return get_ipc_err (regs);            // sender puts return code here   //#
}

PRIVATE inline NEEDS ["map_util.h", Thread::copy_utcb_to,
                      Thread::unlock_receiver]
Ipc_err Thread::transfer_msg (Thread *receiver,
                              Sys_ipc_frame *sender_regs)
{
  //% In our model, we assume that the transferring of a message always
  //% succeeds so all lines in this function have been abstracted away
  //% in our model.

  if (!Config::deceit_bit_disables_switch && sender_regs->snd_desc().deceite())//#
    panic ("deceiving_ipc");    // XXX unimplemented                    //#

  Sys_ipc_frame* dst_regs = receiver->rcv_regs();                       //#
  const L4_msgdope ret_dope(sender_regs->snd_desc(),                    //#
      Sys_ipc_frame::num_reg_words(), 0);                               //#


  dst_regs->msg_dope (ret_dope);        // status code: rcv'd 2 dw       //#

  // copy message register contents
  sender_regs->copy_msg (dst_regs);                                     //#

  copy_utcb_to (receiver);                                              //#
```

```
// copy sender ID
dst_regs->rcv_src (id());                                                    //#

// fast out if only register msg
if(EXPECT_TRUE(sender_regs->snd_desc().is_register_ipc()))                   //#
    return 0;

//% We will never reach this point as we do not model long IPC and thus
//% assume that <is_register_ipc()> always returns true.

// because we do a longer ipc with preemption points, we set a corrent ipc state
state_add_dirty(Thread_send_in_progress |                                    //#
                Thread_ipc_in_progress| Thread_transfer_in_progress);        //#

Mword ret = 0;                                  // status code: IPC successful
// we need the lock here definitly
receiver->thread_lock()->lock_dirty();                                       //#

// short flexpage mapping to register/rmap receiver
if(EXPECT_TRUE (sender_regs->snd_desc().msg() == 0                           //#
                && (dst_regs->rcv_desc().is_register_ipc()                   //#
                    || dst_regs->rcv_desc().rmap())))                        //#
  {
    assert(sender_regs->snd_desc().map());                                   //#

    if (EXPECT_FALSE (! dst_regs->rcv_desc().rmap()) )                       //#
      // rcvr not expecting an fpage?                                        //#
      {
        dst_regs->msg_dope_set_error(Ipc_err::Remsgcut);                     //#
        ret = Ipc_err::Semsgcut;                                             //#
      }
    else                                                                     //#
      {
        Proc::sti();                                                         //#

        dst_regs->msg_dope_combine                                           //#
          (fpage_map(space(),                                                //#
                     L4_fpage(sender_regs->msg_word(1)),                     //#
                     receiver->space(), dst_regs->rcv_desc().fpage(),        //#
                     sender_regs->msg_word(0),                               //#
                     L4_fpage(sender_regs->msg_word(1)).grant()));           //#

        Proc::cli();                                                         //#

        if (dst_regs->msg_dope().rcv_map_failed())                           //#
          ret = Ipc_err::Semapfailed;                                        //#
      }
    return ret;                                                              //#
  }

Proc::preemption_point();                                                    //#

// else long ipc
prepare_long_ipc(receiver, sender_regs);                                     //#

Ipc_err error_ret;                                                           //#

assert(state() & Thread_send_in_progress);                                   //#
receiver->thread_lock()->clear_dirty();                                      //#

CNT_IPC_LONG;                                                                //#
Proc::sti();                                                                 //#
error_ret = do_send_long (receiver, sender_regs);                            //#
Proc::cli();                                                                 //#
```

```
  assert(receiver->thread_lock()->lock_owner() == this);                    //#

  assert(error_ret.has_error() || state()                                   //#
         & (Thread_send_in_progress | Thread_transfer_in_progress));        //#

  return error_ret;                                                         //#
}
```

Appendix J

C++: sender.cpp

```
//------------------------------------------------------------------------
// LEGEND:
//   //% : comments added by us.
//   //# : a line that has not been modelled.
//   //! : a line that has been added by us.
//------------------------------------------------------------------------

/** Sender in a queue of senders?.
    @return true if sender has enqueued in a receiver's list of waiting
            senders
 */
PUBLIC inline
bool
Sender::in_sender_list()
{
  //% We have created a simplified version of the sender list in our model,
  //% therefore the actual implementation below is not directly but
  //% indirectly reflected in our model.
  return sender_next;
}

PROTECTED
//PROTECTED inline NEEDS [<cassert>, "cpu_lock.h", "lock_guard.h",
//                        Sender::replace_node, Sender::tree_insert]
void Sender::sender_enqueue(Sender **head, unsigned short prio)
{
  //% We have created a simplified version of the sender list in our model,
  //% therefore the actual implementation below is not directly but
  //% indirectly reflected in our model.

  assert(prio <256);                                                    //#

  _sender_prio = prio;                                                  //#

  Lock_guard<Cpu_lock> guard (&cpu_lock);                               //#

  if (in_sender_list())                                                 //#
    return;                                                             //#

  _sender_l = _sender_r = 0;                                            //#
  _sender_parent = 0;                                                   //#
  sender_next = sender_prev = this;                                     //#

  Sender *p = *head;                                                    //#

  if(!p)                                                                //#
    {
```

```
      *head = this;                                                    //#
      return;                                                          //#
    }

  Sender *x = this;                                                    //#

  // bigger prio, replace top element
  // we dont handle the case same max prio here!
  if(_sender_prio > p->_sender_prio)                                   //#
    {
      replace_node(p);                                                 //#
      x = p; // ok, insert the old top element too                     //#
      *head = this;                                                    //#
    }

  x->tree_insert(*head);                                               //#
}

PUBLIC
//PUBLIC inline NEEDS [<cassert>, "cpu_lock.h", "lock_guard.h",
//                     Sender::remove_tree_elem, Sender::remove_head]
void Sender::sender_dequeue(Sender **head)
{
  //% We have created a simplified version of the sender list in our model,
  //% therefore the actual implementation below is not directly but
  //% indirectly reflected in our model.

  if (!in_sender_list())                                               //#
    return;                                                            //#

  Lock_guard<Cpu_lock> guard (&cpu_lock);                              //#

  // we are removing top (the sender element with the highest prio
  // so we need to calculate an new top element

  if(this == *head)                                                    //#
    *head = remove_head();                                             //#
  else                                                                 //#
    remove_tree_elem();                                                //#
  // mark as dequeued
  sender_next = 0;                                                     //#
}

// An special version, only to remove the head
// this is neccessary if the receiver removes the old know head
// after an unsuccessful ipc_receiver_ready.
PUBLIC
void Sender::sender_dequeue_head(Sender **head)
{
  //% Our

  if (!in_sender_list())                                               //#
    return;                                                            //#

  Lock_guard<Cpu_lock> guard (&cpu_lock);                              //#

  // we are removing top (the sender element with the highest prio
  // so we need to calculate an new top element

  if(this == *head)                                                    //#
    *head = remove_head();                                             //#

  // mark as dequeued
  sender_next = 0;                                                     //#
}
```

Appendix K

C++: receiver.cpp

```
//-----------------------------------------------------------------------------
// LEGEND:
//   //% : comments added by us.
//   //# : a line that has not been modelled.
//   //! : a line that has been added by us.
//-----------------------------------------------------------------------------

/** IPC partner (sender).
    @return sender of ongoing or previous IPC operation
 */
PROTECTED inline
Sender*
Receiver::partner() const
{
  return _partner;
}

// Interface for senders

/** Head of sender list.
    @return a reference to the receiver's list of senders
 */
PUBLIC inline
Sender**
Receiver::sender_list()
{
  //% We have created a simplified version of the sender list in our model,
  //% therefore the actual implementation below is not directly but
  //% indirectly reflected in our model.
  return &_sender_first;
}

// MANIPULATORS

/** Initiates a receiving IPC and updates the ipc partner.
    @param sender the sender that wants to establish an IPC handshake
 */
PUBLIC inline NEEDS [Receiver::set_partner,
                     Receiver::rcv_regs,
                     Receiver::clear_pagein_request,
                     "entry_frame.h", "sender.h", "l4_types.h"]
void
Receiver::ipc_init (Sender* sender)
{
  set_partner (sender);
  rcv_regs()->rcv_src (sender->id());                                    //#
  state_add_dirty(Thread_transfer_in_progress);
```

```
}

PROTECTED inline
 void Receiver::prepare_receive_dirty(Sender *partner,
                              Sys_ipc_frame *regs)
{
  // cpu lock required, or weird things will happen
  assert(cpu_lock.test());                                              //#

  set_rcv_regs (regs);  // message should be poked in here              //#
  set_partner (partner);

  state_change_dirty(~(Thread_ipc_sending_mask|Thread_transfer_in_progress),
                     Thread_receiving | Thread_ipc_in_progress);
}

PUBLIC inline
bool
Receiver::in_ipc(Sender *sender)
{
  Mword ipc_state = (state() & (Thread_ipc_in_progress
                              | Thread_cancel
                              | Thread_transfer_in_progress));

  if(EXPECT_TRUE
     ((ipc_state == (Thread_transfer_in_progress | Thread_ipc_in_progress))
      &&  _partner == sender))
    return true;

  return false;
}

/** Set the IPC partner (sender).
    @param partner IPC partner
 */
PROTECTED inline
void
Receiver::set_partner(Sender* partner)
{
  _partner = partner;
}

/** Return whether the receiver is ready to accept a message from the
    given sender.
    @param sender thread that wants to send a message to this receiver
    @return true if receiver is in correct state to accept a message
                right now (open wait, or closed wait and waiting for sender).
 */
IMPLEMENT inline NEEDS["std_macros.h", "thread_state.h", Receiver::partner]
bool
Receiver::sender_ok (const Sender *sender) const
{
  unsigned ipc_state = state() & (Thread_receiving |
                                  //              Thread_send_in_progress |
                                  Thread_ipc_in_progress);

  // If Thread_send_in_progress is still set, we're still in the send phase
  if (EXPECT_FALSE (ipc_state != (Thread_receiving | Thread_ipc_in_progress)))
    return false;

  // Check open wait; test if this sender is really the first in queue
  if (EXPECT_TRUE(!partner()
                  && (!_sender_first || sender == _sender_first)))
    {
      assert(!(state() & Thread_polling));                              //!
      return true;
```

```
    }

  // Check closed wait; test if this sender is really who we specified
  if (EXPECT_TRUE (sender == partner()))
    {
      assert(!(state() & Thread_polling));                              //!
      return true;
    }

  return false;
}
```

Bibliography

[AFH+06] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. Technical report, Microsoft Research, 2006. 2.1.4, 4

[AMD05] Inc. Advanced Micro Devices. AMD secure virtual machine architecture reference manual, May 2005. 7

[Aus05] National ICT Australia. NICTA L4-embedded kernel reference manual, version nicta n1. Technical report, Kensington Research Laboratory, Sydney, Australia, November 2005. 2.2.3

[BK05] Marcus Völp Bernhard Kauer. L4.Sec preliminary microkernel reference manual. Technical report, Technische Universität Dresden, Dresden, Germany, October 2005. 2.2.1

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview, October 2004. 2.1.2

[BMS+97] Ricky W. Butler, Paul S. Miner, Mandayam K. Srivas, Dave A. Greve, and Steven P. Miller. A new bitvectors library for pvs, January 1997. 3.2.4

[Cor05] Intel Corporation. Intel virtualization technology specification for the IA-32 intel architecture, April 2005. 7

[Cur92] Paul Curzon. Of what use is a verified compiler specification? Technical Report 274, University of Cambridge, Computer Laboratory, November 1992. 2.1.2

[CV98] Judith Crow and Ben Di Vito. Formalizing space shuttle software requirements: four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, 1998. 2.4.1

[Dau03] Matthias Daum. Development of a semantics compiler for C++. Master's thesis, Dresden University of Technology, September 2003. 2.5.2

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. 2.5.3

[DL05] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, May 2005. 2.1.2

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994. 2.1.2

[End05] Endrawaty. Verification of the fiasco IPC implementation. Master's thesis, Dresden University of Technology, March 2005. 2.5.3

[FL06] Manuel Fähndrich and James R. Larus. Language support for fast and reliable message-based communication in singularity OS, 2006. 2.1.4

[HAB+06] Galen C. Hunt, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James R. Larus, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. Sealing OS processes to improve dependability and security. Technical report, Microsoft Research, April 2006. 2.1.4

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, C. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998. 2.3.2

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Modular system programming in MINIX 3. *j-LOGIN*, 31(2):19–28, April 2006. 2.1.3

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 217–230, Berkeley, CA, USA, 2001. USENIX Association. 2.3.4

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In C. Hankin, editor, *Proceedings of European Symposium on Programming (ESOP '98)*, number 1381 in Lecture Notes in Computer Science, pages 105–121. Springer-Verlag, 1998. 2.1.2, 2.5.1

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, 1997. ACM Press. 1, 2.2.1

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association. 2.1.3

[HJvdB99] Marieke Huisman, Bart Jacobs, and Joachim van den Berg. A case study in class library verification: Java's vector class. In A. Moreira and D. Demeyer, editors, *Object-Oriented Technology: ECOOP'99 Workshop Reader*, volume 1743, pages 109–110, Lisbon, Portugal, 1999. Springer-Verlag. 2.1.2

[HLA+05] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An overview of the singularity project. Technical report, Microsoft Research, October 2005. 2.1, 2.1.4

[HPV00] Klaus Havelund, John Penix, and Willem Visser. Spin model checking and software verification, 7th international spin workshop, stanford, ca, usa, august 30 - september 1, 2000, proceedings. In *SPIN*, volume 1885 of *Lecture Notes in Computer Science*. Springer, September 2000. 2.5.3

[HT03] Michael Hohmuth and Hendrik Tews. The semantics of C++ data types: Towards verifying low-level system components, July 2003. 2.5.2

[HTS03] M. Hohmuth, H. Tews, and S. Stephens. Applying source-code verification to a microkernel — the VFiasco project, May 2003. 1, 2.1, 2.1.3, 2.5

[IFI05] IFIP. Dependable computing and fault tolerance. http://www.dependability.org/wg10.4, 2005. 1

[Int06] SRI International. SRI International, an independent, non-profit R&D organization dedicated to client success. http://www.sri.com/, December 2006. 2.4.1

[JMG+02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C, 2002. 2.1.1

[JS06] Scott Doerrie Jonathan Shapiro, Swaroop Sridhar. BitC language specification, June 2006. 2.1.1

[KK06] Rafal Kolanski and Gerwin Klein. Formalising the l4 microkernel api. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theroy Symposium*, pages 53–68, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. 2.5.5

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999. 2.1.2

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, December 1993. 2.2.2, 2.3.3

[Lie95] Jochen Liedtke. On μ-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM Press. 1, 2.1.3, 2.2.3

[Lie96] Jochen Liedtke. L4 reference manual - 486, Pentium, Pentium Pro, version 2.0. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, September 1996. 1, 2.2.1

[Lie99] Jochen Liedtke. L4 nucleus version x reference manual, x86, version x.0. Technical report, Universität Karlsruhe, September 1999. 2.2.1

[Lie04] Jochen Liedtke. L4 experimental kernel reference manual, version x.2. Technical report, Universität Karlsruhe, June 2004. 2.2.1

[Met96] Werner Metterhausen. L3 referenzhandbuch, 1996. 2.2.1

[MHH02] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems, December 2002. 2.1.4

[MS95] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proceedings of the Workshop on Industrial Strength Formal Specification Techniques (WIFT'95)*, Boca Raton, Florida, 1995. 2.4.1

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 2.1.2, 2.5.5

[Owr] Version September Owre. PVS language reference. 2.1.2

[Pet02] Michael Peter. Leistungs-analyse und -optimierung des L4Linux-systems. Master's thesis, Technische Universität Dresden, June 2002. 2.3.7

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974. 2.1.4

[Reu05] René Reusner. Implementierung eines echtzeit-ipc-pfades mit unterbrechungspunkten für L4/Fiasco. Master's thesis, Technische Universität Dresden, July 2005. 2.3.6, 2.3.7, 2.5.4, 5.2

[RJO+89] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press. 2.1.3

[Sch01] Sebastian Schönberg. A user mode l4 environment. In *2nd International Workshop on Microkernel-based Systems*, Chateau Lake Louise, Banff, AB, Canada, October 2001. 2.3.2

[SN01] Ib Sorensen and David Neilson. B: towards zero defect software, 2001. 2.1.2, 2.5.5

[Ste04] Udo Steinberg. Quality-assuring scheduling in the fiasco microkernel. Master's thesis, Technische Universität Dresden, March 2004. 2.3.3

[Tew00] H. Tews. A case study in coalgebraic specification: Memory management in the FIASCO microkernel. Technical Report TPG2/1/2000, SFB 358, April 2000. 2.5.1

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007. 2.5.5

[Uni06] Johns Hopkins University. The coyotos secure operating system, 2006. 2.1.1

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299+, 2001. 2.1.2, 6.2.4

[Vit03] Ben L. Di Vito. Application of strategies/tactics in higher order logics. In *Proceedings of STRATA 2003, First International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics*, July 2003. 2.4.1