University of Beira Interior
Computer Science Department

Master of Science Thesis

Towards a formally designed and verified embedded operating system: case study using the B Method

by André Brito Passos

Supervisor: Simão Melo de Sousa
Co-supervisor: José Miguel Faria

Coimbra, August 22, 2009

To Marina, my family and friends

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thanks to Professor Simão Melo de Sousa, who always helped me giving me the precise directions to conclude this work.

In second to Critical Software. It was an extraordinary experience the development of this thesis in a company like Critical. In particular I would like to thanks to my co-supervisor and good friend, José Miguel Faria, for all the support, help and freedom of thinking. To the formal methods team, Pedro and Ricardo, always by my side during the time spent in the developing of this work. A word to Pedrosa for all the time wasted explaining me some doubts about operating systems. To all the colleges in Critical Software, I prefer not to say names because all were outstanding.

To my girlfriend, Marina Taramelo, for all the support and encouragement during these hard times for both. At last, to my friends and my family, namely my mother, Elisabete and father, Daniel, who gave me the opportunity of study.

For all of you: Thanks!

Chapter 1

Introduction

The formal development of software systems is the subject of this thesis. More precisely, the development of a verified microkernel using the formal specification language, B Method. This chapter introduces the main concepts of this work.

1.1 Context

In some contexts formal methods are considered as the best means available for developing safe an reliable systems. One well-known researcher, Bertrand Meyer, express the necessity and efficiency of formal methods when he wrote about software engineering: *"It is clear to all the best minds in the field that a more mathematical approach is needed for software to progress much"* [29]. The goal and the results of this work can be seen as an other proof of evidence of the previous fact.

1.1.1 Motivation

According to R.W. Butler [23], *"formal methods refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems"*. The phrase "mathematically rigorous" means that specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (i.e., each step follows from a inference rule and hence can be checked by a mechanical process). A method is formal if it has a sound mathematic basis, typically given by a formal specification language. The value of formal methods is that they provide means to symbolically examine the entire state space of a digital design (whether hardware or software) and establish a correctness or safety property that is valid for all possible inputs. However, this is rarely done in practice today (except for the critical components of safety critical systems) because of the enormous complexity of real systems.

The activities of modeling and formal reasoning are supposed to be performed before undertaking the effective coding of a computer system, so that the software in question can be correct by construction. Formal methods can be very helpful at this point. But, why formal methods are not widely used instead of typically software engineering? Formal methods should be used by people who have realized that the program development process they using is inadequate, or in industry when the customer pretend to use such methodology. Another reason that may

recommend the use of formal methods is that fact that certain standards like Common Criteria (EAL 5-7), EN 50128 (SIL 4) and DO-178B (Design Assurance Level A and B), recommend or requires the use of formal methods. At this stage is possible to question why to bother about formal methods and not use tests and simulation instead. Limitations related to tests, like the problem of how to exactly define a good test and the representativeness are difficult properties of achieve. Nevertheless, exhaustiveness for the general case of all the set of possible values is infinite, so, total coverage is an impossible task. Like W. Dijkstra once said *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence"*. Software correctness is the holy grail, being chasing for some time. Formal methods are not the "perfect solution" but can provide an effective help with in the dealing with bugs.

There are cases where the use of formal methods is a requirement, but which formal method should we use? The choice of a formal method is not easy. Partly because there are many. To decide which formal method to use, a number of questions should be posed:

- Is there any theory behind your formal method?
- What kind of language is your formal method using?
- Does there exist any kind of refinement mechanism associated with your formal method?
- What is your way of reasoning with your formal method?
- Do you prove anything when using your formal method?
- Have you got an efficient automatic prover?

These questions should work like a conductive method to help in the quest of the adequate formal approach. Several other criteria should be analyzed, like for example, the experience in the use of a specific formal method. Other important criteria is the usage purpose. Some kind of formal methods have a large and successful experience in specific areas and types of certification. For example, the B Method is used in the Railway System sector and in the process of certification SIL 4 (see for instance [37][7]).

1.1.2 The choice of the B Method

Engineering disciplines require tools for engineers to reason about the possible solutions for their problems. It is very common to see mechanical engineers working with tools to design, make some measures and experiments before passing to the "real world". These tools are mathematical tools that work with models to make measurements about the work. Strangely, or not, computer science evolution happened very quickly, and some steps were not so well done.

Formal methods provide a large variety of tools to help engineers to reason about the system. These tools are used to specify a system's desired behavioral and structural properties, like done in other engineering disciplines. The reveal of ambiguities, incompleteness and inconsistency in systems are the main objectives in the usage of such tools. Properties that characterize well formal methods are: (i)*exhaustiveness*, the formal reason over the possibly infinite set of values; (ii)*rigor*, well established mathematical foundations; and (iii)*adequacy*, provided tools and techniques are evaluated as adequate means for providing evidences of reliability. The use of formal methods to reason about systems is called formal specification.

The existence of a rather large set of tools with different and more or less restricted domain of application is a convenient property. This means that depending on the purpose of the application, one or other formalism can be better applied. In the following sequel some considerations will be done about the several formalisms. It is not pretended to make a state of the art in the area, only to give a brief perspective of what exists.

Formal specification can be classified in mostly two families. In algebraic specifications give more importance to the data being manipulated. This means that the behavior of the

system is expressed by the data of the system, how this data evolutes along the time, or how the different data relates to each other. On the other hand, it is possible give more meaning to the operations of the system. The behavior is expressed by the operations, internal mechanisms or by the system actions. In this type of formalism, the specification is the modifications that operations can perform in the internal state this is called a model based specification. Other types of organization can be given, for example, based on their prime domain of application, i.e., either specialized in sequential programs, or focusing on concurrent and distributed systems.

Z, VDM and B are well established and representatives of specifications based on models. Petri nets, CCS, CSP or event-B are also model based but with their prime domain of application being concurrent and distributed systems. On the other hand, for algebraic specifications, ACT-ONE, CLEAR, OBJ, SPECWARE, etc. can be used for sequential programs and LOTOS, etc. for concurrent and distributed systems.

Temporal requirements plays an important role in some systems. Model based approaches offers a quite variety of solutions for express temporal requirements. However, transitions between states change, requiring a temporal requirement for a transition to proceed. Tools that can be used with this type of formalism are Lustre, Esterel, Scade, UPPAAL, etc..

As previously said, formal methods can be used to achieve correctness. To verify this property, normally, desired properties are confronted with the specification of the system/algorithm. Correctness of a system can be verified by writing proofs by hand or use more or less automatic tools. Model checkers, Proof systems (also known as theorem provers) and proof assistants are the main groups used to mechanically verify the correctness of a system.

Proof assistances generally require some user intervention. They can check whether a given proof is valid, according to the rules of deduction of the logic concerned, or, given a theorem and its premisses, try to discover a proof such that the theorem follows from that premisses [16]. Examples of proof systems are Coq, HOL, Isabelle and PVS.

Model checking is an alternative to proof systems. They are specially suitable for the analysis (modeling) of concurrent systems. The properties are expressed in temporal logical formulas and the verification is performed exploring the possible transitions of the system. Some of the advantages, comparing with the previous ones, are the automatic demonstration of the properties without any user intervention and the exhibition of a counter-example. The disadvantages are the problem of state space explosion and the fact that model checker do not know how to deal with infinite sets. Examples of model checkers are SMV, SPIN, Kronos, UPPAAL, etc..

Other formalism that in our days is being used for several systems is design-by-contract. The logical annotation of the program using Hoare Logic. Each function is annotated with a pre-condition and a post-condition, like a contract. Then, the source code is confronted with the contract to perform validation. The notion of invariant is also present in this type of formalism, to express the global properties of the complex components of the system. Usually, this type of verification is very connected to the language. With C we have Frama-C (with jessie plug-in), for Java we have JML, for *C♯* we have *Spec♯* and for Ada we have Ada/Spark.

The goal of this work is to start with an high-level abstraction, defining the properties of the system and them refine only part of the system to be developed. It was pretended a methodology that covers all the stages of the development process and the B Method is in concordance with this criteria. Some previous experience and some examples showed that is possible to use the B language to perform such work [**?**] [22]. The B Method does not deal well with concurrency and real time requirements. However, the microkernel to be developed it is not a real time kernel and the concurrency problems will not be part of the problem to be analyzed.

1.1.3 Cautionary notes about formal methods

> *Even perfect program verification can only establish that a program meets its specification. [...] Much of the essence of building a program is in fact the debugging*

of the specification. [18]

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

Formal methods provides a close relationship with requirements. Validation can start sooner and more easily auditable. This property really strengthens the role of formal methods in the development process. More sooner the validation starts more quality the final product will have. A typical problem pointed to formal specification is the fact that an abstraction can be proved correct. But is this specification really doing what is supposed to do? Are the properties well defined to that specification? These questions are now, more or less, answered using animators. This type of tools can give to the person which is specifying the system a certain view and understanding of the specification. Animators permits to animate, run the specification. So, it helps solving the problem of what is the specification really doing.

As seen in the previous section 1.1.2, different formalisms can be applied in different fields of application. A tool that permits to solve in an efficient and satisfactory way all type of problems is still not known.

1.2 Definition of the work

The aim of this work is to produce a embedded microkernel using the B Method. To perform this objective the work was divided into three stages. In the first stage it is necessary to provide a complete requirements analysis and a specification of a secure partitioning microkernel. This is done using the tool Atelier B and ProB for animation.

The second stage is composed by a complete development of part of the secure partitioning microkernel. The chosen part was the partitioning information flow policy (PIFP). Starting with a high level specification, the partition information flow policy is the refined until be possible to generate automatically code.

The third and final part is the integration of the code generated with a chosen microkernel. In this part is necessary to perform a verification/testing over the microkernel with the generated code. Figure 1.1 illustrates the diagram of the three stages of the work.

1.2.1 Objectives

The overall objective is the complete development of a secure partitioning microkernel using the "correct by construction" paradigm (more precisely the B Method). The other objectives are:

- Perform a complete specification using the B Method;
- Use and exploration of the B tools (ProB and Atelier B);
- Proof of correctness for the system;
- Reach a level in the development process where it is possible to automatically generate code;
- Integration of part of the microkernel in a real word system;
- Explore the use of the chosen formalism in the context of microkernel development.

1.2.2 Contribution

The microkernel is the principal part of an operating system. Being the operating system, the place where applications run, it becomes easy to understand that a special careful in the

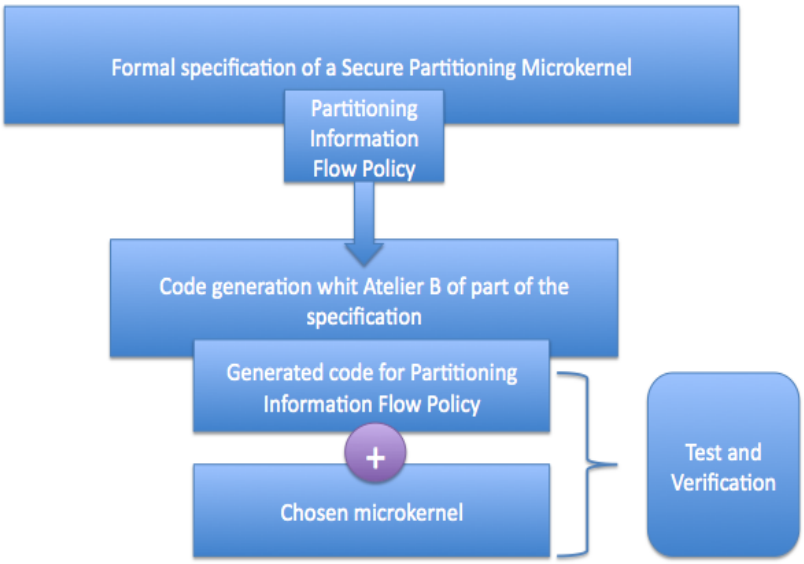

Figure 1.1: Diagram with the complete work

development of a microkernel is very important. It make essential the usage of formal methods to help in the construction of a microkernel. The contribution of this work is to perform a complete formal development of the microkernel. As secondary contribution, the transfer of knowledge from the university to the company Critical Software.

1.3 Outline

This thesis is organized in six chapters. Chapter 1 introduces the main concepts that are subject of study in this thesis. Chapter 2 gives a brief overview about software engineering with formal methods and presents the B Method. Chapter 4 gives an overview about the state of the art involving formal verification for kernels. Chapter 3 presents the work that are to be developed, giving an overview of what is a secure partitioning microkernel and the proposed solution. Chapter 5 presents the work developed, a formal model of the secure partitioning microkernel and the complete development of part. Integration with the target microkernel is also presented. Chapter 6 concludes this thesis by describing the most relevant conclusions of the work herein described.

Chapter 2

Software Engineering and the B Method

In this chapter a description of the B language and methodology will be provided (based on [6]). The aim of this chapter is to prepare the reader for the next following chapters. The goal is not to give a complete overview of the language but only to illustrate the main concepts. For a full description of the language, methodology and mathematical foundations it is recommended to consult [3].

This chapter will be organized in the following way, in the first section a presentation of what is modeling, how to make a good model and other normal descriptions when dealing with this type of formalism. Next, in section 2.2, it is described what is the formal development life cycle, showing the different phases.

After talking about the software development process, it is described the B language (section 2.3). Concepts like abstract machine, refinement and implementation will be explained. The B methodology also will be presented.

An example for making more clear the concepts around the language is given in section 2.4. It is a quite simple and easy to understand, but at the same time complete enough for show the application of the B Method.

In section 2.5 it is presented a description of the tools normally used in the B language. Each tool has a different utility in the development life cycle, the usage of each one in the different parts can give a precious help in the development process.

In the end (section 2.6), a summary for the all chapter will be given.

2.1 System Modeling and Design

Computer Science, like other engineering disciplines, requires a complete view of what is to be done. Engineering design is used to each a complete view and understanding of the problem.This technique is composed by two concerns. The first one is the world view, how the system should behave in the real world. Normally in this phase we are thinking about the system, about its properties. The second one is how to use the technology to achieve a real world view. In this stage we are already thinking in one possible implementation to solve our problem. A technique that deals with both concerns is called modeling.

Usually, in the classical approach a very competent domain-expert person whose main source of inspiration comes from previous systems that he has done is who makes the engineering design. Preceding experiences brings some strengths but also some weaknesses like old and bad habits. Validation of such studies is accomplished by a number of tests and reports to see if the results are consistent with previously decided criteria. An approach that completes the former is namely based on formal (mathematical) description, refinement techniques, decomposition and mechanized proofs. In this approaches the requirements document sometimes are not a valid input because generally it is a mental implementation of the desired system, one good input is a document with the addition of the expected properties for the system.

The main idea of this new way of study the system consists in using models with a certain correctness criteria, with which we can use mathematical simulation, which is not very different from the usual method of tests, except for its exploitation technique. The difference using mathematical simulation is that we will not look to the result to see if it respects the criteria, but to prove that the model respects it. Like J.R. Abrial refers [23], this change in the mode of exploitation of the "simulation" has a number of interesting consequences:

- *Rather than using a simulation language in order to build our model under the form of a simulation program, we are going to use directly a mathematical notation, which will allow us to represent the model in a way that will be more convenient than a simulation program (for expressing the statements and for proving them).*
- *Rather than limiting ourselves to a single simulation program, as is usually the case, we can very well use a series of embedded models that are supposed to be refinements of each others. In this way, the various criteria to prove can be accumulated and proved immediately at their right level of abstraction: the proof process accompany the model elaboration process.*
- *Once a model has reached a dangerous level of complexity (so that the proofs might become cumbersome), it will be possible to decompose it into separate model: the architecture is born.*

So, what is a model? Everyone has a vague idea of what is a model, but is important to give a precise notion to define a formal construction. Models provide a demonstration of some properties (mathematical properties) of a system without necessarily building the system. Modeling requires an abstraction, at a certain level, of the future system. The idea is to put ourselves mentally above the system and try to imagine what we could observe from there. Simple properties or laws that sometimes are forgotten became obvious and these are the ones that we want to catch at a first view. We can now have our first model, a very simple one dealing only in the basics of the system.

The advantages of modeling in the early stages are quite obvious, mistakes can now be found sooner, so a lot of time can be spared. Another advantage is that we can reach a point in our model where we find some inconsistencies, which can tell us that we are not understanding well the problem possibly because some requirements are missing.

These advantages of modeling are not a surprise inside engineering disciplines. In fact, it is quite normal for a mechanical engineer to make a first model and then make some measurements over the model to see if the behavior is the expected one before pass to the phase of building. Models using a mathematical base notation provide us with the capacity of making measurements. Basically what we want to model using mathematical notation is the important characteristics of some "thing" that we will next build or implement, to reveal what our "thing" will look like, to help us understanding how our "thing" will behave, and finally to make sure that our "thing" achieves the required behavior. The main purpose of modeling is to construct correct software by construction and this technique is called software modeling.

Software can go wrong and cause very expensive damages. Ariane 5 is a good example of software failures with expensive results. It took the European Space Agency ten years and

seven billion of Euros to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business. A research taken by Le Lann concludes that the real causes of the failure are faults in the capture of the overall Ariane 5 application/environment requirements, and faults in the design and the dimensioning of the Ariane 5 on-board computing system. These faults result from not following a rigorous system engineering approach, such as applying a proof-based system engineering method (for more details see [26]).

In software modeling we are centered only in the expect behavior of the system. Therefore, we try to achieve an abstraction of the system and focus on what we desire for that system and not in how we will implement those desired functionalities. Like mentioned before the notion of abstraction can be seen as observing the system from a higher view. In these observations we need to have two notions: a notion of objects (static notion) and a notion of movement (dynamic notion). The first one refers to the state of the model and the second one the events or operations that may occur and that we are able to observe. The properties connected to the state of the system are called safety properties and those related to events or operations are called liveness properties. The two properties need to be proved; they have no reason, a priori, to be coherent. Software modeling can be differentiated in three phases:

1. **Specification**: is concerned with what the system should do. There is no requirement that a specification needs to be executable. Specifications model the user or world view of the system, not the internal behavior;

2. **Refinement or Design:** is the process of transforming the specification towards an implementation. This must be achieved without changing the world view;

3. **Implementation**: is the final refinement or design step in which we make the model executable. Implementation is concerned with how the system is to be realized.

In order to clarify the three phases presented it will be used an analogy called the *parachute paradigm*. Suppose that we are suspended in the air by a parachute, we are looking to our system behavior from a upper level. From that view we only see some properties of the system, this properties need to be proved to see if we are thinking correctly. At this time that we finished our proves and we have seen all the properties form that level, we can step down a little bit and thus observe some other interesting things, that is, a more precise properties which we were not able to distinguish before because we are in a too higher level. New properties (safety and liveness) will appear and and these new observations should not invalidate the previous ones, instead they should rather make them more accurate. Now we can step down a little bit more and so on. This is called the *parachute paradigm*.

What we have just briefly view here is the notion of what we call refinement of our models in order to have a gradually more precise view of all the facts of the system. Comparing with the tree phases of system modeling when we are in the upper level we are dealing with the specification phase. Every time we make our model more precise, we are refining the model, making him more accurate. The last step is the when we are in the lower level, the implementation phase. One import property to stress at this point is that all the properties that are proved at a certain level remain valid along the path leading gradually to the ground.

An important fact is when we start modeling a certain system, we are at a higher level of abstraction so is very common to start with a single and very simple model. When we start refining the model, more properties appear and the model starts becoming complex and large. This is the right moment to envisage decomposing the model into several sub-models. The role of the decomposition is clear. Once separated from the main body, a sub-model can be developed further independently from the rest of the system, which becomes its so-called environment. It is important to notice that decomposition is the process used here, this does not mean that composition is impossible, it only means that we cannot compose existing parts without the

supervision of a decomposition process. This is the only way to ensures that some global laws are maintained while decomposing parts are working together.

Abstraction problem is the tentative of achieving a good abstraction level. At the very beginning, when we start modeling, if we are thinking in loops or sequential composition, the best thing to do is stop and start all again. We are going in the wrong direction, not thinking correctly in terms of abstraction. The best way to see if we are in the right level of abstraction is to think about the problem like an interface, this is, do not think in details of how to resolve the problem but think in what to do. Of course that be in the right abstraction level is a difficult task and that most people refers as the major art for modeling correctly.

Modeling is an art, and like other arts we can be good at it or not. I truly believe that getting the write level of abstraction and modeling correctly is a task that, like other, tasks requires a lot of work not because of its difficulty, but because it is different from what is usually done. As a result, it is a question of mentality rather than difficulty.

2.2 Formal Development Life Cycle

To build formally developed applications is necessary to follow a precise and formal development life cycle, like the one mentioned in [3]. Informal statements should be enriched, structured and formalized progressively. Each phase of the construction progress is carried out according to the needs and constraints of the following phases. Results achieved at each phase can be reused by the next one, which optimizes the process and assures good traceability across the phases.

Standard development life cycle and the formal development life cycle (Figure 2.1) are apparently similar in their form. However, formal development life cycle is shorter in the number of steps. The first phases for both the life cycles are the system analysis and the physical modeling. In this phases, informal requirements are taken into account. The capabilities of the software to be developed and the constraints are described in this phase. In both life cycles the procedures are the same. Although, it is recommend that for the formal software development cycle some requirements should be written in a more formal way. Applying this technique, it becomes more easy to express, in the next phases, the requirements to the formal specification [23]. Going

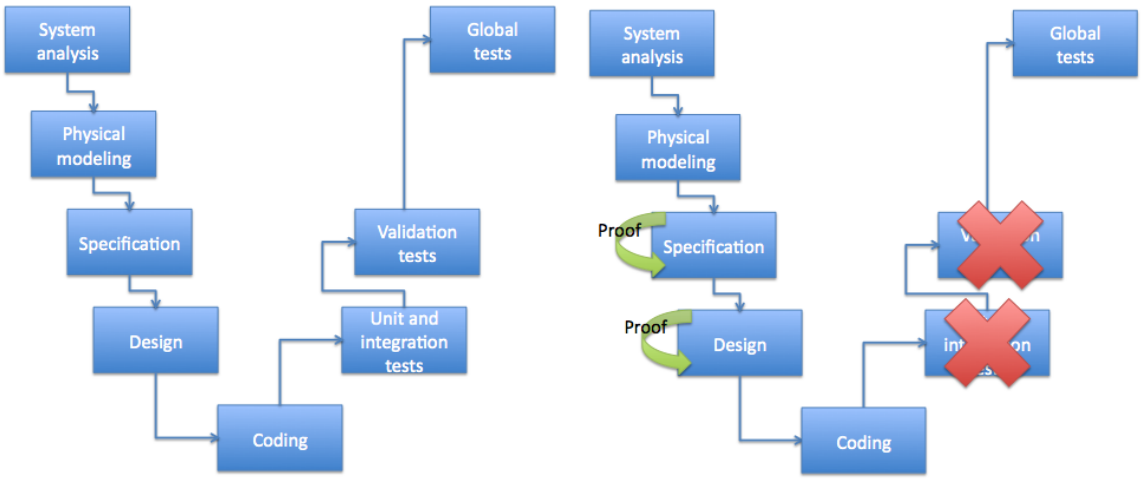

Figure 2.1: Standard development life cycle and formal development life cycle

a little deeper, we find the next phase, the software specification. Here are defined abstractly

which functions the software must perform in order to have the desired capabilities.The way of dealing in this phase is completely different in the different life cycles. In the formal development cycle, the analysis of the problem lead us to define an abstract solution that meet the requirements. The solution is organized from the most abstract to the most detailed and concrete, using a top-down approach. The result is a collection of formal components organized in an architecture describing the dependencies and a decomposition of functions and data. Using this methodology, functions are transformed into operations in the specification language. Data is transformed into variables and operation parameters. Basically, the properties collected from the requirements are decomposed in several parts over the specification. The invariant plays an important role in this process. Properties that are pretend to hold through the all development are specified in the invariant. So, safety and functional requirements are distributed across a set of invariants. In the standard process, it is frequent to found models in UML to help in the specification of the problem. UML can be very helpful to understand and achieve the solution for the problem.[35]

After the specification phase, the objective is the production of an eventual executable program from an implementation programming language. In the standard development process, some code obtained from the UML models is re-utilized. The programmer respects the models generated in UML and implement a solution for the desired functionality. In the formal life cycle, the executable code of the software is obtained by translating, either automatically or manually, the models.

For the testing phase, in the standard development cycle, it can be divided in three sub-phases. It will be described very briefly each one:

- The unit test phase, that checks each procedure of each sub-program;
- The integration tests phase, that checks the cooperation of sub-programs;
- The validation tests phase, that checks the adequacy of the program with respect to the requirements.

In the formal development cycle, tests are also divided into three sub-phases. Unitary tests concern exclusively operations of abstract machines not formally refined and implemented and operations that could not be proved completely. Integration tests concern the integrity of formally and not formally developed modules. As in conventional developments the functional tests are performed normally.

One particular aspect is the substitution of unit, integration tests and validation tests by proofs. Validation proofs is carried out at preliminary and detailed design phases by the validation team. Validation may be based on a number of techniques, the two main ones being formal proof and model checking. When proving a system (that it follows a given behavior), the used proof tool can do it automatically or interactively. In both cases a *formal proof* is constructed. Tools dedicated to automatic proof construction are called *theorem provers*. Interactive proof systems are usually referred as *proof assistants*. Proof tools can be used a diferent stages of the development cycle, but their common usage lies in the validation and design stage where they are used o prove that no erroneous behavior can occur.

2.3 B Method language

At present various formal methods exist and new ones are likely to appear. What is the best formal method is a question that in the opinion of the author depends on the purpose and utility of the system. Essentially two approaches have been developed; one is to express the behavior of a program or a system in an abstract way, a model, and then proof mathematically the consistency of the system. The second one is to attempt to prove properties using an already existing code as input. The B Method is based on the first approach.

B is a tool-supported formal method based on Abstract Machine Notation (AMN), used in the development of computer software. It was originally developed by Jean-Raymond Abrial in France and the UK. The B method due to J.R Abrial is a formal method for the incremental development of specifications and their refinements down to an implementation. The method of software development based on B is known as the B Method. It is a model-based approach similar to Z and VDM. At each step of the B development some proof obligations are generated, enabling the verification of refinement as well that of abstract machine consistency.

AMN provides structuring mechanism which support modularity and abstraction in an object-based style, making provable correctness a realistic and achievable goal through system development. The method is based around the concept of layered development, which constructs larger components from collections of smaller ones.

The B notation is strongly marked by simplicity forcing the user to use well-understood program statements, so only the simplest programming statements are included within B. Structuring, management and control of large volumes of detail in large software systems is a difficult task.To verify their combination and their relationships is very hard without the notion of refinement. The structuring mechanisms provided by the B method are also characterized by simplicity, and are designed particularly with verification in mind.

Developing based in invariant assertions provides consistency conditions between components. These invariants hold the document together and give rise to proof obligations which can be used to guarantee its correctness.

The B language is an extensive language, rich in details and full of proper constructions. The aim of these following sections is not to provide all the details but to give an overview of the different aspects in the language.

2.3.1 Abstract machines

The basic mechanism in the B Method is the abstract machine, it defines in different clauses, data and its properties as well as operations. They represent the base definition of what the program components will do, i.e. the minimal functionality that satisfies the program requirements. Various known programing notions, like modules, classes or abstract data types are concepts very close to abstract machines. A machine contain variables and operations. Variables are encapsulated by the machine and operations enable the access and manipulation of machine variables.

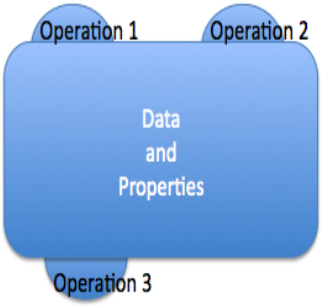

Figure 2.2: Abstract machine representation

Mathematical concepts such as sets, relations, functions and sequences are the elements that describe the basic machines. One central concept in abstract machines are the static laws. This concept, defined by predicates, constitute the invariant of the abstract machine. Variables when changed by operations must always obey to the invariant. Operations behavior is specified

formally using generalized substitutions to change predicates. A given operation can have a pre-condition: it is a predicate expressing the conditions necessary to invoke the operation. Operations can also contain an action: it is a substitution describing how the variables of a machine are manipulated.

Abstract machines belong to the specification phase. Sequencing and loop substitutions are forbidden in this phase. So, the supposed is to describe what an operation should do, not how. A good way of dealing with this limitation is thinking in terms of nondeterminism because it leaves open choices for further developments. Parallel substitutions are used instead of sequencing, order for the application of this type of substitution is no applicable. A mechanism for the verification of the invariant is necessary for guarantee that the substitutions applied by the operations preserve the invariant. Proof obligations are constructed from the formal definition of the substitutions in the abstract machine. If proof obligations are proved then the invariant is always true. This mechanism ensures the correct behavior of the machine ensuring that the predicates defined on the invariant always remain true.

2.3.2 Refinements and Implementations

Refinement is a technique used to transform the "abstract" model of the software system (its specification) into another mathematical model that is more "concrete" (its refinement)[3]. Refinement can be performed in three different ways: the removal of the non-executable elements of the pseudo-code (pre-condicions and choice), the introduction of classical control structures of programming (sequencing and loop), and the transformation of mathematical data structures (sets, relations, functions, sequences and trees) into another structures that might be programmable (simple variables, files, or arrays). So, refinement is basically applied to the operations and data of the abstract machine. The refinement can be applied using various steps, this is for guarantee the careful control of the transformations. During each step, the initial abstract machine is entirely reconstructed. It keeps, however, the same operations, as viewed by its users, although the corresponding pseudo-code is certainly modified. In the intermediate refinement steps, we have a hybrid construct, which is not a mathematical model anymore, but certainly not yet a programming module.

The final step in the refinement is the implementation. An implementation is a ultimate level of refinement of an abstract machine. It is written using a B language sub-set called B0 language, which has the following characteristics. The implementation data must be concrete data (scalars, arrays) that are directly implemented in a high level language like C. Also the body of the implementation is made up of concrete and sequential substitutions, called instructions, that are directly executable in a high level programming language.

During each refinement, operations refinements need to be proven compatible with the operation that refine. The proof of refinement guarantees that the code of the refinement will conform to its specification in the abstract machine. A picture showing the refinement mechanism is presented in Figure 2.3.

2.3.3 B Architecture

A complete development in B corresponds to a B project. A project enables formally modeling a system of any type. Mechanisms of decomposition and composition of abstract machines are used to accomplish a project. As soon as the level of complexity of the refinement of the abstract machines reaches a point where is too high, it is recommended to decompose it into several, more simple parts. The implementation can then be implemented on the specification of one or more several abstract machines, which are themselves refinable. This is done using the IMPORTS clause and calling the operations of the imported machines. Using this methodology, it is possible to construct a B project gradually, according to an architecture made from layers of

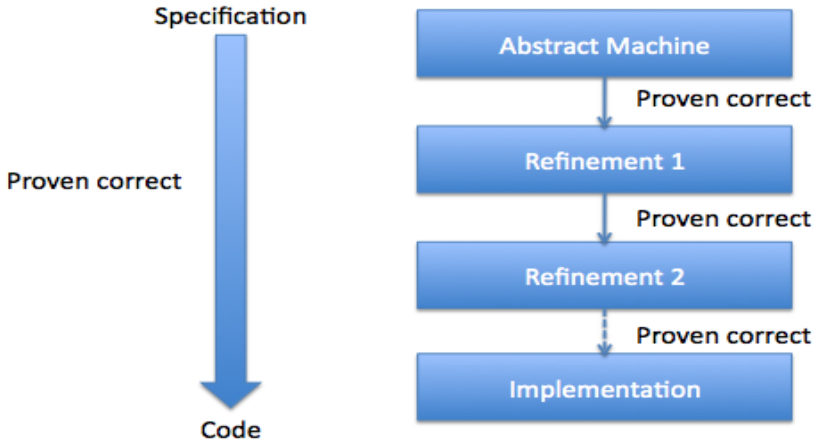

Figure 2.3: Refinement mechanism

abstraction. Figure 2.4 illustrates a possible example of a B project. It is possible to see in the

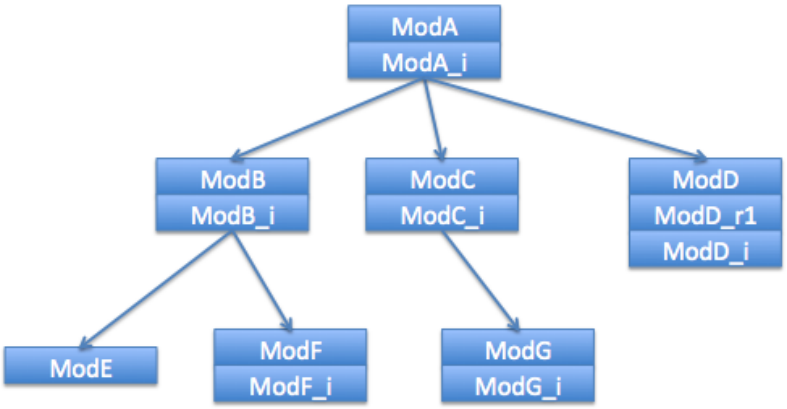

Figure 2.4: A B project

picture that the project is composed by several modules. The arrows represents IMPORT links. The modules are made up of B components. A module has the following properties: it always has an abstract machine, representing the module specification. It may have an implementation (components finishing with "_imp") and possibly some refinements (components finishing with "_r").

The formal development using the B Method fills exactly in the Balzer Software fife cycle 2.5. A formal specification (in which a mathematical text – the machine – is written prescribing "what" the intended software system should do) is linked to (one or more) implementation. The intended coherence between these two phases is achieved by means of a justification, a mathematical document saying "why" the implementation meets the abstract model. This can be done gradually using a technique called refinement.

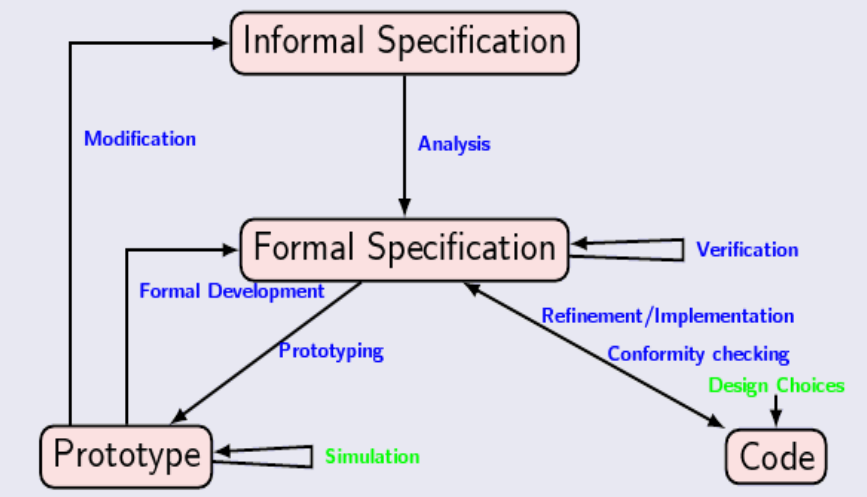

Figure 2.5: Diagram of Balzer's software life cycle

In essence, the life mentioned life cycle is composed by:

- **Informal Specification** – The costumer exposes the problem, which is textually defined via informal requirements;
- **Formal Specification** – The development team builds a mathematical mode of the requirements (several machines in the B Method). This procedure describes the formal understanding of the problem by the team;
- Prototype Machines are animated to see if they correspond to the expected functional behavior. The customer can provide also some input to see if the model correspond to the expectations;
- **Formal Specification** – If the model does exactly what is expected, then, it is refined and formally verified using proof obligations for check the internal consistency. The B Method uses internal mechanism to verify the correctness;
- **Code** – After reach an implementation level, the code is automatically generated using the appropriated B tools.

As can be seen by the previous stages, the B Method fulfills all the stages of the Balzer life cycle.

2.3.4 The B language

Basically the B language is a composition of four elements:

- Components (abstract machines, refinements, implementations)
- Predicates
- Expressions
- Substitutions

All of them will be described in the following text.

Components

Like mention before, components can be an abstract machine, a refinement or an implementation. Components are composed by a set of clauses describing the static and dynamic properties of the behavior. The main clauses are:

- MACHINE – declaration of the machine name and parameters if them exists;
- REFINEMENT – declaration of the name of a refinement;
- IMPLEMENTATION – declaration of the name of an implementation;
- REFINES – clause used to declare the name of the refined component;
- IMPORTS – when a implementation imports an abstract machine, it can use the imported machine operations. However it can not use the imported machine data. This is one of the main principles to achieve decomposition;
- SEES – when a component sees another, it can consult its data and use operations in which do not modify these data;
- INCLUDES – when an abstract machine includes another abstract machine it integrates the data of the included machine;
- PROMOTES – when a component promotes another it is possible to promote operations belonging to machine instances created by the component, this means that the operations promoted from the imported machine pass to belong to the promoting machine;
- EXTENDS – special case of the promotes, in which all operations of the imported machine are promoted;
- DEFINITIONS – declaration of purely syntactic translation. These textual definitions will be expanded in the component before any further analysis. This feature is similar to $\sharp define$ in C and C++, but one definition cannot use another one within the same machine;
- CONSTRAINTS – declaration of properties of the machine's parameters;
- SETS – declaration of abstract and enumerated sets;
- CONCRETE_CONSTANTS – declaration of constants, concrete and implementable, which will be kept during successive refinements;
- ABSTRACT_CONSTANTS – declaration of abstract constants, which are non-implementable and must therefore be refined. Constants may be members of sets, sets, or functions;
- PROPERTIES – declaration of the constants' properties;
- CONCRETE_VARIABLES – declaration of concrete variables, which are implementable and will be kept during the successive refinements. The list of variables is separated by commas. What these variables represent, their types, properties and relationship between them are contained in the invariant clause;
- ABSTRACT_VARIABLES – declaration of abstract variables, which are non-implementable and must therefor be refined;
- INVARIANT – declaration of invariant properties of the variables. This is the logical assertion about the system which shall always hold ("a safety constraint"), but, more importantly, defines what the machine really is. Within it, all variables types, properties, and relationships are defined. It is the most difficult part of the specification to produce, since the invariant must be strong enough to hold information about the system without contradict itself;
- ASSERTIONS – declaration of the "facts" that derive from the invariant. They are mostly used to make proof easier;

- INITIALISATION – initialization of variables. When the "machine" starts the variables initialized in this clause need to stay true and do not break the invariant;
- OPERATIONS – declaration of the operations in the form of a header and a body. This represents the actions that can be performed with the variables.

Predicates

Predicates are a subset of the logic of first logic predicates. They are formulas that can be proved or disproved or that may be part of the assumptions used to determine the proof. Predicates are used to express properties for the components data. An important utilization is in the expression of the invariant clause, in which the predicates express the properties of the variables. Another kind of utilization is in the pre-conditions under which operation can be called (these conditions relate amongst others to input parameters of the operation).

Predicates are also used to express Proof Obligations. Hypotheses under which the proof is made and that can help in the proof or refutation of the goal are also presented in predicates.

The predicates of the B language are grouped in the following families:

- simple propositions (conjunctions, negation, disjunction, implication, equivalence),
- quantified predicates (universal, existential),
- comparison predicates between predicates.

Expressions

Expressions are formulas describing data. Every datum has a type and a value. The categories of expressions are:

- basic expressions;
- boolean expressions;
- arithmetic expressions;
- expression of couples
- set expressions (empty set, set of integers, boolean, ...);
- set construction (set of sub-sets, union, intersection, ...);
- relation expressions (identity, inverse, projection, composition, iteration, domain, ...);
- function expressions (injections, surjections, bijections, ...);
- function constructions (constant functions, lambda expressions, ...).

Substitutions

Generalized substitutions are mathematical notations defined as predicate transformers. They are used to describe the dynamic behavior of B components, this is, their operations.

The description of the behavior with generalized substitutions is used for abstract machines, refinements and implementations. The specification of the substitutions can be non-deterministic and non-executable if we are in the abstract, refinement level of abstraction. Whereas in the implementation level, substitutions correspond to instructions of a classic computing language.

Generalized substitutions are also used for proof obligations construction. This process is automatically achieved using the B components. For example, the proof obligation corresponding to the presentation of the invariant during the call of an operation is constructed by taking the invariant as hypothesis and the proof of substitution of the operation applied to the invariant as the goal. The substitutions of the B language are:

- **substitution becomes equal, substitution becomes such as, substitution becomes element of:**

$$x := E;$$

- **pre-condition substitution, to express pre-conditions of operation calls:**

PRE G **THEN** S **END**;

- **bounded choice substitution,** $SELECT$ **substitution:**

SELECT G **THEN** S **END**;

- **substitutions** ANY **and** LET **which introduce data verifying certain properties:**

ANY v **WHERE** P **THEN** S **END**,
LET v **BE** P **IN** S **END**;

- VAR **substitution which introduces local variables:**

VAR v **IN** S **END**;

- **conditional** IF **and** $CASE$ **substitutions:**

IF P **THEN** S **ELSE** Q **END**,
CASE E **OF**
EITHER l **THEN** S
OR m **THEN** T
...
OR n **THEN** U **END**
END

- **simultaneous substitution:**

$x, ..., y := E, ..., F,$
$x := E||...||y := F;$

- **loop substitution**

2.4 Example

In oder to illustrate the described methodology, a simple example is presented next. First, an informal specification of the problem is given. Then, a B complete development is build, step by step, showing the different aspects of the B language.

The problem was extracted from [3], and the complete solution was presented in [4]. Supposing that an hotel wants to develop a system for control the allocation of their rooms. It is necessary to control the rooms available. The operations require are (i) the reservation of an room, (ii) get the number of available rooms and (iii) free an used room.

The first machine to de specified is the *Reservation* machine. This machine provides the basic operations for the maintainability of the hotel rooms.

MACHINE
Reservation(nb_max)
CONSTRAINTS
nb_max ∈ 1 .. 1000
DEFINITIONS
PLACES ≙ (1 .. nb_max)
VARIABLES
occupied
INVARIANT
occupied ∈ 𝔽(PLACES)
INITIALISATION
occupied := ∅
OPERATIONS

nb ⟵ free_places=
BEGIN
nb := nb_max − card(occupied)
END;

place ⟵ reserve=
PRE
nb_max − card(occupied) ≠ 0
THEN
ANY pp **WHERE**
pp ∈ PLACES − occupied
THEN
place, occupied := pp, occupied ∪ {pp}
END
END;

freePlace(place)=
PRE
place ∈ PLACES ∧
place ∈ occupied
THEN
occupied := occupied − {place}
END

END

Reservation machine receives a formal parameter called *nb_max*, this is the number of rooms available for the hotel. The property that the formal parameter needs to be between 1 and 1000 is declared in the *CONSTRAINTS* clause. The formal parameter is also used in the *DEFINITIONS* clause. Here it is defined the numerated set *PLACES*, that represent the number of available places that the hotel has.

The variable *occupied* tells the places that are occupied and the ones that are not. This variable is typed in the *INVARIANT* clause as a finite set of the type *PLACES*.

The machine is composed by three operations. The first operation is the *free_places*, that returns the number of free places. It is quite simple to understand how it works, if we subtract the number of places actually used in the hotel, represented by *occupied*, to the total number of places, we obtain the number of free places. The next operation is *reserve*. In this operation a pre-condition is presented. It is necessary the hotel to have free rooms to make a reservation.

In the body of this operation is used non-determinism to chose one available room. Selecting a place that belong to *PLACES* but do not belong to *occupied* we obtain a free room. Using non-determinism it is possible to think in what we want to do and not in how. This can be very helpful in the specification phase.

The final operation is *freePlace*. This operation receives an occupied room and free the room for further use. The parameter of the operation needs to be a room belonging to the occupied rooms.

This machine is far away of being an implementable machine. We are using sets, and this type of structure is not implementable in common languages. We are dealing with the specification of the problem. Defining in the abstract machine the properties of our system.

Since we have already defined our properties, we can step down a little bit and try to refine the machine. The firs refinement consists in substituting the free variable representing the number of free rooms by one variable (*nb_free*), the second replacement is change the variable occupied for one function characteristic (*state*) which associates the places with a boolean.

$$\text{state}(\text{tt}) = \text{TRUE} \Leftrightarrow \text{pp} \in \text{occupied}$$

Applying this property to the invariant we obtain the following formula:

$$\text{occupied} = \text{state}^{-1}[\{\text{TRUE}\}]$$

The modifications of the operations are simple reformulations which try to complete the changes in the refined variables.

REFINEMENT
Reservation_r1(nb_max)
REFINES
Reservation
DEFINITIONS
PLACES $\widehat{=}$ (1 .. nb_max)
VARIABLES
state
CONCRETE_VARIABLES
nb_free
INVARIANT
state $\in$ PLACES $\rightarrow \mathbb{B} \wedge$
nb_free $\in$ 0 .. nb_max $\wedge$
occupied = state^{-1}[{TRUE}] $\wedge$
nb_free = nb_max $-$ card(occupied)
INITIALISATION
state := PLACES $\times$ {FALSE} $\|$
nb_free := nb_max

OPERATIONS
nb $\longleftarrow$ free_places =
BEGIN
nb := nb_free
END;

place $\longleftarrow$ reserve =
BEGIN

```
        ANY pp1
        WHERE
            pp1 ∈ state^−1[{FALSE}]
        THEN
            place := pp1 ||
            state(pp1) := TRUE ||
            nb_free := nb_free −1
        END
    END ;

    freePlace ( place ) =
    BEGIN
        state(place) := FALSE ||
        nb_free := nb_free + 1
    END
END
```

The final step is to reach implementation. The variable *state* (total function) can be implemented importing the machine from the basic library provided from B, *BASIC_ARRAY_VAR*. This machine accepts two parameters (domain and range) to form a relation. In our case, the domain is the set with the possible values for the number of places ($1..nb\_max$) and the range is the set of BOOL. Basically, this machine encapsulates the relation *state* from the previous refinement using an array. The imported machine provides two operations, *STR_ARRAY* to set a value in the relation and *VAL_ARRAY* to get a value from the relation. To respect the previous refine machine, it is necessary to initialize with *FALSE*, all the values in the range of the relation of the imported machine.

The invariant clause in the *Reservation_i* implementation, connects the invariant of the refined machine and the invariant of the imported machine. They both are total functions.

```
IMPLEMENTATION
    Reservation_i(nb_max)
REFINES
    Reservation_r1
IMPORTS
    BASIC_ARRAY_VAR(1 .. nb_max, 𝔹)
INVARIANT
    state = arr_vrb
INITIALISATION
    VAR ind IN
        nb_free := nb_max ;
        ind := 1;
        WHILE ind ≤ nb_max DO
            STR_ARRAY(ind,FALSE);
            ind := ind + 1
        INVARIANT
            ind ∈ 1 .. nb_max + 1 ∧
            (1 < ind ⇒ arr_vrb[1 .. ind −1] = {FALSE})
        VARIANT
            nb_max + 1 − ind
        END
    END
```

```
OPERATIONS
   nb ⟵ free_places =
   BEGIN
       nb := nb_free
   END;

   place ⟵ reserve =
   BEGIN
       VAR ind,bb IN
           ind := 1;
           bb ⟵ VAL_ARRAY(ind);
           WHILE bb = TRUE DO
               ASSERT ind < nb_max THEN
                   ind := ind + 1;
                   bb ⟵ VAL_ARRAY(ind)
               END
           INVARIANT
               ind ∈ 1 .. nb_max ∧ bb = arr_vrb(ind) ∧
               FALSE ∈ arr_vrb[ind .. nb_max]
           VARIANT
               nb_max − ind
           END;
           STR_ARRAY(ind,TRUE);
           nb_free := nb_free −1;
           place := ind
        END
   END;

   freePlace ( place ) =
   BEGIN
       STR_ARRAY(place,FALSE);
       nb_free := nb_free + 1
   END
END
```

This small example is only intended to show the previous presented concepts. To be a complete example it is necessary to have another machine working like a "main" machine, providing the operations offered by the *Reservation* machine. Another important aspect, not presented, is the discharging of the proof obligations. It is necessary to proof at each step that the refinement or implementation respects the refined machine.

2.5 B tools

Tools play an important role in the development and validation of software. The B Method has a large variety of tools that can help in the different phases of the software development. A perspective of the most commonly used is present here.

2.5.1 ProB

ProB is an animator and model checker tool for the B-Method [27]. Due to the fact that is an animator, the user can see the behavior of the models and interact with it. This is an

important feature because it gives some confidence of what is modeled. On the other hand, model checking, the other feature of ProB, exhaustively explores the model.

To perform animation and exploration of the B machines in ProB, it is necessary to restrict the given sets of the machine into restricted small numeric ranges. Doing this it becomes possible to determine the enabled operations and allows the checking of all the reachable states of the machine with finite types. Manually exploring the B machine many problems can be found, such as invariant violations, deadlocks (states where no operation is applicable) or other unexpected behavior not described in the invariant. Model checking will also notify when all states have been explored; this situation guarantees the absence of errors in that space (restricted space defined by the size of the sets). When no limitation is performed, the exploration of the state space will happen until it finds an error or runs out of memory. The algorithm used by ProB is a mixed from depth-first breath-first strategy. A random factor decides whether a given node will be treated in depth-first or breath-first. Typically two types of errors are found when exploring a first machine:

- Systematic errors inside an operation that occurs in most states. Here is not important to locate a particular state, just to systematically try out all operations for all arguments;
- Error when the animation is performed long enough (e.g. deadlock errors). Here it is often not important which particular path is taken, just that the machine is animated long enough.

Typically, depth-first strategy is good picking out errors from the first time, but may fail in the second type. Breath-first strategy works in the reverse way.

Exploring the machine operations it is possible to reach states where the invariant, by some reason, is violated. This is called a counterexample. When a counterexample happens, an alert message and the steps performed to reach that invalid state are showed. This type of consistency checking can be very helpful when used as an complement to interactive proof; the error that result in a counterexample should be eliminated before attempting any iterative proof. Consistency checking detects the following conditions:

- Invariant violation errors;
- Assertion violation errors (assertions are properties of a B Machine that should follow from the invariant);
- Deadlock errors;
- User-specific goal predicate becomes true.

Another useful concept in the use of this tool is for refinement. The B Method requires a gluing invariant to be provided in the refinement. Sometimes the refinement does not hold and it may take a while for the B developer to realize that the proof obligations cannot be proven, resulting in a lot of wasted effort. Whit ProB an automatic refinement checker can be used to locate and understand errors before any formal refinement proof be attempted.

It is important, at this point, to refer the different types of analysis to consistency checking. A consistency checking counterexample is a sequence of operations calls that leads to a violation of an invariant in a single machine. A refinement counterexample is a sequence of operation calls that is allowed in a refined machine but is not allowed in its abstraction.

It is obvious that this tool should be seen as a complement to other tools that permit verification by proof-based. Another important point is that this tool should be seen as a sophisticated debugging and test tool and not a verification tool.

In the last version of ProB, 1.3.0, other features are included. Currently, ProB is included as an additional plug-in to the Atelier B. This was true, sometimes ago, in the Rodin platform. Another additional feature is the possibility to use temporal model checking. ProB provides Linear Temporal Logic (LTL) has an extension to the B language.

2.5.2 Atelier B

Developed by ClearSy, Atelier B, is an industrial tool that allows the operational use of the B Method to develop defect-free proven software. It offers an environment for management of projects in the B language. Main functionalities provided by the tool can be groped in:

- Proof aid. To demonstrate proof obligations using suitable proof tools;
- Development aid. Automatic management of dependences between B components;
- User comfort tools. A graphical representation of projects, display of project status and statistics and project archiving.

Atelier B is a complete framework for the development of complete projects using the B language or Event-B. Type checking is the first phase of the verification of a machine. Syntax analyzer is provided to verify the B files. A grammatical verification is performed and a certain number of contextual verifications including the type control and the control of identifier scopes. Components need to pass to the type checker before passing to the next phases. Automatic generation of proof obligations is the next phase. A component, specified in B, is only correct when all its proof obligations are demonstrated. Two methods for discharge proof obligations are provided. In proof automatic mode, most of the proof obligations are demonstrated without user intervention. The remaining proofs need to be verified using an interactive mode. In this case, the user guides the prover in its proof obligations demonstration using interactive commands (lemma additions, proof by case, etc.). When there are no remaining proofs, i.e. the entire project is proved, the B0 checker enters in action. The function of the B0 checker is carrying out a verification of the specific machine is in the B0 language (a sub-division of the B language), to ensure that a model can be translated to the target language. The version used of Atelier B is version 4.0. This version is free and only permits generation of code to the target language C. On the last phase, the project checker checks all the components of a project to control its architecture (the links between the components). The project must be checked before the final translation of the project.

Additional tools are provided to help in the development phase. For example, BART is an automatic refinement tool. BART permits the refinement and implementation generation using refinement rules expanded by the user. Additional refinement rules can be added for refinement personalization of certain components. To help in the documentation, B models can be saved in pdf, rtf and LaTeX formats.

2.5.3 B-Toolkit

B-Toolkit, the tool provided by B Core, is the concurrent tool for Atelier B in the development using the B Method. The B-Toolkit comprises a suite of fully integrated software tools designed to support a rigorous or formal development of software systems using the B Method. Like in Atelier B, it is provided a set of functions to help in the management of all associated files, ensuring that the entire development, including code and documentation, is always in a consistent state. The main functionalities can be divided into tree sub-groups, a configuration management of the project, a proof-based mechanism for formal verification and a set of tools for help in the documentation. Configuration management checks the dependency between components. A set of software specification and analysis tools, which includes syntax checkers, type checkers and a specification animator. For formal verification it is used a proof-based mechanism. This type of mechanism ensures the generation of automatic proof obligations and automatic and interactive prover. A set of documentation generation tools for automatically producing fully cross-referenced and indexed type-set documents from source files. Gathering all this functionalities it is provided a complete framework to the development of complete projects using the B language.

2.5.4 Rodin

The Rodin Platform is an Eclipse-based IDE for Event-B [32]. It is an open source tool developed for the European Union ICT Project Deploy. Event-B language is an evolution of B Method developed by Jean-Raymond Abrial. Rodin, like the other tools previously described, offers a complete environment to the development of projects using Event-B. It supports configuration management for saving and editing the components of the projects and verification of dependencies between them. It is composed by a type checker and a syntax checker for static verification of the machines. This tool is also a proof-based tool. So, a proof obligation generator for the components is provided. An animator is also provided, AnimB. It allows animation of complete model (all refinements), and can be used to create complex animation with graphical interface.

The difference between Rodin and the previous tools is that its environment is completely customizable. Like other environments that use eclipse IDE, it works like a tool for combine different plug-ins. So, different plug-ins come from different sources. The ones that deserve special consideration are the prover plug-in provided by Atelier B, the animator ProB and the U2B translator (plug-in that generate B machines from UML). Other types of plug-ins are available and worthy to use. One plug-in very important is the one that transforms the code from event-B to "common B". Rodin does not provide any tool for translating from event-B to a target language like C. The way to deal with this limitation is translating the code from event-B to b and then use Atelier B to generate the code. The inverse, the translation between B to event-B is also possible.

2.5.5 U2B

The aim of this tool is to use some features of UML diagrams to make the process of writing formal specifications easier, at least for programmers that use UML frequently [17]. The U2B tool converts adapted forms of UML class diagrams and state chart diagrams automatically into specifications in the B language. This is done adding additional expression, using an adapted form of the B abstract, to the class diagram component and to its attached state charts. The resulting UML model is a precise formal specification but in a form which is friendlier to the average programmer, particularly if they use the same UML notation for their program designs work.

2.5.6 Brama

The Brama model animation tool is more frequently used in industrial world [34]. The objective of this tool is to present the models to customers in a more fashion way. ClearSy has a large experience using B to develop software, and when dealing with customers and in general with those who did not write the model, have difficulty in understanding it on the one hand, and, on the other, have difficulty in affirming that the model represents the system. Brama consists in representing the system with the Flash tools and configuring scripts that allow communication with the Brama animation engine. Using this, the customer gain more confidence in the model and the modeler confirms that is really doing the work in the correct way. The visual power that Brama transmits should be considered very helpful when dealing with customers.

2.5.7 Final Remarks

A relevant subset of tools used in the B development process were presented. The ones used in this project were Atelier B, ProB and Rodin. The main reasons for this choice were some previous experiences in the development of projects in B. Atelier B, is apparently a more

actively used tool than B-Toolkit. The proof of this, is the fact that the last version of B-Toolkit is from 2002 and Atelier B is from 2009.

It seems like the B community is adopting event-B. Recent courses in universities tend to teach event-B instead of B. Due to the fact that this new approach is seen like an evolution of B, it is quite normal that practitioners tend to adopt event-B. One of the main reasons for the evolution of B to event-B was simplicity, like Abrial refers in [2].

Rodin is the ideal development platform for event-B. It offers a large variety of features only in one tool. The environment is easily adaptable to the necessity of the user. It is also possible to use event-B in Atelier B, however, it is not his natural language so Rodin should be preferred. Another reason for this choice is the animator AnimB that is completely dedicated to event-B and Rodin. In the previous versions of Atelier B a common problem was the absence of an animator. This gap was covered adding ProB as a plug-in for Atelier B. With this additional feature the task of modeling becomes easier. Animation of the model gives more confidence to the modeler and can be used as an verification of the correctness of the model before going to the proof, preventing a waste of time trying to prove something that is impossible.

2.6 Summary

This chapter starts with an explanation about aspects related to formal methods, such as modeling. Simple processes like the "parachute paradigm", makes the task of modeling more easy. Then an description of the different phases of the formal development life cycle is presented. The benefits of the use of such process are a precious help in achieving the objective of a good quality product.

This chapter also gave a brief overview of the B Method and the tools related to such formalism. The purpose of this chapter was not to present the whole language but only the needed aspects to understand the presented work. The B Method is not only a language but a complete methodology. The B architecture is composed by several phase.In the first, specification phase, it is used abstract machines. The objective of this phase is to express the desired properties to the system to be developed. In this phase the components are not implementable, we are working with sets, predicates, etc.. The refinement is a very important step because it is how we can pass from a specification to an implementation. Refinement can be accomplished using one or various steps. In each step of the refinement, the properties defined in the specification phase need to continue being true.The final step is the implementation. This final phase can be achieved using decomposition and importing various basic machines.

The example provided is quite simple but illustrates well the B language and methodology. It is possible to see the use of decomposition, the elimination of non-determinism and the use of refinement.

Another part of this chapter was dedicated to the description of a subset of tools related to B Method. These tools are not randomly selected. A study of each tools were more used reveled the previous ones. Atelier B and ProB are the most used by the B community. However, Rodin for event-B gives a complete framework, with large choice of plug-ins. In this thesis only two tools were used: Altelier B for editing, type-checking and discharging proof obligations and ProB, for the animation of the specification and model checking.

Chapter 3

Secure Partition Kernel

> Most early operating systems were implemented by means of large monolithic kernels. Loosely speaking the complete operating system – scheduling, file system, networking, device drivers, memory management, paging, and more – was packed into a single kernel. In contrast, the microkernel approach involves minimizing the kernel and implementing servers outside the kernel. Ideally, the kernel implements only address spaces, interprocess communication (IPC), and basic scheduling. [39]

The concept of microkernel is not an innovation. In fact, a common idea is to use old concepts and re-adequate them to new technologies. Secure partitioning microkernel is one of the new concepts created from the old idea of microkernel. Multi-purpose systems can be accomplished using as base a secure partitioning microkernel. However, secure partitioning can only be achieved through high levels of assurance. Common Criteria is the standard for security evaluation, it defines generic security functional requirements and security assurance requirements.The combination of secure partitioning with formal methods provide the foundations to achieve and assure the security of a system.

This chapter starts with a presentation of the problem (section 3.1). Certification can be achieved through the use of formal methods, in section 3.2 the process of certification is presented. In the next section 3.3, it is described what is a secure partitioning microkernel and the requirements necessary to achieve that implementation. The solution proposed in this thesis is presented in section 3.4. The target microkernel is exolained in detail in section 3.5. In the end a summary of the chapter works like a resume of the main ideas here presented.

3.1 General Overview

Partitioning is being applied for a long time. Several applications, with different purposes, can be deployed in the same system using different critically levels. Hardware solution, with two different boards, was the first solution to achieve the objective of separation. Although and despite the successful use for a long time, it is not as effective in terms of the resulting architecture:

- **Physical constraints**: two boards require more space than a single board, consume more power and more weight;
- **Technology**: increasing computational power allows a single computer to perform more tasks;

- **Testability**: a system running in a single board is predominantly easier to test than a system running in several boards;
- **Reuse**: Segregating a system across different boards implies to replicate common functions on those boards

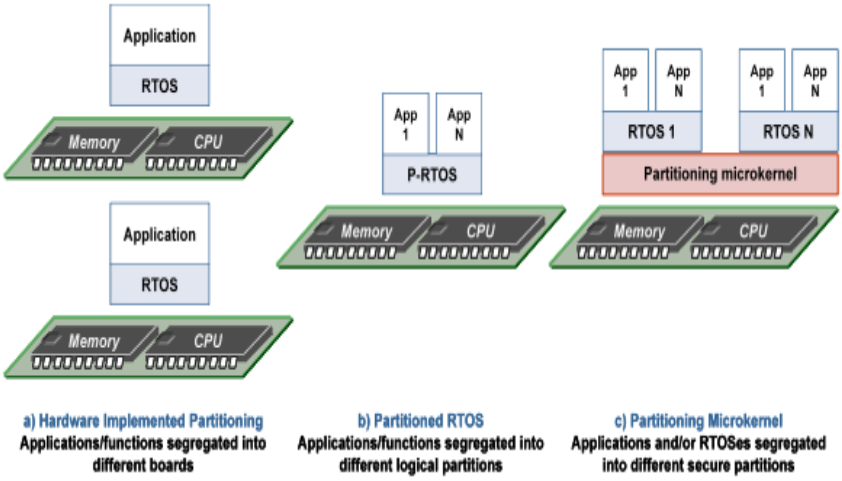

Figure 3.1: Evolution of partitioned systems

As can be seen by the previous picture, an architectural evolution has been applied over the years. In the first times, partitioning was applied as physical segregation of the system. The next step was change from physical segregation to partitions. This is, a real time operating system (RTOS) provides different logical partitions that segregate different applications/functions. The last and pretended solution is the partitioning microkernel. In this architecture several RTOS can be segregated in different secure partitions. It is important to refer that is presented a transition from applications/functions running in a logical partition into different RTOS running separated.

Beside the problem of different critical levels, another important need is isolation of functions developed for strictly different purposes but deployed in the same computer. This is an usual practice, if we are using an email client it should be separated from a browser because they have different purposes. In summary, partitioning may be applied because one or both of the following causes:

- **Criticality segregation**: a given software application encompasses functions of different criticality levels;
- **Multipurpose system**: strictly differentiated applications are deployed in the same platform;

If the previous scenarios are putted working together, different applications with different criticality levels and applications with possible different purposes in the same platform are working together. If those applications need to exchange data among them, for example access to a device driver, them the necessity for secure partitioning arrives again. By merging all the partitioning aspects just presented we obtain:

- **Time Partitioning**: a given time slot is assigned to a given application or task ensuring that this time slot will always be available irrespectively of the computation time required/used by other applications and/or tasks;

- **Space Partitioning**: a given memory block is assigned to a given application or task ensuring that this memory block will always be available irrespectively of the memory required by other applications or tasks. Space partitioning also ensures that other applications or tasks will not access and/or change the data contained in the memory block assigned to a particular partitioning or task - this feature already implements a significant part of the secure partitioning;
- **Secure Partitioning**: communication between multiple communicating processes and/or tasks is performed in a way that ensures that confidential data is protected from unauthorised access. Security requirements may drastically change from system to system and therefore security is typically implemented using a multilevel approach. In functional terms computer security may be described by the well known AAA mnemonic that stands for:
  - **Authentication**: refers to the process of establishing the identity of a given identity;
  - **Authorization**: refers to the process of giving specific privileges to a given identity that has been previously authenticated;
  - **Accounting**: refers to the process of recording the actions of the different entities that use a secure system - this information can later be used for security auditing.

A fourth "A" is typically added to AAA mnemonic and standing that extra A for "Auditing" which is the manual and/or automatic process of handling accounting data.

3.2 Secure Partitioning kernel Protection Profile

Certification is a requirement for systems that will run in "non-safety" environments. Proof/evidence of correctness is a critical issue for critical systems. For separation kernels there is a US made PP entitled "US Government Protection Profile for Separations Kernels in Environments Requiring High Robustness" (also known as SKPP) [19]. The purpose of this document is not to produce a certified kernel but to be used as a reference document with respect to the security aspects.

In terms of security certification the standard commonly used all across several application domains is the "Common Criteria for Information Technology Security Evaluation" (CC). The CC defines seven "Evaluation Assurance Levels" with increasing levels of security assurance requirements (Table 3.2).

Depending on the target for evaluation, CC Protection Profiles (PP) have been defined to group, and adapt as necessary, the CC assurance and functional requirements that are relevant for specific product families such as firewalls, operating systems, etc.. The B Method was used, with success, by GEMPLUS to certify using the Common Criteria the Java Card Virtual Machine [30] [31].

As can be seen by the certification chain presented in the Figure 3.2, all the documents are derived from the CC. The protection profile is a CC subset with the necessary adaptations for a particular purpose. In the figure is also possible to see the Security Target (ST) and the Target of Evaluation (TOE). The objective of the ST is to describe the TOE and should be in conforming to a given Protection Profile (e.g. the SKPP). By the other hand, the TOE is the subject of security certification.

EAL	Definition	Requirements	Functional Specification	High Level Design
EAL1	Functionally tested	Informal	Informal	Informal
EAL2	Structurally tested	Informal	Informal	Informal
EAL3	Methodically tested and checked	Informal	Informal	Informal
EAL4	Methodically designed, tested and reviewed	Informal	Informal	Informal
EAL5	Semi-formally designed and tested	Formal	Semi-formal	Semi-formal
EAL6	Semi-formally verified design and tested	Formal	Formal	Semi-formal
EAL7	Formally verified design and tested	Formal	Formal	Formal

Table 3.1: CC Evaluation Assurance Levels

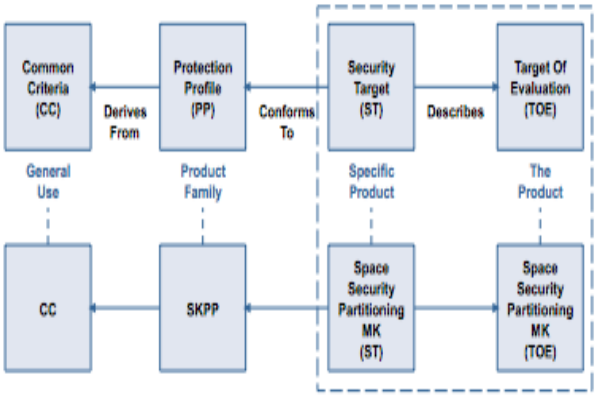

Figure 3.2: Common Criteria certification chain

3.3 Secure Partitioning Microkernel

The goal of this work is to develop an approach to the implementation of a secure partitioning kernel. To achieve this solution it is required to satisfy both safety and security properties. Safety is characterized by the kernel predictability and fault containment. Security is characterized by the kernel nom-bypassability and tamper proofness. Presently some U.S. Secure Partitioning developed kernels are being evaluated and certified. A conclusion that can be achieved from previous approaches is that most of this solutions use an architecture called

Multiple Independent Levels of Security (MILS). Created by J. Rushby [33], it takes advantage of Moore's Law's, performance increases over the last two decades by layering small, formally modeled and mathematically verified components while being affordable. Applications enforce their own security policies instead of relaying on generalized security services. Rushby proposed that memory should be divided into partitions using hardware memory management unit and allow only carefully controlled communications between non-kernel partitions. The implication of this property is that one partition can provide a service to another requiring minimal intervention from the kernel. Due to the fact that a separation kernel only provides very specific functionalities, the security policies that must be enforced at this level are relatively simple. Consequently, the requirements for a separation microkernel can be divided in the following four foundational:

- **Data Isolation** – information in a partition is accessible only by that partition, and private date remains private;
- **Control of Information Flow** – information flow from one partition to another is from an authenticated source to an authenticated recipient. The source of information is authenticated to the recipient, and information goes only where intended;
- **Periods Processing** – the microprocessor and any networking equipment cannot be used as a covert channel to leak information to listening parties;
- **Fault Isolation** – damage is limited by preventing a failure in one partition from cascading to any other partition. Failures are detected, contained, and recovered locally.

As a result of the previous presented properties, the resultant kernel is now much smaller and simpler, and conductive to rigorous inspection and mathematical proof of correctness by techniques such as formal methods. One typical problem in the verification of code is the size of code. Using this approach, the amount of security-critical code is dramatically reduced. MILS is a security foundation that requires that the separation kernel and the trusted components are implemented so that the NEAT security capabilities are guarantied:

- **Non-bypassable** – the security functions cannot be circumvented;
- **Evaluatable** – the security functions are small enough and simple enough to be mathematically verified and evaluated;
- **Always Invoked** – the security functions are invoked each and every time;
- **Tamperproof** – subversive code cannot alter the function of the security functions by exhausting resources, overrunning buffers, or other forms of making the security software fail.

Different criticality levels for isolated software components can be achieved by guarantying safety and security properties. Without isolation, a criticality level is assigned to a complete system function. This is a normal procedure since without partitioning the concern that a fault in less critical software component could have impact on more critical software component is always present. Another advantage, in the use of isolation, is that the verification and certification process will be considerably lightened by having several criticality levels assigned to each software component of a give function. Figure 3.3 shows an example of a MILS architecture. As can be seen by the previous picture, the MILS architecture defines three software layers. The separation kernel (which is the purpose of this work), the application and the middleware. The main responsibilities of the separation kernel are:

- Multi-core time and space multi-threaded partitioning;
- Data isolation
- Inter partition communication
- Periods processing

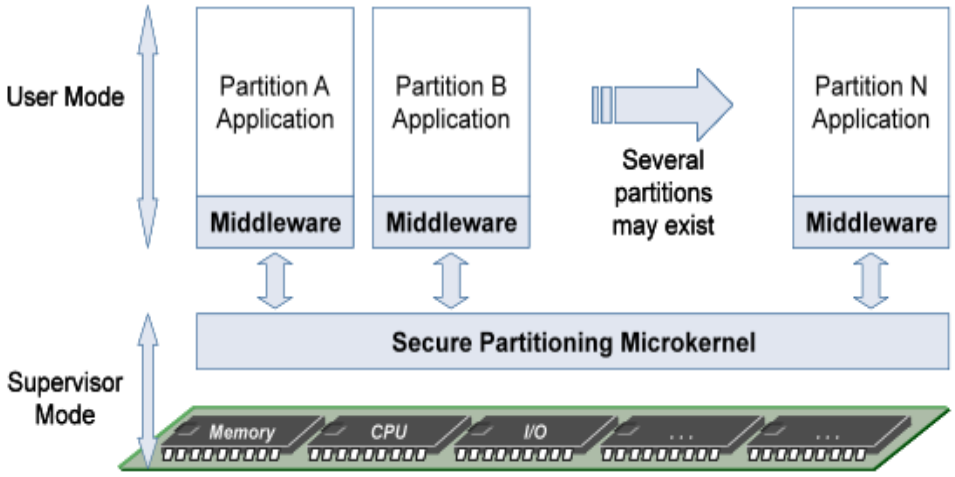

Figure 3.3: Mills architecture

- Minimum interrupt servicing
- Timers

The MILS architecture together with the safety and security requirements provided by the SKPP define a strong foundation to build a microkernel that can provide an environment both safe and secure for applications and systems to run over it. Three main aspects have been addressed to build the secure partitioning microkernel. These aspects are Time partitioning, Space partitioning and Secure partitioning. These top-level functions can be decomposed and combined to provide a full functional block to the microkernel users. Decomposing these top-level functions it is obtained the following set of services:

- **FUN-1 – Time partitioning**
  - FUN-1.1 – Control partitioning execution time
  - FUN-1.2 – Save partition context
  - FUN-1.3 – Restore partition context
  - FUN-1.4 – Provide time partitioning services to the microkernel user
- **FUN-2 – Space partitioning**
  - FUN-2.1 – Manage partition space boundaries (i.e. control MMU)
  - FUN-2.2 – Provide space partitioning services to the microkernel user
- **FUN-3 – Security partitioning**
  - FUN-3.1 – Implement authentication services
  - FUN-3.2 – Implement authorization services
  - FUN-3.3 – Implement accounting services
  - FUN-3.4 – Implement auditing services
  - FUN-3.5 – Ensure integrity of communication data
  - FUN-3.6 – Provide security partitioning services to the microkernel user

In the next subsections an explanation of the top-level functions will be presented.

3.3.1 Time Partitioning

The goal of Time partitioning is to ensure that each configured partition has its own time slot. The total time will be divided between each partition and the kernel. In order to achieve determinism and to increase simplicity at microkernel level, the sequence of scheduling is statically defined by configuration along with the time slot duration for each partition. The sequence of scheduling is cyclic repeated by the microkernel and is named period (T). The criteria for defining the time slot duration depends on the computational weight of the partition and the number of slots in a period depends on the responsiveness needed by the partition. Timer

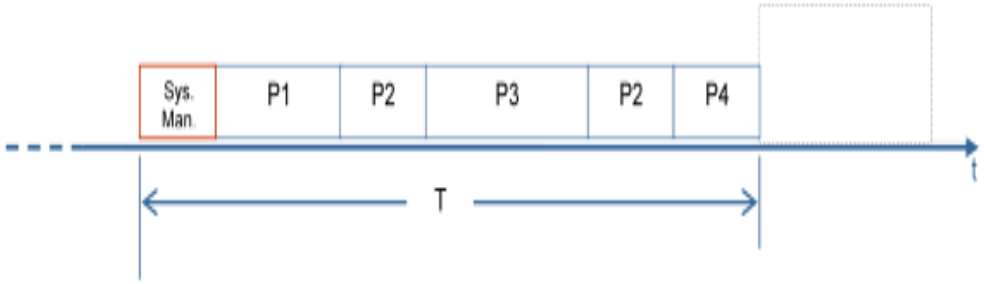

Figure 3.4: Static scheduler period

functionalities should be provided to the partitions. It is important to take some considerations about time dimensioning in the system configuration. The total time for each partition to run should be less that the total period T. The configuration can manage the time slots for each partition (duration and period) taking in consideration the total period T.
As can be seen in the Figure 3.4 it is possible for one partition to run more than once in the total period T. Other aspect that can be important is the period for the kernel partition. Two solutions are valid, in the first one, kernel partition will run between the end of one partition and the start of another. This means that the kernel will run several times during the period T. The second hypotheses is to define a precise period for the kernel to run, for example, in the end of all the partitions stopped running.

3.3.2 Space Partitioning

The main objective of Space partitioning is to provide a well defined virtual address segment isolated from partition to partition. This means that neither one partition will be able to access memory from other partitions, neither other partitions will be able to access memory from this partition. Isolation for each partition is what this property should guarantee.

To achieve this objective the microkernel, in this architecture, will assume that exists an underlying layer that provide these functionalities and will abstract it in basis functions that shall be implemented for every architecture, processor and board. It is mostly common to find an architecture having several privilege running levels, in which the functionalities and resources accessible it the running software are different. The level in which the software can access all functionalities and resources is called supervisor mode. Using this functionality provided by the architecture, it is possible to configure the architecture in a way that every software partition – being it a process, thread or system – will only see the resources that are allocated to it. Another common mechanism provided by the architecture is the memory management unit (MMU). This component have the responsibility for handling accesses to memory requested by the CPU. Its functionalities are translate virtual addresses to physical addresses, memory

protection and others. The translation from virtual to physical address, or form physical to virtual is completely transparent to the partition software. Combining the processor running mode and the memory management unit we have a commonly used protection mechanism to achieve different levels of privileges.

The existing processor units have usually two or more running modes. The available instruction set of the processing unit gets smaller as the processor enters in lower privileged running modes. Usually the instructions that modify configuration registers, namely the running mode itself and the trap table pointer, are only available to the most privilege mode, the supervisor mode. When software running on lower privilege mode tries to execute an instruction that it is not allowed to, the processor will trigger a hardware exception. When an exception happens the processor will automatically halt the current software execution, move into the most privilege mode and will jump into the address specified by the trap table that correspond to the exception occurred. All the addresses in the trap table were previously set by the supervisor software and will point to components of the supervisor software itself. As can be seen in Figure 3.5, the

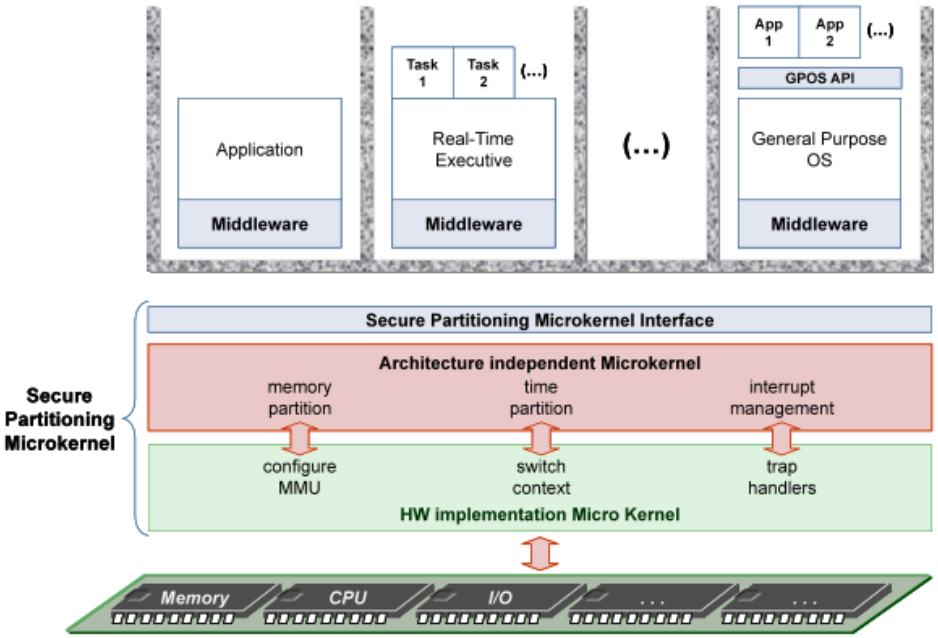

Figure 3.5: Secure partitioning microkernel architecture

microkernel architecture is composed by three layers. The layer in the top is a interface to the kernel, where some basic services are provided to the applications running in the partitions. The layer in the middle is an architecture independent layer. This layer is an abstraction of the hardware, it uses the functionalities provided by the hardware but do not matter by type of the architecture. The last layer is the functions provided by the hardware. In this layer, all the functionalities are extremely depend of the hardware. As can be seen it is where the MMU resides.

3.3.3 Security Partitioning

Secure partitioning works like a complement to space and time partitioning. Guarantying strict isolation between partitions is not simple but if communication between partitions is not

		Partition A			Partition B		
		Subject A	Subject B	Subject C	Suject D	Subject E	Subject F
Partition A	Subject A	–	–	–	Read and Write	Read and Write	Read and Write
	Subject B	–	–	–	Read and Write	Read and Write	Read and Write
	Subject C	–	–	–	Read and Write	Read and Write	Read and Write
Partition B	Subject D	Read	Read	Read	–	–	–
	Subject E	Read	Read	Read	–	–	–
	Subject F	Read	Read	Read	–	–	–

Table 3.2: Partition Abstraction

allowed, isolation can be achieved through the mechanisms of space and time isolation alone. Although it is desirable to provide the possibility of communication between partitions. That is why secure partitioning plays an important role in the secure partitioning microkernel. Several types of security can be enforced, in our case, the type of security is information security. Information flow between partition has to be monitored and controlled by the separation kernel. The SKPP defines the TOE rules for isolation as Partitioned Information Flow Policy (PIFP).

The PIFP is a set of information flows defined at the configuration. Each information flow is identified by the triplet: (1) partition/active entity, (2) partition/resource and (3) mode. The mode can be interpreted as the direction of the information flow.

The PIFP is based on the application of the Principle of Least Privilege (PoLP). The PoLP requires that each component on the system design must have access only to the resources and data that are necessary to its legitimate purpose.

The PIFP as defined in the SKPP is based on the following fundamental principles:

- The scope of the PIFP includes all exported resources; there are no exemptions;
- A controlled operation may result in multiple information flows, in which case, the PIFP must explicitly authorize each flow. Therefore, none of the flows associated with the controlled operation may occur if any one of the multiple information flows is unauthorized. The purpose of this restriction is to reduce the complexity of the TOE;
- The SKPP defines two partition abstractions, each of which represents a different granularity of policy enforcement with respect to information flow.

The two types of partition abstractions are (i) the partition abstraction and (ii) the least privilege abstraction.

Partition Abstraction

In the partition abstraction, all subjects bound to a given partition are enforced by the same restrictions, that is, the flow authorizations assigned to that partition apply equally to all subjects in that partition. For example, if any one subject of a partition requires access to resources in another partition, then all subjects in that partition must have the same access to all of those resources.

An example of a configuration that meets this restriction is shown in table 3.2. As can be seen by the table, all the subjects inside the partition have the same flow authorization. All subjects inside Partition A have the same flow authorization (Read and Write) to Partition B.

Least Privilege Abstraction

In the least privilege (Table 3.3) the flow authorizations is finer grained, active entities may require differentiated rights to access resources in other partitions. The least privilege abstraction requires that both partition-pair and subject resource pair authorizations are used to determine if a flow mode is allowed. This means that, one partition can have an access right to other partition, but the subjects inside one partition can have diferent rights from each other.

The goal of security partitioning is to secure the information and data contained in each partition, in order to this, a strict PIFP should be defined, implemented and followed according

		Partition A			Partition B		
		Subject A	Subject B	Subject C	Suject D	Subject E	Subject F
Partition A	Subject A	–	–	–	Read and Write	Read and Write	Read and Write
	Subject B	–	–	–	Write	Write	Write
	Subject C	–	–	–	Read and Write	Read and Write	Read and Write
Partition B	Subject D	Write	Write	Write	–	–	–
	Subject E	Read	Read	Read	–	–	–
	Subject F	Read	Read	–	–	–	–

Table 3.3: Least Privilege Abstraction

to the SKPP. The SKPP describes the requirements needed in order to implement and PIFP strategy and assure its security. Space and Time partitioning along with PIFP enforcement allow building a strong foundation for a secure application.

3.4 Proposed Solution

As it can be seen by the previous sections [3.1,3.3] it is not an easy task to build a secure partitioning microkernel. In the context of this thesis, the formal development architecture was structured as follow:

Abstract model of the secure partitioning microkernel – Build an abstract model of the secure partition kernel and try to animate him using it. This first model can be seen as the top-level abstract machine that will be further refined.

Generate code only to part of the secure partitioning microkernel – The next step, after build the top-level abstract machine, is to develop part of the secure partitioning microkernel and integrate the developed work with an existing microkernel.

The chosen part to the secure partitioning kernel to be developed into a level of automatic code generation was the PIFP. TO be able to merge the code automatically generated from the B models, it was necessary to found out a microkernel with a set of characteristics:

- **Small complexity** – The microkernel should be small and not too complex. It is important to understand all the details of the microkernel and if it is too complex, more time and effort will be spend investigating the microkernel details;
- **Licensing policy** – The code should be available for us to modify, so, a license like open-source will be extremely necessary;
- **Memory management** – The microkernel should have a memory management system compatible with space separation required for a separation microkernel - i.e. shall have MMU support;
- **Inter-partition communication** – The microkernel should implement some sort of mechanism for communication between schedulable units;
- **Scheduling policy** – The microkernel should provide support for fixed time slots scheduling as to implement the required time partitioning functionalities;
- **HW resources management** – The microkernel should contain interrupt management functionalities allowing associating a given resource to a given partition - i.e. as to apply space partitioning to available hardware resources.

After some time looking over different microkernels (Minix, RTMES, ecos and Fluke), the decision of the chosen kernel fall over Prex. It was the chosen target because of its simplicity and efficiency. The next section is dedicated to describe Prex internal details.

3.5 Prex microkernel

Prex is a real time operating system for embedded systems [1]. It is open source and the main language is C. This microkernel was build to be portable. It has a common layer and an architectural dependent layer, which make him easy to import to another architecture. Another feature in this microkernel is scalability. Kernel objects are not limited inside the kernel, they are dynamically created after system boot. An important feature that every microkernel should have is reliability. Prex was build to prevent crush anytime, even if any invalid parameter is passed via kernel API. The design principle is "garbage in, error out principle". This property guarantees that the kernel never stops even if any malicious program is loaded. Prex code is very clean. It is well commented, and very well structured. It is easy to add or remove parts of the microkernel. Also the debug facilities makes easy to test the microkernel with the desired modifications.

The following figure illustrates the Prex microkernel structure. As can be seen, two layers are well-defined. The first one is a thin interface layer called architecture dependent layer that brings back together all the components dependent of the architecture. The second and bigger layer is the common kernel layer. This layer groups memory components, inter process communication components, synchronization components and the core of the microkernel. The

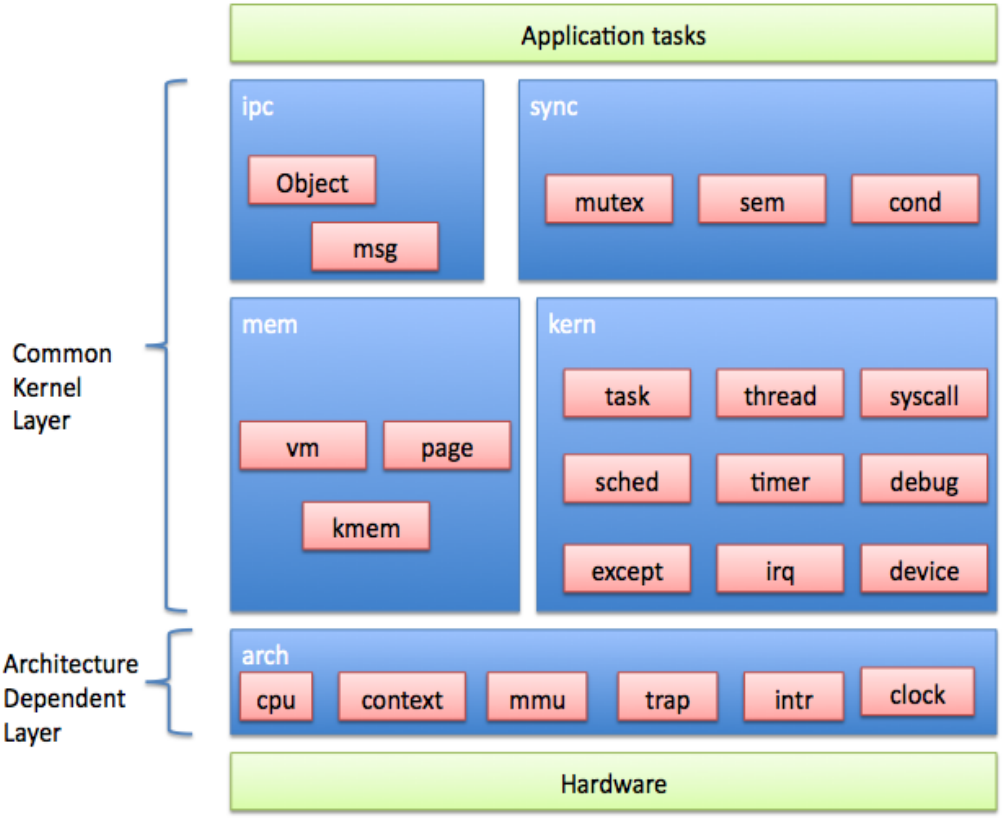

Figure 3.6: Prex microkernel structure

microkernel is structurally well defined. Each component belongs to one of the following groups:

- **kern** – kernel core components;
- **mem** – memory managers;

- **ipc** – inter process communication;
- **sync** – synchronize objects;
- **arch** – architecture dependent components.

We will briefly describe the most important components of each group. The first group is the memory management group (mem). Physical memory is allocated by the basic unit of physical pages. The physical page allocator is responsible for page allocation/deallocation and reservation. Prex microkernel does not swap out any pages to the disk devices. So real-time performance and system simplicity is significantly obtained by these features.

Other important component is the kernel memory allocator. This component is optimized for small memory foot print system. It is the necessary component to allocate kernel memory. Once in a while, when, for example, a thread needs to send data to the microkernel, it uses this type of memory to share data between the microkernel and thread.
The last component inside the group of memory management is the virtual memory manager. A task owns its private virtual address space. All the threads running inside a task share the memory of the task. The microkernel provides allocate/deallocate memory regions for the virtual memory; change memory attributes (read, write and execute); and map another task's memory to the current task. Prex microkernel has an important task, the kernel task. This special task which has the virtual memory mapping for the kernel. All the other user mode tasks will have the same kernel memory image mapped from the kernel task. This feature provides to kernel threads the ability of work with all user mode tasks without context switching the memory map. Figure 3.7 illustrates the microkernel memory mapping between the kernel task and other tasks.

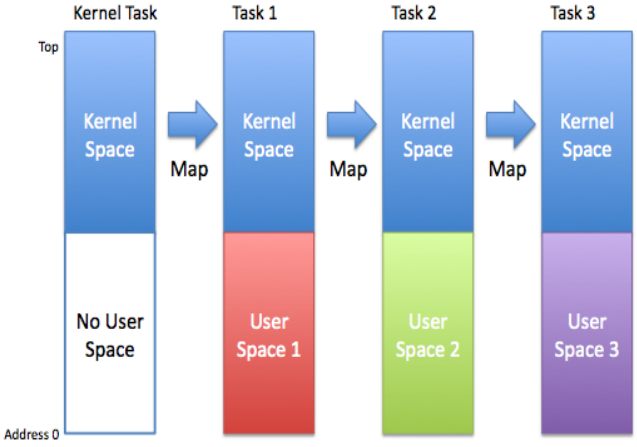

Figure 3.7: Microkernel memory mapping

Like mentioned before, each task has its own virtual address space. In fact, tasks can be seen like a container that holds threads and objects. In Prex, tasks are not the execution unit but the container and memory maps for the units it contains, like threads. Tasks works, in a certain way, like partitions. They both have in common an isolated virtual address space and hold the units of execution. The main difference between a task and a partition is that when a partition is created it is possible to define the size of the partition, something that is impossible to do with a task.

Threads are the minimum execution unit including processor's register state. Each thread has a state, a scheduling policy, a priority and other type of attributes. When a new thread is

created, in the initial state the thread has a owner task, inherited from parents thread and thread state is suspended. So it is necessary start the thread by calling the routine thread_resume(). An important issue in thread creation is the allocation of memory for the thread. Since the Prex microkernel does not allocate any stack buffer for user mode threads, the parent thread has responsible to allocate it. Like exists a kernel task, also exists a kernel thread. This thread is always executed in kernel mode, and it does not have user mode context.

To define which thread will run, Prex has a scheduler. Based on the algorithm known as priority based multi level queue. To each thread it is assigned a priority between 0 and 255, being the lowest priority the one with the higher value, like BSD UNIX. The lowest priority (255) is used for an idle thread. Scheduler based on the priorities of the threads currently in the system, defines the state for each thread. Figure 3.8 shows the different states in which a

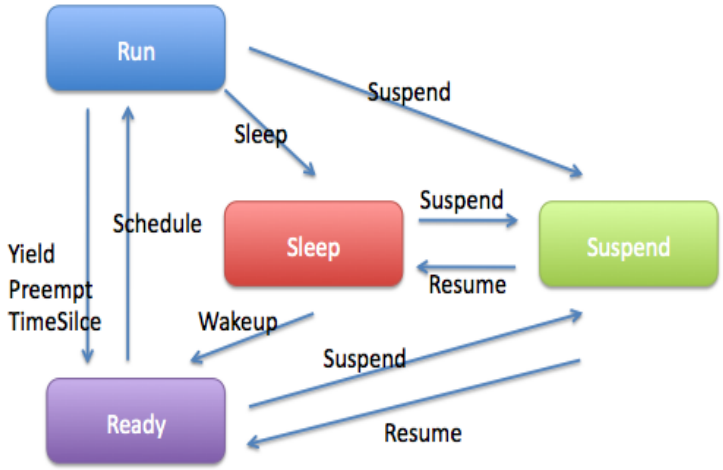

Figure 3.8: Thread states

thread can be. The state run means that the thread is currently running. The threads that are in the state ready are, like the name says, ready to run, those threads are only waiting for their time to run. One thread reaches the state sleep when is pushed for this state by some event. Suspend state happens when the time given to that thread ends and the thread still not finished her work. Finally, the last state is exit, when a thread finished is job.

Communication in Prex is quite simple. Messages are sent to object from thread to thread. The concept of object is similar to the well-known terminology of port in other microkernels. Objects are owned by a task and can be used to hold a queue of messages. Objects can be safety used for communication, each object is stored in kernel space, and are protected form user mode code. Objects are identified, inside the kernel space, by their names. Three basic functions are provided for objects: object_create(), object_lookup() and object_delete(). To send a message to the specified object, the sender must obtain the ID of the target object using object_lookup(). If an object is created without a name, the object is considered private and can be used for threads in the same task to communicate.

Betwen objects messages are changed. Each message needs to include the message header in it. The header is composed by the ID of the sender, the message code and a return status. Although, the header is not filled by the sender task but by the microkernel. This mechanism ensures the receiver task can get the exact task ID of the sender task.

Messages are sent from one thread to a specific object. The transmission mechanism is synchronous. Therefor, the thread which sent the message is blocked until it receives a response from another task. The receiver thread is also blocked until a new message reaches to the target

object. In the end, when the receiver thread receives the message, is also necessary to send a reply message saying that the receiver task had received the message. Figure 3.9 illustrates the all process of communication. It is important to mention that the receiver thread can not

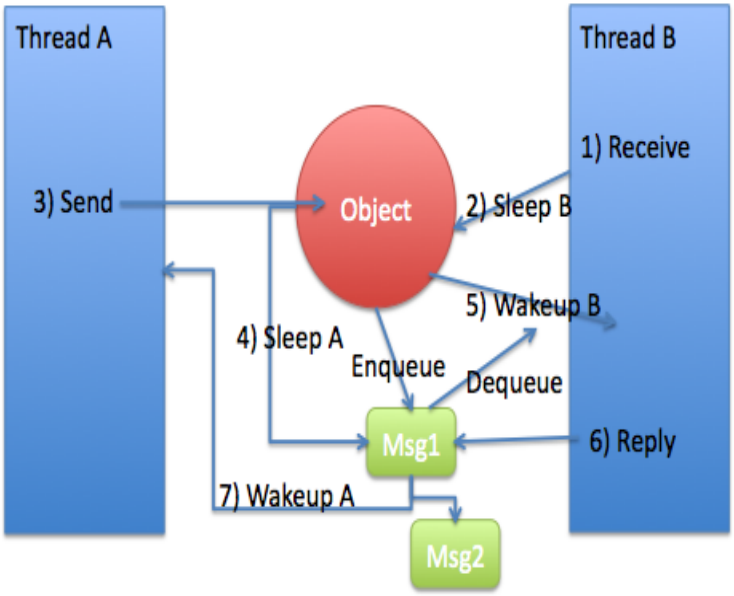

Figure 3.9: Message passing sequence

receive another message until it replies to the sender. This means, that a thread can only receive one message at once. After replying to the message received, it can send another message to different objects.

For the message transfer mechanism (Figure 3.10) the microkernel is used. The memory region of sent message is automatically mapped to the receiver's memory without the kernel. This mechanism allows to reduce the number of copy time while message transfer. Since there is no page out of memory in Prex, we can copy the message data via physical memory at anytime.

Prex is small and not very complex. The source code is available and well commented, additionally, it provides enough documentation to understand the implementation details. The chosen microkernel needs to have a memory management system compatible with space segregation. The microkernel provides this space segregation using tasks. Tasks are isolated in space.

An inter-partition communication mechanism is provided using objects. This mechanism is ideal for communication in a secure partitioning microkernel. PIFP can be implemented in a microkernel using this type of mechanism for communication. The scheduling policy is not the pretended one. However, this type of limitation can be passed if the implementor takes some attention on the way it implements the code. It is responsibility of the implementor to ensure that every task will have some time for scheduling.

All the features that Prex provides lead us to choose this microkernel was a good target for our work.

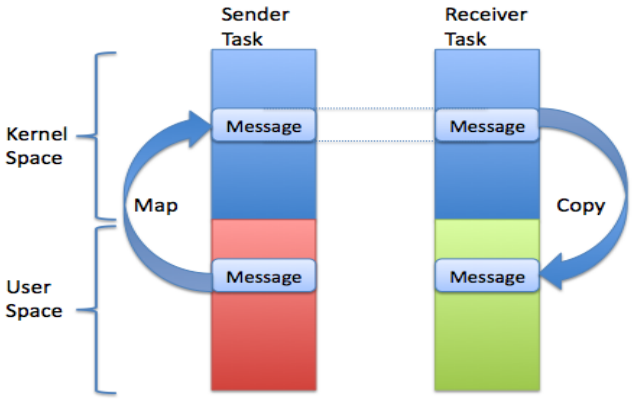

Figure 3.10: Message transfer in Prex

3.6 Summary

In this chapter, a secure partitioning microkernel was introduced. Space partitioning and time partitioning can be accomplished by almost all microkernels. Safety communication is what makes the secure partitioning microkernel different from the others.

The proposed solution introduced in the sequel is an abstract model of a secure partitioning microkernel and a complete development of a flow policy to be integrated in the chosen microkernel (Prex). The abstract model is important for a complete understanding of the functionalities and properties of the secure partition microkernel. After this, a formal development of the flow policy adds new functionalities to Prex.

Prex is a very simple and efficient microkernel. It has good documentation and a good support. Characteristic like the previous ones, and the ones showed in section 3.5 makes him a good candidate to merge the partition information flow policy.

Chapter 4

Verified Kernels: State of the Art

Security in microkernels is an important issue that brings together scientific and industrial areas. Specific areas like the space industry or aeronautics require the use of secure applications for using in high robustness environments. Being the kernel the main component in which those applications run, the security of such component requires even more attention. In this chapter a view of some previous work related to the development of verified microkernels is described. Two types of work are described in the next sections:

- Work related to the verification of microkernels and their properties;
- Verification of microkernels using the B Method.

4.1 Verified Microkernels

The problem in verification of kernel systems is a problem with some years. In [33], John Rushby presents the difficulty that arrive with the design and verification of kernels. It is presented a technique to strength the security in microkernels. This technique consists in building the microkernel as a distributed system in which the security resides in the physical separation of their individual components and the use of some components to perform trusted functions. Placing the security critical software in the kernel, all non kernel software becomes irrelevant to the security of the system. Kernel verification using formal methods as basis for the construction of a secure kernel is provided. A common problem is security of the information. The problem of indirect leakage or direct access to confidential information is also presented. To prevent this problem, the security flow policy should be enforced not only to the non-kernel software but also to the kernel itself. One single system-wide security policy is applied to the all kernel.

The passing of non essential software to the outside of the kernel and the usage of formal verification only crucial components inside the microkernel, together with a separation microkernel can be used to provide a system that can be used in high robustness environments.

Wiliam Bevier in his doctoral thesis [5] developed a verified kernel, Kernel Isolated Tasks (KIT), using Boyer-Moore Logic. The services provided by the kernel are process scheduling, error handling, message passing and an interface to asynchronous devices. The main objective of this work is achieving a correct implementation of the processes and communication between them. The KIT project conceives an attempt to prove task isolation in a kernel written for being used in a very simple Von Neumann machine. Tasks must be able to communicate by some mean.

Therefore, task isolation really means limited task communication. It is defined an address space for a task, and each task has his own pages; shared pages are used for communication.

In KIT verification important security properties are accomplished, mainly, task isolation, protection of the operating system from tasks, and the inability of tasks to enter supervised mode. At the end the system is proved to be tamper proof. However, like other kernel to embedded systems, KIT does not have virtual memory so it only uses memory provided by the hardware and no guaranty in the correct memory usage is provided.

The use of formal methods in the construction and verification of kernels is not a common practice. Classical approaches, for this type of work, are preferred instead of the use of formal methods. The main reason for this is the "misperception gap" that separate formal methods practitioners and OS developers. A work presenting the benefits of using formal methods in OS implementation [41], presents the benefits in using formal methods for appropriated structured kernels. The SPIN model checker is used to analyze the Fluke microkernel's Inter Process Communication (IPC) subsystem. This subsystem was chosen due to its complex implementation and being a system extremely concurrent, making it a worthy target for formal methods. In this paper tools for formal verification of hardware or software are divided in two groups, model checker and theorem prover. The benefits and disadvantages of each one are presented. The main reasons for the choice of model checking and SPIN are that model checking has made major advances in the last years and the learning curve, for newbie's users in formal methods, is shorter when compared to theorem proving. In the end of the project some conclusions are achieved, one is that SPIN or another model checking can be a practical tool for OS implementors. Other conclusion is that if they tried to verify the all kernel, model checking would not have been a good choice. Mainly because enlarging the scale of the software to be validated the problem of state space explosion, common to model checkers, would arrived. One of the main conclusions in this paper, and that really shows the benefits of applying formal methods in kernel validation and verification, is that some bugs were founded and that probably those subtle bugs could evaded traditional testing techniques.

There are more two works in kernel verification using SPIN model checker. The first one is the verification of IPC for the RUBIS kernel [14]. The main objective of this work is to model the already implemented kernel and then, using the model, verify that the kernel has the expected behaviour. Using this work the authors pretend *"to find out a method to formalize and to verify such systems"*. The technique for achieving this objective is to use a model of the system, specified abstractly, and them analyze it. Scenarios and properties are then formalized in the model using Promela, the language of SPIN, and linear temporal logic. The authors have chosen PROMELA as the specification language and SPIN to check the system because according to them *"Communicating finite state machines seemed to be the best choice to model our system"*. Two levels of abstraction are used. In the first level, the abstraction is very high, and some details can be forgotten. For example, in the IPC the type of data is not important to verify some properties like the absence of deadlocks. When the verification is complete, in this level of abstraction, they pass and use more detailed models. Decreasing the level of abstraction, some details of implementation are them taken into account. At the end, the detailed models have to be an exact reflection of the system behavior. Different scenarios for verifying the IPC are also defined and verified using SPIN.

At the end of this work, some important conclusions are achieved. Different kinds of error were detected. Return errors for IPC between tasks were founded. Not returning an error message, the calling task does not know that a problem occurred. This could reach a state where a system crash could happen. Other types of errors concerning memory management were also detected.

The other work using SPIN for verification is [13]. A model of the kernel HARMONY, a portable real-time multitasking multiprocessor kernel, is verified. Like in the previous work the strategy is the same. After modeling, in PROMELA, the kernel and formalizing the properties,

the result using different scenarios is checked using the SPIN tool. To achieve this objective it is used the principle of "divide to conquer". The complete model of the kernel is divided in sub-models. The models taken into account are interrupts, inter task communication, task management and scheduling. The complexity of the models are gradually increased in two different directions. The first direction is the combination of the different sub-models until reaching a complete model. The second direction is the level of abstraction, from more to less abstract. At the end, a tractable model that enables the expression, simulation and verification of any scenario is obtained. The definition of a scenario consists of a bounded number of tasks that may use differet combinations of kernel services. In this paper two sub-models are presented, the inter process communication and task management. Using the models of inter-task communication, task management and running a priority based scheduling it was possible to simulate a quite large number of scenarios. The test of all cases is a complicated issue. However for simple scenarios, they reached some important conclusions. A quite considerable number of errors were founded when executing different scenarios on the model. In the inter task communication primitives a bug was found that did not showed up during ten years of utilization.

Because the system already existed the approach of reengineering was adopted. According to the authors, the usage of reengineering makes difficult the usage of a method of layered refinement. Starting in an abstraction and use consecutive refinements to produce a concrete model was clearly the better approach. Another problem that the authors clearly identify is the choice of the scenarios for verification. It is impossible to cover all the scenarios of utilization, and a problem arrives when deciding the best scenarios for testing. The last conclusions is related to the SPIN tool. It was proven that SPIN was very well suited for addressing this problem because its high verification power. SPIN should be used as a design help and not only in porteriori validations.

Model checking for the verification of kernels is very popular. Another example of this type of verification is the work in [16]. Due to the fact that the kernel is a concurrent system, it becomes appropriate to use techniques to perform a correct design and validation of the system. Using TLA^+ as the specification language and TLC as the model checker, a set of solutions for real time operating systems are proposed. This work focus principally in the resource management mechanism and the protocols to ensure the consistency of the data in those shared resources. Communication mechanism between tasks is also modeled in a simple and efficient way. At the end of the work, the results achieved provide good basis for the development of a "nano-kernel" operating system.

The reason for the choice of a model checking technique instead of other technique is, according to the author, the simplicity that this type of formal method can offer for analyzing concurrent systems. A set of unique characteristics make this type of verification very profitable, namely, the use of branching or linear temporal logic, symbolic or explicit state verification, breadth-first search or depth-first search, real-time or timeless verification, etc..

Using these techniques, is possible to reason about the system abstracting from some details that not influence directly the system. Resource is the key abstraction on this work. Any software structure (for example a set of variables, a memory area, a file, a set of registers, etc.) that can be used by a process to advance its execution is abstracted using resources. This abstraction makes easier to reason about the system behavior leaving details of implementation outside of the scope of analysis. Resources are mutually exclusive to ensure the consistency of data in shared resources. In the first part the model of resource access management is provided. Then to accomplish the objective of the kernel be a real-time kernel, it is presented a priority handling mechanism. To handle time constraints tasks are given priorities and scheduled according to some priority based protocol. Communication ans synchronization between tasks are described in the last part of this work. The showed mechanism can be used in both distributed and non-distributed systems.

The conclusion in this work, using the TLA^+ as the specification language and TLC as the model checker, shows the benefits of using verification techniques, like model checking, to implement systems from the beginning. Normally this type of verification is applied to systems already implemented, using a process of re-engineering for formally achieve a certain degree of confidence in the system. In my point of view I agree with the author of this work, such verification should be performed at the beginning of the development.

Other methodologies, different from model checking also revealed well successful in the verification of kernels. The following text present a description of some of the developed work using techniques different from model checking.

The existence of formal security policies are one of the most important requirements to accomplish a verified kernel according to the US Government Protection Profile for Separation Kernels in Environments Requiring High Robustness (SKPP) [19]. Rockwell Collins and the U.S. Department of Defense presented a work describing a formal security policy for a separation kernel [12]. Separation kernels are normally used as a high-assurance product. This type of products requires precise and unambiguous specifications for high-level certification. A formal security policy is *"a formal specification of what a system allows and guards against"*. In this work, a security policy for a separation kernel for a Multiple Independent Levels of Security (MILS) architecture is presented. Two properties are required for a good specification of the kernel formal security policy:

1. The specification can be proved about a particular system component;
2. The specification can be used in larger system that contains the component about which the specification has been proved.

To prove that the formal security policy proposed in this work is correct, some theorems similar to what others have used to describe a separation kernel are proved in the proposed security policy. It is also presented a case study - a firewall - that uses the separation kernel and shows that the separation kernel security policy implies that the application works properly. ACL2 is the chosen tool for the specification of the formal security policy. The reason for the choice of ACL2 is because it is useful in modelling and reasoning about computing systems.

In this document, are presented a set of previous defined theorems for proving the correct formal security policy. The first property that needs to be proved is *exfiltration*. This is, when a partition is the currently-executing partition, a partition's memory segments can only be effected in a way that is consistent with the communication mechanism. The second property is, supposing that one partition is executing, the effect on a segment cannot depend on anything other than the segment's original value and the values of the current partition. This property is named *meditation*. The final property is *infiltration*. The definition of this property is that, when a partition executes, the values of the current partition's memory segments do not depend on other segments that should not effect it. These tree properties are proved using ACL2 theorem prover for their current formal security policy. At the end, it is formalized a firewall application that uses the previous separation kernel. This is done to show that the firewall works properly using the formal security policy in the separation kernel. To conclude, this work is extremely interesting and important, however the lack of an implementation is an important issue.

In [21], an approach to formulate a formal specification and verification of data separation in a separation kernel for an embedded system is presented to provide evidence for a Common Criteria evaluation. The way to verify the kernel code and the artifacts used to the evaluation are a sequence of five steps:

1. Specification of a Top Level Specification (TLS) of the kernel as a state machine model;
2. Formalization of the data separation property in terms of the inputs, state variables, and transitions defined in the state machine model that underlines the TLS;
3. Translation the TLS and data separation property into the language of a mechanical prover, and prove formally that the TLS satisfies the data separation property;

4. Implementation of the kernel code annotated whit pre- and post-conditions, partition the code into Event, Other, and Trusted Code, where, informally Event code is code corresponding to an event in TLS that touches a Memory Area of Interest, Trusted Code is code that touches a Memory Area of Interest but is not Event Code, and Other Code in neither Event Code nor Trusted Code.
5. Demonstration that the Event Code does not violate separation by constructing:
   a) A mapping from Event Code to the TLS events and from the code states to the states in TLS;
   b) A mapping from pre- and post-conditions of the TLS events to pre- and post-conditions that annotate the corresponding Event code.

   Demonstration separately that Trusted Code and Other Code do not violate data separation.

The first part for the verification of the kernel is the TLS. The major goals of the TLS are to provide an explicit description of the required behavior and to make the assumptions on which the specification is based. To describe the TLS it is necessary to use different stakeholders with different perspectives to communicate and understand each others, so, a formal context and a precise vocabulary is needed. State machine models using precise natural language help achieving a good comprehension of the specification.

A set of properties need to be implemented by the kernel to enforce a security policy to data separation.

- No-Exfiltration – data in one partition cannot influence the data outside the partition;
- No-Infiltration – the data inside a partition cannot be influenced by data outside that partition;
- Temporal separation – the data areas in one partition must be clear when the system is not processing data in that partition;
- Separation of control – when data processing is in progress in one partition, no data is being processed in a different partition, until the processing partition terminates;
- Kernel Integrity – when data processing is in progress in one partition, the data stored in other partition does not change.

To verify that the TLS enforces data separation it is necessary to prove these previous properties with the defined TLS. In this paper this is done using the prover PVS.

Demonstrate the code conformance is necessary. This correspondence is obtained by two mappings. The first one is the mapping between the concrete states in kernel code and the TLS abstract states. The second mapping relates assertions at the abstract TLS level to assertions at the code level. This requires that the code needs to be annotated with pre- and post-conditions. The match between assertions in the TLS and derived code-level assertions is not exact because auxiliary assertions were added to:

- Express the correspondence between variables in the code and physical memory areas;
- Save values in memory areas as values of logical variables;
- Express error conditions that the TLS implicitly assumes to be impossible.

The last part of the verification is the formal foundations. This is necessary to show that the kernel code conforms to the behaviour captured in the TLS.

Some of the conclusions achieved at the end of this project are that an animator is a very helpful tool to reason about the specification. It is said that running the simulator exposed some gaps in the understanding of the kernel. Keeping the size of the TLS specification small helped in very ways, such as proving the data separation, and helped in the discussion between

the different stakeholders. The discussion between the different stakeholders helped to ensure that misunderstandings were avoided and issues solved at the early certification process.

Motorola and the Natinal Security Agency (NSA) worked together in the development of a Mathematical Analyzed Separation Kernel (MASK) [28]. The formal methodology used in the development of MASK was SPECWARE. This environment permits the specification and formal development of software with its primary objective being the correct development of the entire system. More specifically, SPECWARE provides the formal composition of specifications and the refinement of specification into code. The process of specification in this work is divided in three phases. In the first phase, the separation specification, a set of mathematical properties for separation are specified. The second phase is the multiple cell abstraction level specification where it is detailed how the MASK kernel shall meet the separation specification. The last phase is the kernel specification. The purpose of this part is to detail the actual data structures and algorithms used to construct the implementation of the kernel. At the end of the SPECWARE MASK kernel specification, a code implemented in the C language needs to be checked if it is in accordance with the specification. In the end, a complete mathematically verified kernel is obtained using the SPECWARE methodology.

Sometimes it is difficult to gather the different stakeholder (i.e. developers, formal methods practitioners, project manager, etc.) in the development process to work together. The main reason for this is that they have different views of the system. Reconciling the approach taken by kernel developers with the one taken by formal developers practitioners is the scope of [15]. A small kernel is implemented, with the name of seL4, using a complete cycle of developing. Issues like design, specification, implementation and verification of the kernel are presented. The small size of current kernels, and the increase power of iterative theorem proving environments, means that the time is right to attempt formal verification by proof of real world microkernels [40]. The final goal of the project is to achieve a verification of the kernel implementation behavior as it is formally specified in the abstract model and properties such as spatial partitioning of processes hold both for the model and the implementation.

A problem presented in this work is the cooperation between the team of formal methods and the team of kernel developers. To achieve an implementation using formal methodologies is necessary for both teams to work together. The fact that kernel developers tend to adopt a bottom-up approach while formal method practitioners take a top-down approach makes the task more difficult. A need for a methodology that facilitates both teams working together and enables them to efficiently iterate through the design, specification, implementation and verification of the system. The proposed solution is to use an intermediate language that permits both teams to express their ideas, in this case, Haskell. Using this approach, the specification is an implementation so that kernel developers are forced to think about implementation details. On the other hand, the task of the formal verification team is to extract a formal model of the prototype in order to reason about it in a theorem proving environment, Isabelle/HOL. The reason for choosing Haskell language is the precise semantics of the language and the lack of side-effects of functional languages in general. This made easier the translation between Haskell to a formal model that can be used in Isabelle/HOL. The translation between Haskell to a model that can be used in theorem proving environment was mostly syntactic. However, it is important to prove that the model obtained from the Haskell code, really is a simplified, more abstract formal model of the kernel. If the model is really a valid abstraction then is used to facilitate proofs of more complex safety and invariant properties of the kernel without need to go into implementation details.Using this methodology some evidences were found. The verification process found some incorrect behaviours in the model but also in the Haskell code. These problems were captured in an early phase of the process and were easily fixed.

The final step on the previous work is the implementation of the kernel formally verified using a more traditional language such as C. The plan is to use the Haskell code only as an intermediary bridge to unify the formal methods team and the development team. This code

is then passed to a model used in the theorem proving environment, and the final part of the process is the refinement to a C implementation of the formal model. At the end, the produced kernel and its formal proof are sufficient, the Haskell code can be considered redundant.

One of the most important conclusions of this project is that the use of an intermediate language for both teams can be very helpful to cooperatively and iteratively develop a formally verified design and implementation of a small kernel. On the other hand, the translation of the Haskell code to a formal model, performed by the verification team, found a number of problems. However, in this phase much errors can be considered small bugs and easily fixed. Other types of problems were found when the verification team started making the verification proofs in the Isabelle/HOL environment. This shows that formalisation and the use of theorem proving tools is beneficial even if full verification is not yet performed. The final conclusion is that this methodology, with the team working in parallel, enables kernel developers, formal modellers and higher-level programmers to work more closely together, leading to faster results than using a sequential development. It also proves that an intermediate language as Haskell can be very convenient and helpful, making feasible, even easy, for kernel developers and formal methods team to collaborate on the specification, design, implementation and formal verification of a high performance kernel.

Other type of approach is presented in [10]. The Z notation (a predecessor of the B Method) is used to show that the formal specification of kernels is not only possible but also necessary if the operating system is required to achieve high levels of reliability, safety and security. The reliability of the entire operating system, as well as its performance, depend upon having a reliable kernel. In this book, the concepts related to the kernel functionalities are presented in a fashion and comprehensive way. The formal models of three operating systems kernels are presented. Besides that, some important concepts that sometimes are not addressed, like hardware abstraction model, virtual storage and Interrupt Service Routines (ISR) are here presented.

The first model is a simple kernel, like the ones often used in embedded and real-time systems. It is very simple and does not deals with such things as ISR or device drivers. The objective of the author in modeling such kernel is to show that it is possible to produce a formal model of an operating system kernel. Despite being a simple kernel, it should not be underestimated because it is possible to refine the model of the kernel and produce a real working code.

The second kernel presented is an extension of the first one. The model now includes device drivers and in particular a clock process that is central to the process-swapping mechanism. Inter Process Communication (IPC) is implemented using shared memory and semaphores for control synchronization. Other feature is the use of a swapping mechanism to give the kernel the ability for support more processes that can be simultaneously maintained in main store. In relation to the first kernel the complexity is higher. It can be seen as a Tannenbaum's MINIX system [38] except signals, file system and terminal interface. The kernel properties for each part of the kernel are provided and the proofs of the correct behavior are included.

The third kernel is a variation of the previous one. The main difference is the IPC mechanism. The IPC is implemented using a message passing mechanism using ISR. All communication and synchronization in this kernel is based upon synchronous message exchange. Another improvement in relation to the previous one is the modeling of the interface and the respectively system calls. An important proof is presented in this chapter; the proof that only one process can be in the kernel at any time. This property is important to guarantee process isolation in execution, one of the properties that the separation kernel needs to achieve.

In the last part of the book the virtual storage, a particular part of the kernel, is modeled. Recent kernel use virtual memory for system and user processes. Virtual storage offers a considerable number of benefits including automatic storage management at page level, management of large address spaces and support for more processes than will simultaneously fit into main

store without having to resort to the all-nothing techniques exemplified by the swapping mechanisms of the previous kernels. Message passing mechanism is also changed to support virtual memory.

The work previously presented is very helpful when using formal methods to design, verify and implement kernels or parts. However, some details were left to be done in the future. For example, asynchronous signals and kernel initialization. Another missing point is refinement, no refinement is present. Models are simply models and no guarantee that can be implemented is presented.

In response to the fact that in the previous book no refinement is presented, the same author wrote another book completely dedicated to refinement for operating system kernels [11]. The book is about the specification, design and refinement to executable code of two operating systems. Each specification is relatively complete and the refinements reach the level at which executable code in a language such as C or Ada can be extracted from the Z schemata. Proofs for the refinement are presented to show that it is possible to refine and achieve code trough refinement. An advantage in presenting the proofs is that is possible for reason about the specifications and consequently about the refinements.

Two models of kernels are presented. The first model is a small kernel that can be used in embedded systems. The other is a separation kernel as proposed by John Rushby. This is an important reference mainly because it is performed in the Z notation, that is similar to the B Method, and because the separation kernel is the main objective of this thesis.

Already some commercial products of verified kernels are available in the market. The next text describes some commercial versions of verified kernels.

Two companies already have developed their verified kernels. Details about the implementation of each one are not easy to find. Nevertheless, it will be described briefly each one. The first real time operating system is Integrity, provided by Green Hills, is one of the most secure operating system in the world having been certified by the NSA-managed NIAP lab to EAL6+ High Robustness. It is the first operating system achieving this level of certification. The SKPP contains provisions necessary to certify a separation kernel up to EAL 6/7. SKPP requirements include the use of formal methods to mathematically prove the security policies, formal specifications, formal correspondence between design and implementation, complete test coverage of all functional requirements, and penetration testing by the NSA, which has complete access to the source code. Integrity accomplish all this previous requirements to achieve EAL6+. It was designed to be used in systems that require maximum reliability and security. Examples of usage of this kernel are the several aircrafts equipped with this kernel (for example Lockheed Martin's F-16), military computers and other industrial applications

Wind River VxWorks MILS Platform is the second verified kernel. Already achieved the requirements according to SKPP for medium robustness (EAL4+) and currently is being evaluated for achieve high robustness (EAL6+). In a Multiple Independent Levels of Security (MILS) operating system, developers and system integrators define rules for resource allocation (space and time) and information flow. Wind River already had some experience with partitioned environments from the development of Wind River VxWorks 653 Platform for safety-certified Integrated Modular Avionics (IMA) systems. Therefore to create VxWorks MILS some basis of security-certified partitioned systems were already known.

All the previous works showed that it is possible to verify operating system kernels. For some years this was considered an impossible task,in part due to the large amount of code used to implement the kernel. Microkernels make the work of verification feasible.

4.2 The B Method in the Verification of Microkernels

The B Method was successfully applied in several projects. For the verification of microkernels some work was already done. B4L4 project [36] is a consortium that brings together companies like ST Microelectronics and ClearSy, with the objective to achieve a secure operating system using formal modeling and the validation of its development. B-Method is used to model the system and the APIs for the various servers that compose the OS. Studies on the use of these models as test oracles are also conducted. This project already presents some results [22]. The objective is to model the L4 microkernel with formal techniques, in this case with a variation of the B language, event-B. L4 microkernel is second-generation microkernel based on the principles of minimalism, flexibility and efficiency. The size of the kernel and its relevance for security makes it a good candidate for verification. A formal model of the L4 API is presented. The API can be seen as a set of operations that change the internal state of the kernel. Basically, the operations provided by the API invoke a system call in the kernel. The model uses events to simulate system calls, only when the guards of the event are true it is possible to the operation to be executed. The internal data is represented using variables and constants. Mainly two data structures are used to represent the internal state of the kernel: memory; and threads. So, a system call is an event. Guards and system call parameters cause the events to trigger.

To perform to a validation of the model, the technique is to perform some tests over the model and compare them whit the running code. For that, the Brama animator framework is used. The results obtained from each one are them compared and evaluated. This comparison is performed using an interface that allows to convert a kernel state into a mode state, associating the model variables with the implementation data.

The conclusions achieved, at this state of the project, are that the development of a model allows achieving more confidence in the correctness of the API, to describe the behavior from a different point of view and to establish a deeper understanding on how it works. The test framework also permits to validate the model against the implementation. This last feature is a good help when the objective is to validate software already implemented. This project is still ongoing and more results will certainly appear.

Another project that used the B-Method was also the verification of the L4 "Pistachio" API (but this time, using the B language instead of event-B). This work was performed by Rafal Kolanski during his thesis, in the University of New South Wales [24]. The objectives were to produce a formal model of the API and identify potential faults in the API. User threads, privilege threads, scheduling, IPC and other type of internal components that compose the kernel are modeled according to their behavior. Although, due to the fact that the objective is to model the API, the memory management is not detailed, a simple model of which spaces are used by the system and which of those have been initialized is provided. This work was performed using B-Toolkit. This framework offers an animator. However, the animator is quite limited and some dificults arrived from this limitation. According to the author, event-b is the future of B, and any possible attempt of continuing this work should be done using event-b.

4.3 Summary

Some work has already done in the verification of kernels. Mostly of them has used model checking as the primary technique for verification. Mainly two approaches for verification are used. In the first one, the kernel code is already done and most of the work is to see if some bug is found in such code. The second approach is to start all from the beginning and model the whole kernel.

In almost all of the previous works, the memory is not addressed. The main reason for this

is that most of the kernels are embedded systems and memory for this type of kernels is very simple. This thesis memory issues will be addressed.

Other important conclusion of this analysis is that some work involving B in the verification of kernels was already done. Because this is an argument in favor of the choice of the B Method. Both of the cases, in which the B Method was used, the purpose was the verification of the kernel API. The work on this thesis will be different in this aspect.

Chapter 5

Formal Development of a Secure Partitioning Microkernel

This chapter is dedicated to the formal development of the secure partitioning microkernel. The section 5.1 introduces the overall methodology used for the formal development of the microkernel. Next, in section 5.2, a complete abstract model of the secure partitioning microkernel is presented. From this part, it is extracted a small part for a complete development (section 5.3). The objective is to reach the level of automatic generation using Atelier B. In section 5.4 the automatic generated code is integrated with a microkernel. The verification and validation of the work is presented in section 5.5 and in the end a summary works like a conclusion (section 5.6)

5.1 General strategy

Some design decisions were made about the architecture of the secure partitioning microkernel. The chosen part for a complete development using the B methodology was the PIFP. Two types of abstraction can be used for communication. Partition Abstraction or Least Privilege Abstraction. In this work the decision fall over Partition Abstraction. Since in this case all subjects in the same partition are enforced by the same restrictions, a task can be seen like an partition.

Another aspect in terms of design is the scheduler. Tasks are the unit that is scheduled, not threads. The scheduler use a FIFO algorithm. The first task that is added to the system will be the first task running and so on.

Since the microkernel is to be used in embedded systems, no mechanism of swapping is necessary. The only available memory is physical memory (RAM) and virtual memory, provided by the MMU.

All the types and constants are placed in context machines (sufixed “Ctx”). Apart from helpfully dividing the development into state machines and those providing abstract sets, the practice helps with refinement. They can also be used as a method of slowly building up the specification, adding only those context machines that are necessary at every stage. The composition of the machines in the abstract model is the presented in Figure 5.1.

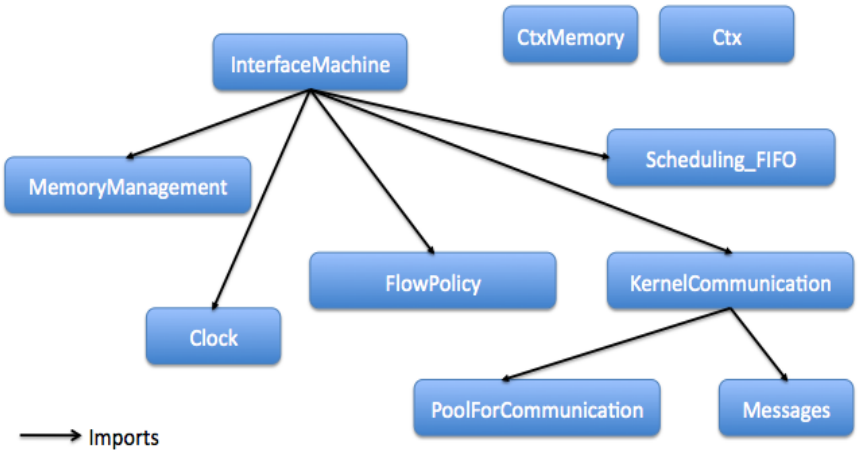

Figure 5.1: Architecture of the abstract model for the secure partitioning microkernel

5.2 Formal specification of the Secure Partitioning microkernel

This is the first section of this thesis dedicated to the development of the secure partitioning microkernel. A detailed description (and justification) of the models that are the result of this thesis is provided, as well as the methods and approach used to obtain it. The aim of this section is:

- to produce a formal model of the secure partitioning microkernel using the B Method;
- gain a very thorough understanding of the secure partitioning microkernel from a functional perspective;
- identify potential faults and shortcomings that may be useful to current implementers and any future formal verification;
- create a starting point to future implementors that want to use the B Method in the development of microkernels.

The complete code is visible in Appendix A, only the relevant parts of the machines will be described in this section. For a full view, please see the complete model.

5.2.1 Machine Ctx

The B Method uses first-order logic based on sets and set membership. Any type information is therefore also conveyed in terms of set membership. In order to define a system inside this methodology, we must first define abstract sets of "objects" inside it.

Sets are the basis for defining the microkernel types. Abstractly, we start defining sets for the resources inside the microkernel. For example, thread number, task number, etc.. Using the clause PROPERTIES, we can add properties for the sets. One important property for each set is the notion of limit. All systems manage a finite set of resources. The way to define upper limits to the resources inside the system is using the properties clause and restricting the cardinality of the abstract sets. B later ask us to prove that these limits are not exceed.

This machine called Ctx serves to define all the sets, constants and properties of the system. The abstract sets we have are, (i) the set named TASKS, which represents the possible tasks for the system; (ii) the set of the possible messages for the system MSG; and finally (iii) the set of possible threads for the system, THREADS. One enumerated set represents also defined in this machine, MODE. This set is the possible values for the communication access modes between two tasks. The mode of communication is used in the Partition Information Flow Policy (explained in detail in section 5.3). The triplet:

$$< t1, t2, m >$$
$$t1, t2 \in TASKS, \qquad t1 \neq t2, \qquad m \in MODE$$

represents the flow policies for communication between the tasks in the system. Only four modes are permitted:

- *noflow* – means that is not possible to have communication between the two tasks;
- *read* – means that a task can communicate with other tasks. However it is only possible to read from the other task;
- *write* – means that a task can communicate with other tasks. However it is only possible to write to the other task;
- *readWrite* - means that a tasks can communicate, using read or write, with other tasks.

SETS

TASKS;
MSG;
THREADS;
MODE = { noflow , read , write , readWrite }

The secure partitioning microkernel is composed by one special tasks and one special thread (*kernelTask* and *kernelThread*). Both of them belong to the set of TAKS and THREADS respectively. Since, we are giving a time slice for each task, *kernelTaskPeriod* represents the time given for the *kernelTask* to process. The total time for the system to run is given by the constant *maxtime*. The total time of the system will vary between zero and *maxtime*.

Another constants are introduced in this machine. They play an important role in the communication process. In the next subsection, the communication mechanism is explained, for now it is only important to know that *nullMsg* works like a null message. If we think in terms of a language like C, it is comparable to a structure defining a message pointing to null.

PROPERTIES

card (TASKS) $\geq$ 1 $\wedge$
card (THREADS) $\geq$ 1 $\wedge$
card (MSG) $\geq$ 1 $\wedge$
kernelTask $\in$ TASKS $\wedge$
kernelThread $\in$ THREADS $\wedge$
nullMsg $\in$ MSG $\wedge$
kernelTaskPeriod $\in$ NAT1 $\wedge$
maxPortSize $\in$ NAT1 $\wedge$
communicationTime $\in$ NAT1 $\wedge$
maxtime $\in$ NAT

The values for these constants stay open. The variables values are only introduced when the implementation level is reached. Although, some properties can be already specified. The cardinality for the set of TASKS must be greater than one, because this set contains the constant *kernelTaks*. The same reasoning can by applied to the sets THREAD and MSG.

5.2.2 Machine CtxMemory

In order to talk about memory within the model, further context is defined. Memory is an important resource that must be carefully managed. Memory is divided in three parts: kernel space, user space and kernel space for communication.

Kernel space is where the microkernel executes (i.e., runs) and provides its services. This space in memory is reserved and only the kernel thread can access it. User space is a set of memory locations in which user processes (i.e., everything other than the microkernel) run. Kernel space for communication is a set of memory locations used for communication.

The secure partitioning microkernel modeled here will use two techniques very frequently used in nowadays kernels, namely virtual memory and paging. The basis idea behind virtual memory is that the combined size of the program, data and stack may exceed the amount of physical memory available for it. Systems that use this technique make programming of large applications easier and use real physical memory (e.g. RAM) more efficiently than those without virtual memory. Another functionality for the use of virtual memory is for protection. The common mechanisms provided by most of the architectures to achieve different levels of privileges are a combination of two techniques: the processor running mode and the memory management unit (MMU). The MMU performs address translation to the real physical addresses completely transparent to the software. The program does not have any knowledge on the real physical addresses it is using. The same program may running on a physical space or another without changing the program itself. A program-generated addresses are then virtual addresses and for the virtual address space. The virtual address space is divided up into units called pages. The corresponding units in physical memory are called page frames. The context presented here deal with the constants used to perform virtual memory, pages and memory sizes for each space.

SETS

STATE = { idle , normal , coldstart , warmstart }

PROPERTIES

memorysize ∈ NAT1 ∧
memorysize ≥ 3 ∧
pagesize ∈ NAT1 ∧
pagesize < memorysize ∧
kernelspace ∈ NAT1 ∧
kernelspacecommunication ∈ NAT1 ∧
userspace ∈ NAT1 ∧
kernelspace + kernelspacecommunication + userspace = memorysize ∧
userspace mod pagesize = 0 ∧
kernelspace mod pagesize = 0 ∧
kernelspacecommunication mod pagesize = 0 ∧

INDEX = 0 .. (pagesize − 1) ∧
ADDRESS = 0 .. (memorysize − 1) ∧
PAGE = 0 .. ((userspace / pagesize) +
(kernelspacecommunication / pagesize) − 1) ∧
PAGESCOMMUNICATION = (userspace / pagesize) .. ((userspace / pagesize) +
(kernelspacecommunication / pagesize) − 1) ∧
VPAGE = 0 .. ((userspace / pagesize) − 1) ∧

The total size of available memory is given by the variable *memorysize*. The size for one page is defined in *pagesize*. The size of the kernel space plus the size of the user space and plus the size of the kernel space for communication must be equal to the total size of memory. Other property for the sizes is that each of them must be multiple of *pagesize*. The set INDEX works like an offset to navigate in one page. If we want to access the second address inside

one page, it is used the page number and the the value for index is equal to one. The set ADDRESS is equal to the total addresses that compose the memory of the microkernel. This set start in zero and goes until *memorysize* minus one. Using the previous variables the sets PAGE (representing the physical memory pages, or page frames), PAGESCOMMUNICATION (the pages dedicated for the kernel space for communication) and the VPAGE (the set of virtual pages) are constructed. To make this explanation clearer an example with some fixed values is provided. Suppose that the memory size is equal to forty and the page size is two. Then, kernel space can be equal to two and the kernel space for communication to four. From the property $kernelspace + kernelspacecommunication + userspace = memorysize$ we obtain that *userspace* is equal to thirty four. So we have the following values to the sets:
$ADDRESS = 0..39$
$PAGE = 0..18$
$VPAGE = 0..16$
$INDEX = \{0, 1\}$
$PAGESCOMMUNICATION = \{17, 18\}$
The missing page 19 belongs to the microkernel. This page is used to allocate the space occupied by the microkernel code.

In the secure partitioning microkernel every task have a state. This state can be idle, normal, coldstart or warmstart. Task state is represented in the enumerated set STATE, which contains the states mentioned before. Task states and their state transitions are shown in the following figure:

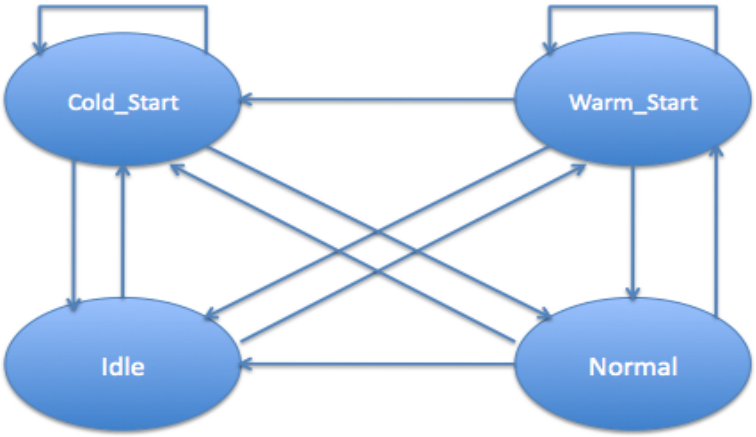

Figure 5.2: States and possible transitions for tasks

5.2.3 Machine MemoryManagement

Memory management is the job of keeping track of which parts of the memory are in use and which parts are not in use. Allocation of memory to tasks when they need it and deallocating it when they are done is the other pretended functionality. In terms of security, memory management is essential. The mechanism of protection is provided by the hardware (MMU). Although, the hardware mechanism is controlled by the microkernel.

Every task that is initialized in memory has a set of attributes. Those attributes are represented by functions connecting the tasks in memory to the respective attribute. The

mandatory attributes are a state and a duration (the time that each task will occupy the processor). Each task must have, at least four pages (i.e. does not make any sense to create a task without any page).

This machine is the memory manager of the secure partitioning microkernel. It SEES Ctx (see 5.2.1) and CtxMemory (see 5.2.2). The following variables are introduced:

- *usedTks* – represents those tasks that have been created and are in memory. It is a subset of TASKS without the kernel task;
- *tksPages* – represents the function that maps virtual pages to *usedTks*. It is a partial function because some virtual pages may not have been mapped;
- *tksDuration* – represents the function that maps *usedTks* to a duration for scheduling. It is also a total function because the attribute is mandatory;
- *tksState* – represents the function that maps *usedTks* to the state of the task.

INVARIANT

usedTks $\subseteq$ TASKS $-$ {kernelTask} $\wedge$
tksPages $\in$ VPAGE $\nrightarrow$ usedTks $\wedge$
tksDuration $\in$ usedTks $\rightarrow$ NAT1 $\wedge$
tksState $\in$ usedTks $\rightarrow$ STATE $\wedge$
ran (tksPages) = usedTks $\wedge$
dom (tksDuration) = ran (tksPages) $\wedge$
dom (tksState) = ran (tksPages) $\wedge$
$\forall$(tt).(tt $\in$ usedTks $\wedge$ tt $\in$ ran(tksPages) $\Rightarrow$ card(tksPages $\rhd$ {tt}) $\geq$ 4)

The INVARIANT clause is used to type the variables but also to define some properties that are pretended to hold to the rest of the development. In all the functions, where the variable *usedTks*, the range or domain must be same. This is the easy way to assure that the mandatory properties are assured. The last statement in the invariant guarantees that each task must have at least four virtual pages, one for heap, another for stack, other for code and finally one for data. The INITIALISATION clause is used to initialize the variables, it must respect the invariant.

INITIALISATION

usedTks := $\varnothing$ ||
tksPages := $\varnothing$ ||
tksDuration := $\varnothing$ ||
tksState := $\varnothing$

Like mentioned at the start of this section, it is the primary objective for memory management to keep track of the parts of memory in use and provide the operations for allocating and freeing parts of the memory. The first operation provided by the machine MemoryManagement is *createTaskM*.

createTaskM (pnumber , bounds , tm) =
PRE
pnumber $\in$ TASKS $\wedge$
pnumber /$\in$ usedTks $\wedge$
bounds $\in$ NAT1 $\wedge$
tm $\in$ NAT1 $\wedge$
bounds $\geq$ 4 $\wedge$
card(usedTks) + bounds $\leq$ card(TASKS)
THEN
usedTks := usedTks $\cup$ { pnumber } ||

tksDuration (pnumber) := tm ||
tksState (pnumber) := coldstart ||
ANY vpgs **WHERE**
vpgs ⊆ VPAGE ∧ card (vpgs) = bounds ∧
∀ (vp) . (vp ∈ VPAGE ∧ vp ∈ vpgs ⇒ vp /∈ dom (tksPages))
THEN
tksPages := tksPages ∪ vpgs × { pnumber }
END
END ;

This operation is dedicated to allocate memory for one task. The parameters are an elements of the set TASKS that are not member of *usedTks*, the size of the task and a duration by this order. Basically, what is done in the operation is the addition of the parameters to the variables of this machine. However, the allocation of the virtual pages is done using non-determinism, which is a good principle for a specification. We are not telling how to allocate the pages, instead, we are saying what we want to do within this operation. The request in the allocation of virtual memory for a new task is to get a subset from the set of virtual pages *vpgs*, where in this new subset none of the pages belong to the pages already in use (*dom (tksPages)*) and the size of this subset must be equal to the bounds of the task. One aspect in this operation is that when a task is created, its state is fixed in coldstart.

The aim of the operation *eliminateTaskM* is the opposite of *createTaskM*. It is used to free a task, releasing the memory that the task was occupying.

eliminateTaskM (pnumber) =
PRE
pnumber ∈ usedTks
THEN
usedTks := usedTks − { pnumber } ||
tksDuration := { pnumber } ⩤ tksDuration ||
tksState := { pnumber } ⩤ tksState ||
tksPages := tksPages ⩥ { pnumber }
END ;

In this operation the parameter *pnumber* identifies the task that is to be removed from the microkernel address space. What is done in this operation is the removing of the element from the functions previously defined.

The next operation is to change the state of the task. The entrance parameters are the task and the next mode. In the pre-condition, it is taken in consideration the previous state and the next state for the task. As described in figure 5.2, the possible transitions depends on the actual mode and the next mode, the pre-conditons for this operation guarantee that no invalid transition can happen.

setTaskMMode (pnumber , mode) =
PRE
pnumber ∈ usedTks ∧
mode ∈ STATE ∧
((tksState (pnumber) = coldstart) ⇒
(mode = coldstart) ∨ (mode = normal) ∨ (mode = idle)) ∧
((tksState (pnumber) = idle)
⇒ (mode = coldstart) ∨ (mode = warmstart)) ∧
((tksState (pnumber) = normal)
⇒ (mode = coldstart) ∨ (mode = warmstart)) ∧
((tksState (pnumber) = warmstart)
⇒ (mode = coldstart) ∨ (mode = warmstart) or

(mode = idle) ∨ (mode = normal))
THEN
tksState (pnumber) := mode
END

The last operation is only for get the mode of one task.

val ⟵ getTaskMMode (pnumber) =
PRE
pnumber ∈ usedTks
THEN
val := tksState (pnumber)
END

5.2.4 Machine Clock

This machine is used to model the clock for the system. Time properties are not easy to model using the B Method. However, some work around this thematic is being developed [20], [9], [8]. The idea is to extend the B Method so that it can help specifying and validating systems with complex timed constraints. Duration calculus is used in order to express the semantics for the B language and deduce a conservative extension allowing its use in both its original context and in the context of time-constrainded systems. For the secure partitioning microkernel, time constraints are not problematic the only necessary idea is to have a clock controlling the global system time.

In the machine Ctx the constant *maxtime* is defined. This machine SEES Ctx mainly because it uses this constant to define the possible values for the time.The utilization of *maxtime* can be seen in the INVARIANT clause.

ABSTRACT_VARIABLES
now
INVARIANT
now ∈ 0 .. maxtime
INITIALISATION
now := 0

The variable *now* indicates the current time for the system. This variable is initialized with the initial value zero. Four operations are used to change the state of the variable. The first one is the *reset* operation. In this operation the actual time is reseted to the initial value (zero).

reset =
BEGIN
now := 0
END ;

The other operation is called *tick_u*. This operation is used to increment time, changing the value of the variable *now*. Using non-determinism, *now* is changed. This operation is required when we do not know exactly the time that the microkernel will take to process a task. So the best way to model this is using non-deteminism [25]. What is done in this operation is to define an interval between the actual time plus a minimum time and the actual time plus a maximum time. Then a value from this interval is chosen to *now*.

tick_u (from , to) =
PRE
from ∈ NAT ∧

to ∈ NAT ∧
from < to ∧
now + to ≤ maxtime
THEN
now :∈ (now + from) .. (now + to)
END ;

The next operation does exactly the same from the previous one. However, non-determinism is not used to change the time.

tick (to) =
PRE
to ∈ NAT ∧
now + to ≤ maxtime
THEN
now := (now + to)
END ;

The final operation (*current_time*) is simply a getter. It returns the actual time returning the value of the variable *now*.

tt ⟵ current_time =
BEGIN
tt := now
END

5.2.5 Machine Messages

Tasks communicate using a message passing mechanism. This machine (Messages) is like an abstraction of the data in a message. A message is characterized by the following attributes:

- Source – The source of the message;
- Destination – The destination of the message;
- SIze – The size of the message.

These fields are necessary when we desire to send a message. Machine Messages SEES Ctx (see 5.2.1), it is where the set MSG is defined.

INVARIANT
message ⊆ MSG − {nullMsg} ∧
messageSource ∈ message → TASKS ∧
messageDestination ∈ message → TASKS ∧
messageSize ∈ message → NAT1 ∧
dom (messageSource) = dom (messageDestination) ∧
dom (messageSource) = dom (messageSize) ∧
∀ (mm) . (mm ∈ message ∧ mm ∈ dom (messageSource) ∧
mm ∈ dom (messageDestination)
⇒
messageSource (mm) ≠ messageDestination (mm))

The variables presented in the invariant are:

- *message* – represents the messages currently in the system. It is a subset of MSG without the *nullMsg*;
- *messageSource* – represents the total function that maps a message to the source task;

- *messageDestination* – represents the total function that maps a message with the destination task;
- *messageSize* – represents the total function that maps a message with a size.

All of the previous variables are initialized as empty sets.

INVARIANT
message := ∅ ||
messageSource := ∅ ||
messageDestination := ∅ ||
messageSize := ∅

Operations for creation and removal of messages are necessary. These operations are performed, like usually, using the B Method with pre-conditions. The first operation defined is *addNewMessage*. For this operation, four parameters are required. The first parameter is *msg*, a new element of the set MSG that does not belong to the subset *message* and is different from *nullMsg*, the null identifier for a message. Then *taskOwner* which is an element of TASKS. Like the name suggests, it is the task that is sending the message. Analogously *taskDestiny* is the task destination for the message. In the pre-condition, a clause defines that the sending task should be different from the receiver task. The fourth necessary parameter is the size of the message. Length is particularly important because the microkernel needs to know if is has enough space for retaining the message. A space limitation is required for the message. Because of the chosen design, messages cannot be divided, so the maximum size for a message is limited.

addNewMessage (msg , taskOwner , taskDestiny , length) =
PRE
msg ∈ MSG ∧
msg ≠ nullMsg ∧
taskOwner ∈ TASKS ∧
taskDestiny ∈ TASKS ∧
taskOwner ≠ taskDestiny ∧
length ∈ NAT1 ∧
length < maxPortSize
THEN
message := message ∪ { msg } ||
messageSource (msg) := taskOwner ||
messageDestination (msg) := taskDestiny ||
messageSize (msg) := length
END ;

To remove a message, it is required to pass the element to be removed *msg*. This element is then removed from the set *message* and from the respective functions.

removeMessage (msg) =
PRE
msg ∈ message ∧
msg ≠ nullMsg
THEN
message := message − { msg } ||
messageSource := messageSource − { msg ↦ messageSource (msg) } ||
messageDestination := messageDestination − { msg ↦ messageDestination (msg) } ||
messageSize := messageSize − { msg ↦ messageSize (msg) }
END ;

The next operations are only getter to the message. Operation *getMessagelength* returns the size of the message, *getMessageSource* returns the task source in a message and *getMessageDestionation* the destination for a message.

5.2.6 Machine PoolForCommunication

The mechanism for communication is composed by a pool of memory pages shared between the microkernel and the tasks. When a task want to communicate with another task, a page is mapped to the sending task. If it is the case of receiving a message, the page with the message is mapped into the receiver task. Due to the fact that we want to achieve secure partitioning, a page could not have messages from the same source to different destinations. This assures that when a page is mapped to a receiver, it only can access to messages that were sent to him. In figure 5.3 it is illustrated an example of a possible state for the pool of pages for communication. Supposing that is the task one that is running, we have two pages mapped for task one. One page with messages from task one to task two and other with messages from task one to task three. If the next task chosen to run will be task two, then the page with messages from task one to task two will be mapped to task two allowing task two to access the messages that are for herself.

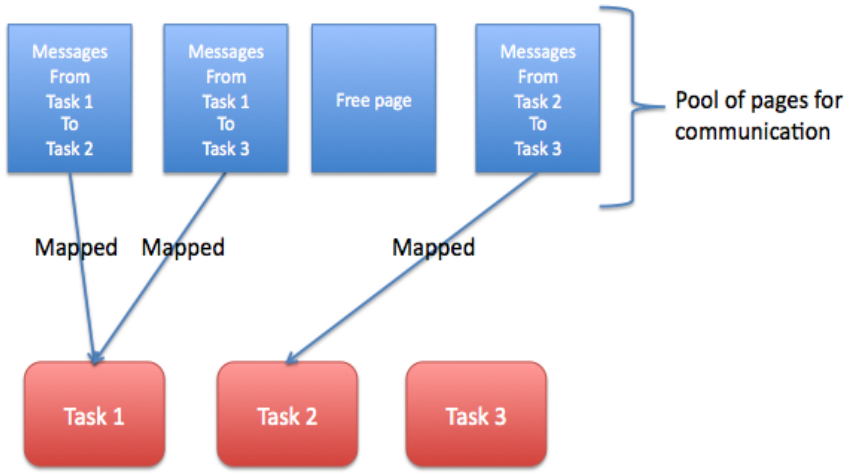

Figure 5.3: Example of a possible state for the pool for communication

This machine is responsible for the allocation, mapping and deallocation of pages belonging to the kernel space for communication (the pool of pages for communication). Two variables are used to control the pool. The first one is *usedPagesCommunication*. It is a subset of PAGESCOMMUNICATION and represents the pages currently allocate (see subsection 5.2.2). The second one is *pagesCommunicationMapping*. It is a total function that maps the pages currently being used with the task that is using it. It works like a page table for the kernel to know which page is allocated and to who. Both of these variables are initialized as empty.

INVARIANT

usedPagesCommunication $\subseteq$ PAGESCOMMUNICATION $\wedge$
pagesCommunicationMapping $\in$ usedPagesCommunication $\rightarrow$ TASKS

INITIALISATION

usedPagesCommunication := $\varnothing$ ||

pagesCommunicationMapping := ∅

Two operations are related to the action of mapping and one with deallocation. *mapNewFreePage* is the operation for mapping a new page for a task to communicate. This operation requires two arguments, an task (*tt*) and a page for communication (*fp*). The page can not belong to the set of pages already allocated for communication, i.e. must be an empty page.

mapNewFreePage (tt , fp) =
PRE
tt ∈ TASKS ∧
fp ∈ PAGESCOMMUNICATION ∧
fp /∈ usedPagesCommunication ∧
fp /∈ dom (pagesCommunicationMapping)
THEN
usedPagesCommunication := usedPagesCommunication ∪ { fp } ||
pagesCommunicationMapping (fp) := tt
END ;

The second operation is also for allocation of a page. But it works in a different way. In this operation we are changing the mapping of one page. During the communication process pages are changed from one task to another. This process of changing pages permits to tasks to send and receive messages. This operation is called by *ProcessTaskToReceive* (subsection 5.2.7). This means that the page we are mapping must belong to the set of pages already mapped.

mapPage (pg , tt) =
PRE
pg ∈ usedPagesCommunication ∧
tt ∈ TASKS
THEN
pagesCommunicationMapping (pg) := tt
END ;

The last operation is for deallocation of one page. The argument required for deallocation is a page belonging to the set of already mapped pages. The operation what does is remove the page from the set of used pages (*usedPagesCommunication*) and also from the total function (*pagesCommunicationMapping*).

unmapPage (pg) =
PRE
pg ∈ usedPagesCommunication ∧
pg ∈ dom (pagesCommunicationMapping)
THEN
usedPagesCommunication := usedPagesCommunication − { pg } ||
pagesCommunicationMapping := pagesCommunicationMapping −
{ pg ↦ pagesCommunicationMapping (pg) }
END

5.2.7 Machine KernelCommunication

As mentioned in the subsection 5.2.6, the communication is performed using a pool of pages for communication. It is necessary to provide a mechanism for information flow between partitions compliant with PIFP as a mean to achieve secure partitioning. The chosen mechanism for this is an abstraction for inter task communication called ports. Each task can contain one or several ports. A port is a page that is mapped to one task. Ports are implemented by

the process of changing pages between processes. Secure partitioning microkernel allow passing information between the tasks under his supervision since the task of transferring the information is performed by the microkernel.

A port is an endpoint of a unidirectional communication channel between a task that requests a service and a task that provides the service. In most cases, the resource that is accessed by the port (that is, named by it) is referred to as an object. Ports rights can be used to implement the PIFP efficiently.

Machine KernelCommunication is responsible for port management in the secure partitioning microkernel. This machine SEES Ctx and CtxMemomory and also INCLUDES PoolForCommunication and Messages.

INVARIANT

ports ∈ usedPagesCommunication → iseq (message) ∧
portSender ∈ usedPagesCommunication → TASKS ∧
portReceiver ∈ usedPagesCommunication → TASKS ∧
dom (portSender) = dom (portReceiver) ∧
∀ (us1 , us2) . (us1 ∈ usedPagesCommunication ∧
us2 ∈ usedPagesCommunication ∧
us1 ∈ dom (portSender) ∧ us2 ∈ dom (portReceiver)
∧ us1 = us2
⇒ portSender (us1) ≠ portReceiver (us2)) ∧
portsSize ∈ usedPagesCommunication → NAT ∧
dom(portsSize) = dom(ports) ∧
∀ (us) . (us ∈ usedPagesCommunication
⇒ ∀ (mm) . (mm ∈ message ∧ mm ∈ ran (ports (us))
⇒
SIGMA (xx) . (xx ∈ message ∧ xx ∈ ran (ports (us)) | messageSize (xx))
= portsSize (us))) ∧
∀ (up) . (up ∈ usedPagesCommunication
⇒ portsSize (up) ≤ maxPortSize) ∧
∀ (us) . (us ∈ usedPagesCommunication ⇒
∀ (mm1 , mm2) . (mm1 ∈ message ∧ mm1 ∈ ran (ports (us))
∧ mm2 ∈ message ∧ mm2 ∈ ran (ports (us))
⇒ messageDestination (mm1) = messageDestination (mm2))) ∧
∀ (us) . (us ∈ usedPagesCommunication
⇒ ∀ (mm1 , mm2) . (mm1 ∈ message ∧ mm1 ∈ ran (ports (us)) ∧
mm2 ∈ message ∧ mm2 ∈ ran (ports (us))
⇒ messageSource (mm1) = messageSource (mm2))) ∧
com ∈ TASKS ↔ TASKS

Five functions are here defined:

- *ports* – is the total function that connects pages used in communication with a injective sequence of messages. Injective sequence of messages means that in this sequence is not possible to find repeated elemets;
- *portSender* – is the total function that connects used pages in communication with the task that is sending;
- *portReceiver* – is the total function that connects used pages in communication with the task that is receiving;
- *portsSize* – is the total function that connects used pages in communication with the actual size of the port. The actual size of one port is equal to the sum off all the messages within this port;

- *com* – is a relation that connects tasks to tasks. This variable is necessary to know with each task is communicating with each other task.

The INVARIANT clause in this machine introduces some important properties. The domain for both function using *usedPagesCommunication* should be the same. Other property is that a port cannot have the same task to send and to receive. This is the property that says that a task cannot send a message for herself. A number of properties for the size of the port are then defined. The sum of each message belonging to one port must be equal to the value in the function *portsSize*. On the other hand, the size of port must be smaller or equal than the maximum size for one port (*maxPortSize*).

All the previously defined variables are initialized as empty.

INITIALISATION
ports := ∅ ||
portSender := ∅ ||
portReceiver := ∅ ||
portsSize := ∅ ||
com := ∅

The operations on this machine can be divided in two groups. One group are services offered by the microkernel and the other are internal routines for the microkernel. The first service that the microkernel provides is send a new message.

sendNewMessage (t1 , t2 , sz) =
PRE
t1 ∈ TASKS ∧
t2 ∈ TASKS ∧
t1 ≠ t2 ∧
sz ∈ NAT1 ∧
sz ≤ maxPortSize ∧
(card (usedPagesCommunication) <
card (PAGESCOMMUNICATION)) ∧
(¬ (∃ (up) . (up ∈ usedPagesCommunication ∧
portSender (up) = t1 ∧ portReceiver (up) = t2 ∧
portsSize (up) + sz ≤ maxPortSize
∧ pagesCommunicationMapping (up) = t1)))
THEN
ANY newMessage , up **WHERE**
newMessage ∈ MSG ∧
newMessage /∈ message ∧
newMessage ≠ nullMsg ∧
up ∈ PAGESCOMMUNICATION ∧
up /∈ usedPagesCommunication ∧
up /∈ dom (pagesCommunicationMapping)
THEN
mapNewFreePage (t1 , up) ||
addNewMessage (newMessage , t1 , t2 , sz) ||
ports (up) := [] ↩ newMessage ||
portSender (up) := t1 ||
portReceiver (up) := t2 ||
portsSize (up) := sz ||
com := com ∪ { t1 ↦ t2 }
END
END ;

The aim of this operation is to create a new message and add it to the port. The *sendNewMessage* operation creates a new port, mapping a free page to this task. The parameters are two tasks (sender and receiver task) and the size of the message. The required pre-conditions for this operation are:

- The sender task (*t1*) and the receiver task (*t2*) must be different;
- The size of the message must be less or equal than the total size of a port;
- The microkernel must have at least one empty page for communication;
- The sender task and receiver task cannot have any other allocated page with space (i.e. free memory), unless the space in that page is less than the size of the new message.

If all of the previous pre-conditons are true, then the body of the operation is executed. Once again, non-determinism is used. A new message is created, with the desired sender, receiver and size. A free page for communication is chosen, this page will be used to allocate the new message. The free page is allocated to the sender task using the operation *mapNewFreePage* provided by PoolForCommunication machine. At the end, it is added the message to the sequence of messages in that port. Since it is a new message, the port sequence for that page must be equal to empty. The variable *com* is also updated, registering that new channel is connecting the task *t1* and *t2*.

The last operation is used to send a new message. Supposing that we already sent a message from task one to task two with a size less than the maximum size of a port. It does not make sense to waste space and allocate a new page if we are sending from the same task to the same receiver and the port has enough space to handle the message. Operation *sendMessage* represents this case.

sendMessage (t1 , t2 , sz) =
PRE
t1 ∈ TASKS ∧
t2 ∈ TASKS ∧
t1 ≠ t2 ∧
sz ∈ NAT1 ∧
∃ (up) . (up ∈ usedPagesCommunication ∧
portSender (up) = t1 ∧ portReceiver (up) = t2 ∧
portsSize (up) + sz ≤ maxPortSize ∧
pagesCommunicationMapping (up) = t1)
THEN
ANY newMessage , up **WHERE**
newMessage ∈ MSG ∧
newMessage /∈ message ∧
newMessage ≠ nullMsg ∧
up ∈ usedPagesCommunication ∧
portSender (up) = t1 ∧ portReceiver (up) = t2 ∧
portsSize (up) + sz ≤ maxPortSize
THEN
addNewMessage (newMessage , t1 , t2 , sz) ‖
ports (up) := ports (up) ↩ newMessage ‖
portsSize (up) := portsSize (up) + sz
END
END ;

The changes comparing with *sendNewMessage* are few. If we take a deeper look, the only change in the pre-condition is that instead of having the negation in the last part, we do not

have anything. The behavior of the operation is almost the same, but instead of getting a new page we will obtain an already used page with the same sender and receiver.

The last operation for services that the microkernel offers is *receiveMessage*. This operation receives two parameters, *tto* (the receiver task) and *tfrom* (the sender task).

```
res ⟵ receiveMessage ( tto , tfrom ) =
    PRE
        tto ∈ TASKS ∧
        tfrom ∈ TASKS ∧
        tto ≠ tfrom ∧
        ∃ ( up ) . ( up ∈ usedPagesCommunication
        ∧ portSender ( up ) = tfrom
        ∧ portReceiver ( up ) = tto ∧ ports ( up ) ≠ []
        ∧ pagesCommunicationMapping ( up ) = tto
        )
    THEN
        ANY msg , up WHERE
            msg ∈ message ∧
            messageSource ( msg ) = tfrom ∧
            messageDestination ( msg ) = tto ∧
            up ∈ usedPagesCommunication ∧
            pagesCommunicationMapping ( up ) = tto ∧
            portSender ( up ) = tfrom ∧
            portReceiver ( up ) = tto ∧
            ports ( up ) ≠ [] ∧
            msg = last ( ports ( up ) )
        THEN
            res := msg ||
            IF size ( ports ( up ) ) = 1
            THEN ports ( up ) := []
            ELSE ports ( up ) := front ( ports ( up ) )
            END ||
            portsSize ( up ) := portsSize ( up ) − messageSize ( msg ) ||
            removeMessage ( msg )
        END
    END ;
```

To a task receive a message, it is necessary to have, at least, one page for communication where the sender is the argument passed in *tfrom* and the receiver the task *tto*. The message that is taken from the sequence is always the one that is in the head of the sequence. The body of the previous operation, selects a message, (i) where the source is *tfrom* and the sender is *tto*, (ii) the page must belong to the task that is requesting the service, and (iii) the sequence must be different from empty. If is only a message in the port, then the sequence will become equal to empty else it is removed the first message from the sequence. It is also necessary to update the size of the port and eliminate the message from the system. The operation *removeMessage* is imported from the machine Messages.

Operations for internal routines of the microkernel are also defined in this machine. The first operation is *processTaskToReceive*.

```
processTaskToReceive ( tt ) =
    PRE
        tt ∈ TASKS
    THEN
```

ANY pg **WHERE**
pg ∈ usedPagesCommunication ∧
portReceiver (pg) = tt ∧
pagesCommunicationMapping (pg) ≠ tt
THEN
mapPage (pg , tt)
END
END ;

When a context switch is performed, a new task starts running. It is necessary to see if the new task that will run has any messages to receive. If it has, then is necessary to map the page to the new task, otherwise it cannot access to the page. The operation *processTaskToReceive* checks if the task *tt* have any port in which the messages are for herself. If is this the case, then the imported operation *mapPage* is performed. The last internal routine is the operation *cleanEmptyPorts*.

cleanEmptyPorts =
PRE
∃ (up) . (up ∈ usedPagesCommunication ∧ ports (up) = [])
THEN
ANY up **WHERE**
up ∈ usedPagesCommunication ∧
ports (up) = []
THEN
portSender := portSender − { up ↦ portSender (up) } ||
portReceiver := portReceiver − { up ↦ portReceiver (up) } ||
ports := ports − { up ↦ ports (up) } ||
portsSize := portsSize − { up ↦ portsSize (up) } ||
unmapPage (up)
END
END

This operation works like a daemon. When the kernel task assumes control, this is one of the operations that we have to proceed (Figure 5.4). It is necessary to eliminate the mapping for pages in which the ports are empty. The kernel task execute always between the execution of two tasks. This is, if task one is running, when she terminates, kernel task will execute before changing to the next task.

5.2.8 Machine Scheduling_FIFO

Machine Scheduling_FIFO, like the name says, is a fifo. It has a formal parameter, in this case, a SET CC.

MACHINE
Scheduling_FIFO (CC)

ABSTRACT_VARIABLES
fifo
INVARIANT
fifo ∈ iseq (CC)
INITIALISATION
fifo := []

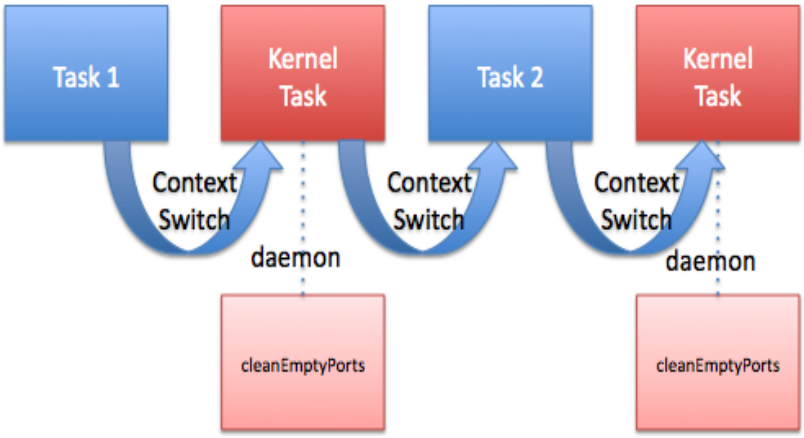

Figure 5.4: Behavior of the operation *cleanEmptyPorts*

An abstract variable *fifo* represents the state of the machine. This variable implements a injective sequence of elements of the type CC. Five operations are defined for this machine.

- *add* – Adds an element to the sequence;
- *isEmpty* – Checks if the sequence is empty. Returns true case is empty, false otherwise;
- *topFifo* – Returns the top element of the sequence;
- *remove* – Removes the top element of the sequence;
- *change* – Puts the top element at the tail and push each element one position straight forward.

This machine works like a scheduler for the tasks. The element on the top is the task that is running. The other ones are waiting for their time.

5.2.9 Machine Interface

The purpose for this machine is only to provide a efficient and enlightening animation using the ProB. For achieve this objective, this machine is pretended to work like a sequential program. It call the operations of the imported machines almost sequentialy. In a "normal" development using the B Method, this machine would not be required, it is only here for animation. This machine IMPORTS the machines clock, MemoryManagement, kernelCommunication, Scheduling_FIFO(TASKS) and FlowPolicy (this last one will be thoroughly explained in the section 5.3) and SEES Ctx and CtxMemory.

The variables for this machine are:

- *elapseTime* – represents the time passed for the currently executing partition;
- *actualTask* – reprsenets the task that is currently executing;
- *actualThread* – represents the actual running thread;
- *tperiod* - represents the period of time that a task will run;
- *tasks* – constitutes the subset of tasks that are present in system;
- *threads* – constitutes the subset of threads that are present in the system;

- *tthreads* – is the total function that maps each thread present in *threads* to the tasks in *tasks*;
- *reSched* – is a boolean variable indicating if is necessary to reschedule the system or not;
- *configure* – is a boolean variable indicating if the configuration was already done or not;
- *kernelRemainTime* – represents the time that the kernel still have to run.

Interface machine code is presented in the Appendix A, in this subsection only a explanation of what each function does is described. Those functions are:

- *createTask* – This function is used to create a new task. The task is added to memory and to scheduler;
- *addThreadToTask* – This function permits the creation of a thread. The created thread is added to the correspondent task;
- *endConf* – Function that is used to end the configuration. This operation uses *generateConf* from the machine FlowPolicy;
- *changeToKernelTask* – Function that passes the control to the kernel. The scheduler is updated and *cleanEmptyPorts* in case of necessity;
- *kernelTaskExecution* – This function is used by the microkernel to map pages for communication and to update the clock;
- *kernelTaskEndExecution* – Function that changes the control of the microkernel ti a next task that will execute;
- *run* – Function that is used to simulate a thread execution;
- *sendMessageCurrentTask* – Function that is used to send a message from the current task to a destination task;
- *receiveMessageCurrentTask* – Function that is used to receive a message.

5.2.10 Animation with ProB

To a better understanding of the secure partitioning kernel behavior, animation provides an important role. ProB, the animation tool, is used for animate and understand the models. Because machine Interface (see 5.2.9) includes and uses the operations of the included machines, it is possible to see all the machines interacting with each other. The behavior of the secure partitioning microkernel is well visible using animation.

Several differences between the expected behavior of the model and the real behavior were found during the development stage. ProB discovers situations of deadlocks and invariant violations very easily. Model checking is also very useful, because it automatically tries to reach all the states of the model and warns if some problem is found. The all secure partitioning microkernel was animated and model checked using different parameters. It is possible to change the cardinality of sets, the number of possible operations at each stage and the number of initializations.

ProB has some limitations. During the development process of the project, some machines were not able to be animated. Apparently, the reasoning applied in the machines was correct. After some time trying to understand what was going wrong, it was found that ProB sometimes does not animate well the machines due to some limitations. This is quite normal, for a tool that is being development in the context of research. It must be said, that ProB developers always answered to any doubt that was presented, revealing an extraordinary availability.

ProB should be considered an useful tool to help in the development process. However, the animation and model checking that this tool provides never should be considered a possible substitution to proof.

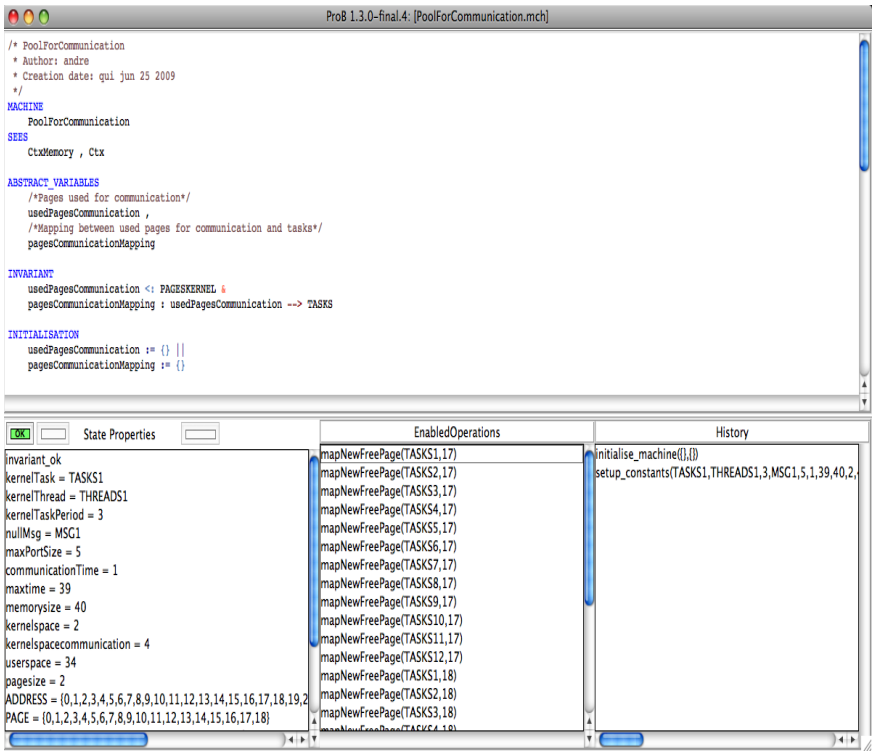

Figure 5.5: Example of ProB animation

5.3 Partition Information Flow Policy

Partition information flow policy is an important issue in the communication process. Access to information depends on the rights of each task. The aim of the partition information flow policy is to control the rights in the communication process to each task.

A flow policy is the set of possible flows for all the tasks in the system. When a task tries to communicate with another task, the kernel first checks to see if that flow is possible. Initially, a configuration indicating the different flows privileges between each task is provided. By default, no communication between tasks is allowed. So, if no configuration is given, tasks cannot communicate with each others.

A flow is a triplet composed by a first task, a second task and a privilege. The possible types for privileges are:

- *noflow* – means that the first task cannot communicate to the second task;
- *read* – means that the first task can only read from the second task;
- *write* – means that the first task can only write to the second task;
- *readWrite* – means that the first task have full privileges, it can read and write to the second task.

A task cannot communicate with herself, so the type of communication is *noflow*. This property is an easy one of escape, it is so obvious that is natural to the implementor not to think about it. However, formal methods are good dealing with this type of inconsistencies.

To communicate tasks need to use ports. So, when a task request a service, it first needs to lookup for the task port that is providing the service. If is possible to connect to the port, it can send or receive a message (depending on the privileges) to the connected port.

In this section a complete development of a partition information flow policy is presented (Figure 5.6).

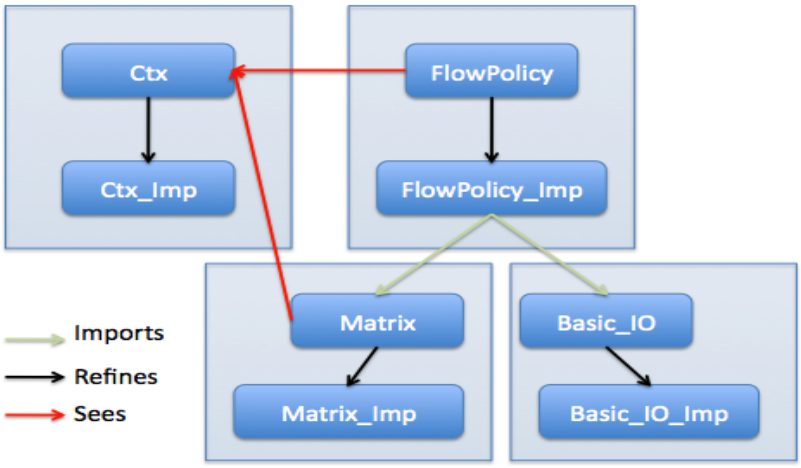

Figure 5.6: Partition Information Flow Policy Architecture

The next subsections explain in detail each component in the development process until reach a level in which is possible to generate code automatically.

5.3.1 Machine FlowPolicy

Machine FlowPolicy defines the properties for the partition information flow policy. It SEES the machine Ctx (see subsection 5.2.1).

INVARIANT

flowPolicy $\in$ ((0 .. nmaxTasks) $\times$ (0 .. nmaxTasks)) $\rightarrow$ MODE $\wedge$
$\forall$ (t1 , t2) . (t1 $\in$ (0 .. nmaxTasks) $\wedge$ t2 $\in$ (0 .. nmaxTasks)
$\wedge$ (t1 , t2) $\in$ dom (flowPolicy) $\wedge$ t1 = t2
$\Rightarrow$ flowPolicy (t1 , t2) = noflow) $\wedge$
conf $\in \mathbb{B}$

Two variables are used, *flowPolicy* and *conf*. The first variable represents the flows between all the tasks in the system. A total function, where the domain is a tuple with two enumerated sets (from zero to *nmaxTasks*). The range is the privilege mode.

As can be seen in the INVARIANT, the second clause says that if two tasks are the same, then the mode of communication between them are *noflow*. An example for a better comprehension is present next. Supposing that the number of *nmaxTasks* is two, we have the following: $dom(flowPolicy) = \{(0,0);(0,1);(0,2);(1,0);(1,1);(1,2);(2,0);(2,1);(2,2)\}$

The second variable is *conf*. This variable is a boolean indicating if the configuration was already done or not.

The operations in this machine are used to control flows in the communication process. Four operations are defined in this machine. The first operation is *generateConf*. The purpose is to generate a possible configuration of flows to the system.

generateConf =
PRE
conf = FALSE

THEN
ANY newConf **WHERE**
newConf ∈ ((0 .. nmaxTasks) × (0 .. nmaxTasks)) → MODE ∧
∀ (t1 , t2) . (t1 ∈ (0 .. nmaxTasks) ∧ t2 ∈ (0 .. nmaxTasks) ∧
(t1 , t2) ∈ dom (newConf) ∧ t1 = t2 ⇒ newConf (t1 , t2) = noflow)
THEN
flowPolicy := newConf
END ‖
conf :∈ 𝔹
END ;

This operation has the pre-condition that the variable *conf* must be equal to false. This means that no configuration was already done. The body of the operation, once again, is defined using non-determinism. A new configuration is generated respecting the invariant. The variable *flowPolicy* is then replaced by the new configuration.

The second operation is *lookup*. If a task wants to communicate with another task, it is necessary to see first if is possible to communicate with the object belonging to the task that is providing the service. The next operation checks if the mode for the tasks involved in the communication is different from *noflow*.

res ⟵ lookup (t1 , t2) =
PRE
t1 ∈ (0 .. nmaxTasks) ∧
t2 ∈ (0 .. nmaxTasks)
THEN
IF flowPolicy (t1 , t2) ≠ noflow
THEN res := TRUE
ELSE res := FALSE **END**
END ;

Two parameters are used in this machine. The first one, *t1*, is the task that is requesting the service. The second, *t2*, is the task that is providing the service. Both parameters must be a valid task, between the interval zero to *nmaxTasks*. The result for this operation is true if the the flow between *t1* and *t2* is different from *noflow*.

Operation *send* is third operation, it is used to see if a task can send a message to another task.

res ⟵ send (t1 , t2) =
PRE
t1 ∈ (0 .. nmaxTasks) ∧
t2 ∈ (0 .. nmaxTasks) ∧
flowPolicy (t1 , t2) ≠ noflow
THEN
IF flowPolicy (t1 , t2) = write ∨ flowPolicy (t1 , t2) = readWrite
THEN res := TRUE
ELSE res := FALSE **END**
END ;

For a task *t1* to send a message to *t2* is necessary that the *flowPolicy* between them be equal to *write* or *readWrite*. The precondition $flowPolicy(t1, t2) \neq noflow$ is used to assure that the task can communicate with the object of the destination task. This operation returns true case is possible to send a message form *t1* to *t2* and false otherwise.

The last operation is *receive*. This operation is used to see if a task can receive a message from another task.

res ⟵ receive (t1 , t2) =
PRE
t1 ∈ (0 .. nmaxTasks) ∧
t2 ∈ (0 .. nmaxTasks) ∧
flowPolicy (t1 , t2) ≠ noflow
THEN
IF flowPolicy (t1 , t2) = read ∨ flowPolicy (t1 , t2) = readWrite
THEN res := TRUE
ELSE res := FALSE **END**
END

As in the previous operation (*send*), this operation also requires that the receiver task can communicate with the sending task. This property is assured in the invariant clause. To a task receive from another task, the *flowPolicy* between them must be equal to *read* or *readWrite*. In the case of success this operation returns true, otherwise returns false.

5.3.2 Machine Matrix

Machine *FlowPolicy* needs to be refined and implemented to generate code automatically. Like seen in the previous machine, the invariant *flowPolicy* is very closed to a matrix. This machine is then a model of a matrix to use in the implementation of the *FlowPolicy* machine.

The variable used here is *matrix* and as the name says is the abstract representation of a matrix.

matrix ∈ ((0 .. nmaxTasks) × (0 .. nmaxTasks)) → MODE

Two operations are required to interact with the matrix, *add* and *get*. The first operation add a value *mm* to the position *(t1,t2)*. The second operation gets a value from the position *t1,t2*.

add (t1 , t2 , mm) =
PRE
t1 ∈ 0 .. nmaxTasks ∧
t2 ∈ 0 .. nmaxTasks ∧
mm ∈ MODE
THEN
matrix (t1 , t2) := mm
END ;

val ⟵ get (t1 , t2) =
PRE
t1 ∈ 0 .. nmaxTasks ∧
t2 ∈ 0 .. nmaxTasks
THEN
val := matrix (t1 , t2)
END

In both operations, the pre-conditon guarantees that the parameters are valid.

5.3.3 Machine BASIC_IO

Machine BASIC_IO is the definition of input and output functions. This machine belongs to the library of basic machines provided by Atelier B. The code for this machine is in the appendix A. This machine has no invariant, only a set of operations. Those operations are:

- *INTERVAL_READ* – reads a number in one expected interval;
- *INT_WRITE* – writes a number into stdout;
- *BOOL_WRITE* – write true or false into stdout;
- *CHAR_READ* – reads a character from the stdin;
- *CHAR_WRITE* – write a character into the stdout;
- *STRING_WRITE* – write a string into the stdout.

5.3.4 Machine FlowPolicy_Imp

For the generation of code automatically, it is necessary to have an implementation using the B0 language (a subset of the B language). Machine *FlowPolicy_Imp* is the implementation of *FlowPolicy*. This machine IMPORTS *BASIC_IO* and *Matrix*. The process of decomposition using imports, makes more easy the process of code generation. To implement this machine, invariant of the imported machines are connected to the invariants of the abstract machine. This process of "gluing the invariants" is necessary to prove that the implementation does exactly the same that the abstract machine. The SEES clause is also used, like in the abstraction. This machine continues seeing the Ctx machine.

A new variable is introduced here. Variables to be implementable must belong to the B0 language and be defined like concretes.

CONCRETE_VARIABLES
confC

Invariants are glued using the INVARIANT clause in the implementation machine.

INVARIANT
confC $\in \mathbb{B} \wedge$
confC = conf $\wedge$
flowPolicy = matrix

In this case, the invariants are equal $flowPolicy = matrix$ so there are not many changes to do. Basically, it is only used the process of decomposition to achieve the implementation. The concrete variable *confC* will be equal to the abstract variable *conf*,

The first operation is *generateConf*. Like in the abstract machine, this operation has the purpose of generate a configuration.

```
generateConf =
    BEGIN
        VAR cont1 , cont2 , ch IN
            cont1 := 0 ;
            cont2 := 0 ;
            IF confC = TRUE THEN
                STRING_WRITE ( "\tCONFIGURATION ALREADY DONE\n" )
            ELSE
                WHILE cont1 ≤ nmaxTasks DO
                    WHILE cont2 ≤ nmaxTasks DO
                        IF cont1 ≠ cont2 THEN
                            STRING_WRITE ( "\tSET MODE FOR∈\n" ) ;
                            INT_WRITE ( cont1 ) ;
                            STRING_WRITE ( "\t→\t" ) ;
                            INT_WRITE ( cont1 ) ;
                            STRING_WRITE ( "\n" ) ;
```

```
                    STRING_WRITE ( "\t 1 − NO FLOW \n" ) ;
                    STRING_WRITE ( "\t 2 − READ \n" ) ;
                    STRING_WRITE ( "\t 3 − WRITE \n" ) ;
                    STRING_WRITE ( "\t 4 − READ AND WRITE \n" ) ;
                    ch ⟵ CHAR_READ ;
                    CASE ch OF
                        EITHER 1 THEN add ( cont1 , cont2 , noflow )
                        OR 2 THEN add ( cont1 , cont2 , read )
                        OR 3 THEN add ( cont1 , cont2 , write )
                        OR 4 THEN add ( cont1 , cont2 , readWrite )
                        ELSE add ( cont1 , cont2 , noflow ) END
                    END
                ELSE
                    add ( cont1 , cont2 , noflow )
                END ;
                cont2 := cont2 + 1
            INVARIANT
                cont2 ∈ 0 .. nmaxTasks + 1
            VARIANT
                nmaxTasks + 1 − cont2
            END ;
            cont1 := cont1 + 1
        INVARIANT
            cont1 ∈ 0 .. nmaxTasks + 1
        VARIANT
            nmaxTasks + 1 − cont1
        END
    END ;
    confC := TRUE
  END
END ;
```

Two local variables are used in this operation (*cont1* and *cont2*). Both of them are counters to be used within the while cycles. To fill the matrix, both counters are used to first run the lines and then the columns. The invariant of the first cycle is $cont1 \in 0..nmaxTasks + 1$. It goes until $nmaxTasks + 1$ because the variable is first incremented and only after that is actualized. The variant is a value that is decreasing at each iteration of the cycle. In this case, $nmaxTasks + 1 - cont1$ is an expression that decreases at each iteration, because the value of $cont1$ is increasing and $nmaxTasks + 1$ is a constant value. The same reasoning can be applied to the second cycle.

Inside both cycles, various operations from the imported machines are called. One that is particular important is *add*. Depending on the value that the user chooses, *add* is called with the correspondent parameter. Although, if $cont1$ and $cont2$ are equal, the parameter for *add* is *noflow* , this is because the tasks are the same, so the parameter must be *noflow*.

The second operation is *lookup*. The pre-conditions in this operation are replaced by if conditions. The local variable *aux* gets the value in the matrix for the parameters. If the value is equal to *noflow* the return value for the operation is false, else returns true. If the parameter are not valid, the result is also false and a error message is showed.

```
res ⟵ lookup ( t1 , t2 ) =
  BEGIN
    VAR aux IN
```

```
            IF t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0  ∧ confC = TRUE THEN
                aux ⟵ get ( t1 , t2 ) ;
                IF aux = noflow THEN res := FALSE ELSE res := TRUE END
            ELSE
                STRING_WRITE ( "\tERROR − TASKS DO NOT EXIST IN THE SYSTEM\n" ) ;
                res := FALSE
            END
        END
    END ;
```

Other operations are *send* and *receive*. Like in the previous operation, the general strategy is the substitution of the pre-conditions, defined in the abstract machine, by if conditions and use of imported operations.

```
res ⟵ send ( t1 , t2 ) =
    BEGIN
        VAR aux IN
            IF t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0 ∧ confC = TRUE THEN
            aux ⟵ get ( t1 , t2 ) ;
                IF aux = write ∨ aux = readWrite THEN res := TRUE ELSE res := FALSE END
            ELSE
                STRING_WRITE ( "\tERROR − TASKS DO NOT EXIST IN THE SYSTEM\n" ) ;
                res := FALSE
            END
        END
    END ;

    res ⟵ receive ( t1 , t2 ) =
    BEGIN
        VAR aux IN
            IF t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0 ∧ confC = TRUE THEN
                aux ⟵ get ( t1 , t2 ) ;
                IF aux = read ∨ aux = readWrite THEN res := TRUE ELSE res := FALSE END
            ELSE
                STRING_WRITE ( "\tERROR − TASKS DO NOT EXIST IN THE SYSTEM\n" ) ;
                res := FALSE
            END
        END
    END
```

5.3.5 Machine Matrix_Imp

This machine is the implementation of the Matrix abstract machine. Matrix machine is easily implemented, transforming the invariant of the abstract machine in a concrete one.

CONCRETE_VARIABLES
matrixC
INVARIANT
matrixC ∈ ((0 .. nmaxTasks) × (0 .. nmaxTasks)) → MODE ∧
matrixC = matrix

This is easy because the type of *matrixC* is already an implementable type. So is possible to generate code for this type. The operations are implemented using the technique of substitution

of the pre-conditions by if clauses. The code for the implemented machine is presented in Appendix A.

5.4 Prex with Partitioning Information Flow Policy

Like mentioned in the previous sections, ports rights can be used to implement the PIFP efficiently. Prex does not have such ports, however, it has something similar called objects. Prex communication mechanism was already explained in chapter 3 in section 3.5. Objects in Prex provide all the characteristics to implement PIFP.

To integrate the PIFP with the target microkernel, some changes in the code of the target microkernel were necessary. Although, these changes were done in a manner to become the added PIFP like an invisible layer in Prex. All the necessary functions to perform communication were duplicated and added a suffix "my" at the end. This gluing code, like is showed in Figure 5.7, is the code that connects the code generated from Atelier B with Prex.

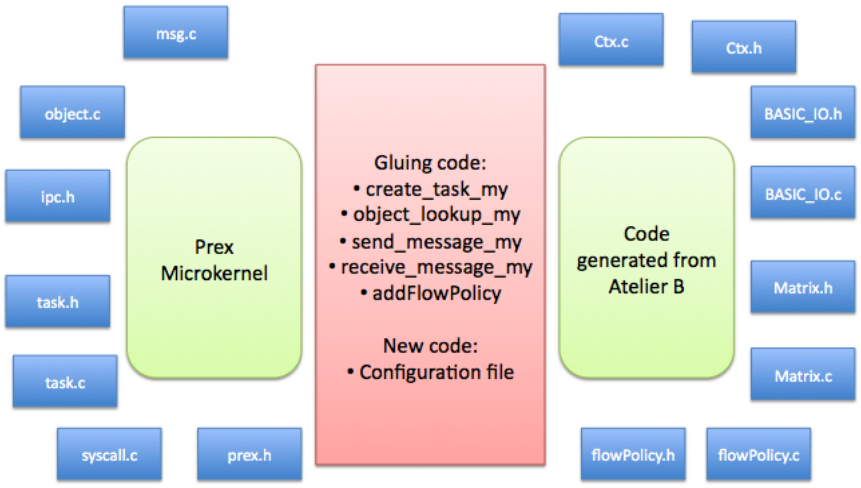

Figure 5.7: Prex and Partition Information Flow Policy

Tasks in Prex does not have any unique identifier. They have an attribute called *name*, but this attribute can change during task execution using the function *task_name*. A necessary mechanism to assure unique identification to tasks were provided adding a new camp to the task structure called task_number. So the new structure to tasks will be:

```
/*
   Task struct
 */
struct task {
        int             magic;          /* magic number */
        char            name[MAXTASKNAME]; /* task name */
        struct list     link;           /* link for all tasks in system */
        struct list     objects;        /* objects owned by this task */
        struct list     threads;        /* threads in this task */
        vm_map_t        map;            /* address space description */
        int             suscnt;         /* suspend counter */
        cap_t           capability;     /* security permission flag */
```

```
        struct timer     alarm;          /* alarm timer */
        void (*handler)(int);            /* pointer to exception handler */
        task_t           parent;         /* parent task */
        int              task_number;    /* @André Passos -
        task number for commmunication */
};
```

Now, when it is necessary to create a new task, the new function *create_task_my* receives an integer that corresponds to the task number. A configuration for the flow policy is necessary when we start executing the microkernel. By default, all the tasks cannot communicate with each other. A configuration file is passed before the microkernel starts executing and mechanism for the secure generation of the flow policies transforms the configuration file in a set of calls to the microkernel generate the flow policy. An example of a transformation of a configuration file in a C file is presented in the Appendix B. The function with the responsibility of add a flow policy is *addFlowPolicy*.

To send or receive messages, it is necessary for tasks to be in contact with objects. The function, *object_lookup_my*, replaces the normal lookup for objects in Prex. The only difference between this one and the original one, is that *object_lookup_my* asks to the flow policy if is possible for the task that is requesting the service to talk with the task that is providing the service. If is possible to communicate, this is, the flow policy is different from *noflow*, then a pointer to the object is passed to the task that is requesting the service. Case it is not possible to talk, a message is showed and the pointer is returned as null.

After connecting to the object, every time a task wants to send or receive a message, it should use *send_message_my* to send a message or *receive_message_my*. Both of these functions access to the flow policy to see if the task that is requesting the service has enough privileges for it.

The difficulty of integrating the code generated by Atelier B with Prex was not very difficult. The code generated automatically does not changed anything. Most of the work was done on the side of Prex. A comprehension of all the aspects involved in the communication and the understanding of how a microkernel works were a precious help. It was necessary to add new system calls, new routines to the API and other more or less complex changes. As conclusion, it is possible to be said that it was more harder to change the code of Prex than the integration of PIFP and Prex.

5.5 Verification and Validation

Verification is an activity that takes place before validation. Verification evaluates documents, plans, code, requirements and specification. Validation involves the execution of tests designed to cover each of the specific requirements. Atelier B provides tools for verification. Mainly the automatic and interactive prover. In this section some number about the complete development of the PIFP work will be presented. The global state of the project is shown in Table 5.1.

The B project comprises eight components (abstract machines and implementation), of which two basic machines (BASIC_IO and BASIC_IO_Imp). The number of proof obligations, for the totally of its components, is a hundred and forty-three. Where a hundred and twenty-six were trivial proof obligations; i.e. directly demonstrated by the Atelier B. No proof work is to be executed for such proof obligations. Seventeen non trivial proof obligations are need to demonstrate.

In a qualitative evolution, two aspects are need to be taken in consideration. The architecture of the project and the proof coverage. Considering the first, this B project does not compromise any refinement. The structures of the manipulated data and the algorithms de-

Component	TC	POG	nPO	nUN	%Pr	B0C
BASIC_IO	OK	OK	0	0	-	OK
BASIC_IO_Imp	OK	OK	3	3	0	OK
Ctx	OK	OK	0	0	-	OK
Ctx_Imp	OK	OK	1	0	100	OK
FlowPolicy	OK	OK	4	0	100	OK
FlowPolicy_Imp	OK	OK	108	14	87	OK
Matrix	OK	OK	4	0	100	OK
Matrix_Imp	OK	OK	23	0	100	OK

Table 5.1: Global state of PIFP project

scribed in the body of the operations being sufficient clear, it has not been necessary to introduce any data or algorithm refinements. The architecture clauses used are of INCLUDES and SEES. The link of SEES type are few, this is in order to improve the readability and maintainability of the components. These links are created to consult context components (Ctx). The link of IMPORTS is used in the decomposition process to achieve implementation.

The proof coverage rate corresponds to the percentage of proof obligations obtained by the use of the automatic prover is 88%. The interactive prover were not used in this project. So the PIFP project is 88% proved. However, animation/model-checking performed by ProB and manual inspection to the non proved proof obligations (12%) reveal, at first look no problem. Off course, the project is only a 100% correct if all the proof obligations are discharged.

All of the proof obligations revealed (by visual inspection) that is necessary to proof the remaining proof obligations (12% of the total). Currently, this work is being performed. Only after the conclusion of this part of the work, it is possible to say that the project is 100% correct.

Generally the proof coverage rate obtained by the use of automatic prover only is approximately 70%. It is estimated that below 50%, the B project necessitates re-handling (either by decomposing the B modules or introducing refinements, or expressing the need in another way). The performance achieved by the automatic prover on this project can be explained by (i) the use of simple data structures, easily manipulated by the prover; (ii) few calculus in the algorithms; (iii) use of few refinements; and finally (iv) the decomposition obtained by the IMPORTS links permits not only to divide the B model in specific activities but also to divide the proof work. The proof obligations generated are, as such, ore simple and in less number.

Validation was performed by tests. Using a machine emulator and virtualizer, QEMU, the microkernel is tested. Boot tasks are special tasks that when the microkernel is load into memory, the code of the tasks will also be load. Then, when the microkernel starts running, those tasks will immediately also run.

Boot task are C programs, using the API of Prex to make system calls. Prex and the PIFP is then tested using boat tasks. The inter process communication is tested creating different tasks and defining the flow policy between them. Maintaining the number of tasks and changing the flow policy between them, it is possible to test the communication process and more precisely the PIFP.

Table 5.2 presents the flow policy for two tasks.

Each one of the different flow policies is tested using boot tasks. The sequence of execution for the boot tasks is:

1. Two tasks are created, task $t1$ and $t2$. Associated to each task there are an object;
2. A flow policy is created. All the flow policies described in Table 5.2 are tested;
3. Task $t1$ connects to the object of task $t2$;

	Task $t1$	Task $t2$
Task $t1$	noflow	noflow
Task $t2$	noflow	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	read
Task $t2$	read	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	write
Task $t2$	write	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	read
Task $t2$	write	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	write
Task $t2$	read	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	readWrite
Task $t2$	readWrite	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	readWrite
Task $t2$	write	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	readWrite
Task $t2$	read	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	write
Task $t2$	readWrite	noflow

	Task $t1$	Task $t2$
Task $t1$	noflow	read
Task $t2$	readWrite	noflow

Table 5.2: Possible configurations for two tasks

4. Task $t2$ connects to the object of task $t1$;
5. Task $t1$ sends a message to task $t2$;
6. Task $t2$ receives the message from $t2$;
7. Task $t2$ sends a message to task $t1$;
8. Task $t1$ receives the message from $t2$.

This sequence defines the protocol for testing. Depending on the flow policy implemented, the execution can complete all the steps or not. For example, if the flow policy is *readWrite* between task $t1$ and task $t2$ and the same for $t2$ and $t1$. It should cover all the previous points. On the other hand, if the flow policy between $t1$ and $t2$ is *readWrite* and between $t2$ and $t1$ is *noflow*. Task $t1$ can connect to the object of $t1$ and send the message. But task $t2$ could not receive from $t1$ because of the flow policy is *noflow*. All the previous configuration were tested. The same was done with three and four tasks. No problem was found for each configuration. It is obvious that this is not enough for guarantee the absence of errors. This verification should be seen like a initial and not complete validation. Much more work should be done to guarantee correctness, mainly the proof of all the proof obligations.

5.6 Summary

This chapter presented all the realized work. A complete abstract model of the secure partitioning microkernel is presented in section 5.2. This part was important for a better understanding of the problem. ProB helped very much in the modeling stage. At the same time that the machines were modeled with Atelier B, animation and model checking provided by ProB made the work much more easy.

After a complete model of the secure partitioning microkernel, it was time to develop entirely the PIFP (section 5.3). Atelier B revealed some drawbacks in this part. Atelier B free license provides ComenC translator. This tool have some limitations. The normal development, like is usual to see, must to be changed because of the tool limitations. Some changes in the models were due to that limitations. One important aspect is that to provide full confidence using the B Method, is necessary to have all the components proved. In this case the objective was not achieved (section 5.5).

The next part is the integration of the PIFP with Prex (section 5.4). The difficulty in this part was on the side of Prex. A complete understanding of the kernel was necessary to introduce there a flow policy. In the end, the preliminary tests reveled good results.

Chapter 6

Conclusion

A formal model of the secure partitioning microkernel has been presented, along with several aspects of its internal structure and functioning. It can be animated to further check for correctness. A complete development for part of the microkernel (Partition Information Flow Policy) and the integration with Prex was also presented. Prex now have a flow policy integrated. Proof of consistency, for this part, has not been completed.

6.1 Contribute

This work was a collaboration between Critical Software and the University of Beira Interior. The process of knowledge transfer plays an important role to Critical Software successful growing. It was one objective to add know how to the company to embrace future projects involving the use of formal methods. This particular project, using the B Method, gives internal know how in the use of formal methods (in this case, the B Method) to the company.

Other contribution was the extension of Prex to have a flow policy. Prex is an open source microkernel with a very active work group. It is frequent to find new features in Prex. Until the day of the this thesis, no flow policy was joined to Prex. In the future it will be publish the work developed during this thesis in the Prex site.

The publication of this work in some conferences are also planned. However is necessary to complete the part of the proof consistency.

6.2 Challenge

In this thesis various subjects were addresses. Beside the use of the B Method, concepts related to microkernels were completely unknown. Some of the time spend on this thesis was on the discover of techniques related to the development of microkernels. It is important to say that, Critical Software played an important role in this part. Their help in the understanding of microkernels and experience was crucial to the success of this project.

Some work related to the development of microkernels using the B Method were available. Although, the verification of such microkernels was always addressed to the verification of the API. Using the B Method it is possible to achieve the complete development of the secure partitioning microkernel. However, the use of different formal methods, according with the specific part to be developed, can become the work easier. An example of this is time properties,

B does not deal well with this properties and they have an important place in the microkernel requirements.

The challenge of developing this work inside a company like Critical Software was one of the best experiences. This type of projects involving researchers and companies are very profit and should become more frequent. The project offers a view of this, sometimes, very different worlds.

6.3 Future Work

A complete proof of consistency is need for the partition information flow policy. Beside of tests that were already realized, the best way to guarantee the absence of errors in the code is a project with 100% proved components. In this thesis this was one of the objectives, but it was not accomplished.

The tests realized over Prex with the flow policy integrated showed good results. Although it is necessary the use of other type of verification besides tests. One idea is perform a work like the one mentioned in [14]. Modeling Prex extended with the flow policy with some model checker and then verify that all is working fine.

The formal development of the whole microkernel is also a future work. The formal mode here presented can be refined and then try to reach a level where is possible to generate code. In my opinion, this is important to be done, however the use of different formal methods could make the work less painful.

By last, other future work is the testing of Prex and PIFP together with a already existent microkernel that implements secure partitioning. If we generate the same inputs to both microkernels and the outputs of both are the same, it is possible to say that Prex with PIFP are working correctly.

Bibliography

[1] Prex Documentation. [cited at p. 39]

[2] J-R Abrial. *Modeling in Event-B: System and Software Engineering*. Forthcoming. [cited at p. 28]

[3] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996. [cited at p. 9, 12, 15, 20]

[4] Didier Bert. Exemples de conduite de preuve interactive avec l'atelier b, version 3.5, 2000. [cited at p. 20]

[5] William R. Bevier and Texas At Austin. A verified operating system kernel. Technical report, University of Texas at Austin, 1987. [cited at p. 45]

[6] Michael Butler Carla Ferreira. *Practitioner's Handbook*. Methodologies and Technologies for Industrial Strength Systems Engineering (MATISSE), 2003. [cited at p. 9]

[7] Clearsy. Industrial use of the b, 2000. [cited at p. 4]

[8] Samuel Colin. *Contribution à l'intégration de temporalité au formalisme B : utilisation du calcul des durées en tant que sémantique pour B*. PhD thesis, Université de Valenciennes et du Hainaut-Cambrésis, October 2006. [cited at p. 62]

[9] Samuel Colin, Georges Mariano, and Vincent Poirriez. Duration calculus: a real-time semantic for B. In *First International Colloquium on Theoretical Aspects of Computing*. UNU-IIST, September 2004. Guiyang, China. [cited at p. 62]

[10] Iain D. Craig. *Formal Models of Operating System Kernels*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. [cited at p. 51]

[11] Iain D. Craig. *Formal Refinement for Operating System Kernels*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. [cited at p. 52]

[12] David Greve, Matthew Wilding, and W. Mark Van Eet. A separation kernel formal security policy. 2003. [cited at p. 48]

[13] Council Canada De, Thierry Cattel, and Thierry Cattel. National research conseil national. In *In 7th Internation Conference on Formal Description Techniques*, pages 35–51, 1994. [cited at p. 46]

[14] Gregory Duval and Jacques Julliand. Modeling and verification of the rubis microkernel with spin. In *In Proceedings of the First SPIN Workshop*, 1995. [cited at p. 46, 88]

[15] Kevin Elphinstone, Gerwin Klein, Philip Derrin, and Timothy Roscoe. Towards a practical, verified kernel. In *In 11th HotOS*, 2007. [cited at p. 50]

[16] Jose Miguel Faria. Formal Development of Solutions for Real-Time Operating System with TLA^+/TLC. Master's thesis, Universidade do Porto, 2008. [cited at p. 5, 47]

[17] Houda Fekih, Leila Jemni Ben Ayed, and Stephan Merz. Transformation of B specifications into UML class diagrams and state machines. In *21st Annual ACM Symposium on Applied Computing - SAC 2006*, volume 2, pages 1840–1844, Dijon, France, April 2006. [cited at p. 27]

[18] Steven D. Fraser, Frederick P. Brooks, Jr., Martin Fowler, Ricardo Lopez, Aki Namioka, Linda Northrop, David Lorge Parnas, and David Thomas. "no silver bullet" reloaded: retrospective on "essence and accidents of software engineering". In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 1026–1030, New York, NY, USA, 2007. ACM. [cited at p. 6]

[19] U.S. Government. U.s. government protection profile for separation kernels in environments requiring high robustness, 2004. [cited at p. 31, 48]

[20] Ahmed Hammad, Jacques Julliand, Hassan Mountassir, and Dieudonné Okalas-Ossami. Expression en B et raffinement des sytèmes réactifs temps réel. In *AFADL'2003*, pages 211–226, 2003. [cited at p. 62]

[21] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. pages 346–355, 2006. [cited at p. 48]

[22] Sarah Hoffmann, Germain Haugou, Sophie Gabriele, and Lilian Burdy. The B-method for the construction of microkernel-based systems. In *The 7th International B Conference*, pages 257–259, 2007. [cited at p. 5, 53]

[23] J.-R.Abrial. Guidelines to Formal System Studies. 2000. [cited at p. 3, 10, 12]

[24] Rafal Kolanski. A formal model of the L4 micro-kernel API using the B method. Number Technical Report 05-00029-1, 2005. [cited at p. 53]

[25] Kevin Lano. *The B Language and Method: A guide to Practical Formal Development*. Springer-Verlag London Ltd., 1996. [cited at p. 62]

[26] Gérard Le Lann. The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. (RR-3079), 1996. Projet REFLECS. [cited at p. 11]

[27] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003. [cited at p. 24]

[28] W. Martin, P. White, F. S. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 133, Washington, DC, USA, 2000. IEEE Computer Society. [cited at p. 50]

[29] Betrand Meyer. From process to product: Where is the software headed? *IEE E Computer*, 1995. [cited at p. 3]

[30] Stéphanie Motré and Corinne Téri. Using formal and semi-formal methods for a common criteria evaluation. In *Eurosmart*, Marseille (France), June 2000. [cited at p. 31]

[31] Antoine Requet. A B model for ensuring soundness of the Java card virtual machine. In *FMICS'2000*, Berlin, March 2000. [cited at p. 31]

[32] Rodin. Rodin Site. [cited at p. 27]

[33] John Rushby. Design and verification of secure systems. *ACM Operating Systems Review*, 15(5):12–21, dec 1981. [cited at p. 33, 45]

[34] Thierry Servat. BRAMA: A new graphic animation tool for B models. pages 274–276, 2007. [cited at p. 27]

[35] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science Series)*. Addison Wesley, 8 edition, June 2006. [cited at p. 13]

[36] ENST: LabSoc ClearSy ST Microelectronics, CEA. B4l4 project. [cited at p. 53]

[37] G. Pouzancre T. Lecomte, T. Servat. Formal methods in satefy critical railway systems. In *Conference SBMF*, 2007. [cited at p. 4]

[38] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, January 2006. [cited at p. 51]

[39] Jochen Liedtke Toward and Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39, 1996. [cited at p. 29]

[40] Harvey Tuch, Gerwin Klein, and Gernot Heiser. Os verification - now. In *In Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, 2005. [cited at p. 50]

[41] Patrick Tullmann, Jeff Turner, John Mccorquodale, Jay Lepreau, Ajay Chitturi, and Godmar Back. Formal methods: A practical tool for os implementors. 1997. [cited at p. 46]

Appendices

Appendix A

B components

A.1 Machine Ctx

MACHINE
Ctx
SETS
TASKS ;
MSG ;
THREADS ;
MODE = { noflow , read , write , readWrite }
CONCRETE_CONSTANTS
kernelTask ,
kernelThread ,
kernelTaskPeriod ,
nullMsg ,
maxPortSize ,
communicationTime ,
maxtime
PROPERTIES
card (TASKS) $\geq$ 1 $\wedge$
card (THREADS) $\geq$ 1 $\wedge$
card (MSG) $\geq$ 1 $\wedge$
kernelTask $\in$ TASKS $\wedge$
kernelThread $\in$ THREADS $\wedge$
kernelTaskPeriod $\in$ NAT1 $\wedge$
nullMsg $\in$ MSG $\wedge$
maxPortSize $\in$ NAT1 $\wedge$
communicationTime $\in$ NAT1 $\wedge$
maxtime $\in$ NAT $\wedge$
END

A.2 Machine CtxMemory

MACHINE
CtxMemory
SETS
STATE = { idle , normal , coldstart , warmstart }

CONCRETE_CONSTANTS
memorysize ,
kernelspace ,
kernelspacecommunication ,
userspace ,
pagesize ,
ADDRESS ,
PAGE ,
VPAGE ,
INDEX ,
PAGESCOMMUNICATION

PROPERTIES
memorysize ∈ NAT1 ∧
memorysize ≥ 3 ∧
pagesize ∈ NAT1 ∧
pagesize < memorysize ∧
kernelspace ∈ NAT1 ∧
kernelspacecommunication ∈ NAT1 ∧
userspace ∈ NAT1 ∧
kernelspace + kernelspacecommunication + userspace = memorysize ∧
userspace mod pagesize = 0 ∧
kernelspace mod pagesize = 0 ∧
kernelspacecommunication mod pagesize = 0 ∧

INDEX = 0 .. (pagesize − 1) ∧
ADDRESS = 0 .. (memorysize − 1) ∧
PAGE = 0 .. ((userspace / pagesize) + (kernelspacecommunication / pagesize) − 1) ∧
PAGESCOMMUNICATION = (userspace / pagesize) .. ((userspace / pagesize) +
(kernelspacecommunication / pagesize) − 1) ∧
VPAGE = 0 .. ((userspace / pagesize) − 1)

A.3 Machine MemoryManagement

MACHINE
MemoryManagement
SEES
CtxMemory ,
Ctx

ABSTRACT_VARIABLES
usedTks ,
tksPages ,
tksPeriod ,
tksDuration ,
tksState

INVARIANT

usedTks $\subseteq$ TASKS $-$ { kernelTask } $\wedge$
tksPages $\in$ VPAGE $\nrightarrow$ usedTks $\wedge$
tksPeriod $\in$ usedTks $\rightarrow$ NAT1 $\wedge$
tksDuration $\in$ usedTks $\rightarrow$ NAT1 $\wedge$
tksState $\in$ usedTks $\rightarrow$ STATE $\wedge$
ran (tksPages) = usedTks $\wedge$
dom (tksPeriod) = ran (tksPages) $\wedge$
dom (tksDuration) = ran (tksPages) $\wedge$
dom (tksState) = ran (tksPages) $\wedge$
$\forall$ (tt) . (tt $\in$ usedTks $\wedge$ tt $\in$ ran (tksPages) $\Rightarrow$ card (tksPages $\rhd$ { tt }) $\geq$ 4)

INITIALISATION

usedTks := $\varnothing$ ||
tksPages := $\varnothing$ ||
tksPeriod := $\varnothing$ ||
tksDuration := $\varnothing$ ||
tksState := $\varnothing$

OPERATIONS

createTaskM (pnumber , bounds , per , tm) =
PRE
pnumber $\in$ TASKS $\wedge$
pnumber /$\in$ usedTks $\wedge$
pnumber $\neq$ kernelTask $\wedge$
bounds $\in$ NAT1 $\wedge$
per $\in$ NAT1 $\wedge$
tm $\in$ NAT1 $\wedge$
bounds $\geq$ 4 $\wedge$
card (usedTks) + bounds $\leq$ card (TASKS)
THEN
usedTks := usedTks $\cup$ { pnumber } ||
tksPeriod (pnumber) := per ||
tksDuration (pnumber) := tm ||
tksState (pnumber) := coldstart ||
ANY vpgs **WHERE** vpgs $\subseteq$ VPAGE $\wedge$ card (vpgs) = bounds $\wedge$
$\forall$ (vp) . (vp $\in$ VPAGE $\wedge$ vp $\in$ vpgs $\Rightarrow$ vp /$\in$ dom (tksPages)) **THEN**
tksPages := tksPages $\cup$ vpgs $\times$ { pnumber }
END
END ;

eliminateTaskM (pnumber) =
PRE
pnumber $\in$ usedTks $\wedge$
pnumber $\neq$ kernelTask
THEN
usedTks := usedTks $-$ { pnumber } ||

```
        tksPeriod := { pnumber } ⩤ tksPeriod ||
        tksDuration := { pnumber } ⩤ tksDuration ||
        tksState := { pnumber } ⩤ tksState ||
        tksPages := tksPages ⩥ { pnumber }
    END ;

    setTaskMMode ( pnumber , mode ) =
    PRE
        pnumber ∈ usedTks ∧
        mode ∈ STATE ∧
        ( ( tksState ( pnumber ) = coldstart ) ⇒ ( mode = coldstart )
        ∨ ( mode = normal ) ∨ ( mode = idle ) ) ∧
        ( ( tksState ( pnumber ) = idle ) ⇒ ( mode = coldstart )
        ∨ ( mode = warmstart ) ) ∧
        ( ( tksState ( pnumber ) = normal ) ⇒ ( mode = coldstart )
        ∨ ( mode = warmstart ) ) ∧
        ( ( tksState ( pnumber ) = warmstart ) ⇒ ( mode = coldstart )
         ∨ ( mode = warmstart ) ∨ ( mode = idle ) ∨ ( mode = normal ) )
    THEN
        tksState ( pnumber ) := mode
    END ;

    val ⟵ getTaskMMode ( pnumber ) =
    PRE
        pnumber ∈ usedTks
    THEN
        val := tksState ( pnumber )
    END
END
```

A.4 Machine Clock

```
MACHINE
    Clock
SEES
    Ctx
ABSTRACT_VARIABLES
    now
INVARIANT
    now ∈ 0 .. maxtime
INITIALISATION
    now := 0
OPERATIONS

    reset =
    BEGIN
        now := 0
    END ;

    tick_u ( from , to ) =
    PRE
```

from ∈ NAT ∧
to ∈ NAT ∧
from < to ∧
now + to ≤ maxtime
THEN
now :∈ (now + from) .. (now + to)
END ;

tick (to) =
PRE
to ∈ NAT ∧
now + to ≤ maxtime
THEN
now := (now + to)
END ;

tt ⟵ current_time =
BEGIN
tt := now
END
END

A.5 Machine Messages

MACHINE
Messages
SEES
Ctx
ABSTRACT_VARIABLES
message ,
messageSource ,
messageDestination ,
messageSize
INVARIANT
message ⊆ MSG − {nullMsg}∧
messageSource ∈ message → TASKS ∧
messageDestination ∈ message → TASKS ∧
messageSize ∈ message → NAT1 ∧
dom (messageSource) = dom (messageDestination) ∧
dom (messageSource) = dom (messageSize) ∧
∀ (mm) . (mm ∈ message ∧ mm ∈ dom (messageSource)
∧ mm ∈ dom (messageDestination)
⇒ messageSource (mm) ≠ messageDestination (mm))

INITIALISATION
message := ∅ ||
messageSource := ∅ ||
messageDestination := ∅ ||
messageSize := ∅

OPERATIONS

addNewMessage (msg , taskOwner , taskDestiny , lenght) =
PRE
msg ∈ MSG ∧
msg ≠ nullMsg ∧
taskOwner ∈ TASKS ∧
taskDestiny ∈ TASKS ∧
taskOwner ≠ taskDestiny ∧
lenght ∈ NAT1 ∧
lenght < maxPortSize
THEN
message := message ∪ { msg } ‖
messageSource (msg) := taskOwner ‖
messageDestination (msg) := taskDestiny ‖
messageSize (msg) := lenght
END ;

removeMessage (msg) =
PRE
msg ∈ message ∧
msg ≠ nullMsg
THEN
message := message − { msg } ‖
messageSource :=
messageSource − { msg ↦ messageSource (msg) } ‖
messageDestination :=
messageDestination − { msg ↦ messageDestination (msg) } ‖
messageSize :=
messageSize − { msg ↦ messageSize (msg) }
END ;

msglgt ⟵ getMessageLenght (msg) =
PRE
msg ∈ message ∧
msg ≠ nullMsg
THEN
msglgt := messageSize (msg)
END ;

msgsrc ⟵ getMessageSource (msg) =
PRE
msg ∈ message ∧
msg ≠ nullMsg
THEN
msgsrc := messageSource (msg)
END ;

msgdest ⟵ getMessageDestionation (msg) =
PRE
msg ∈ message ∧
msg ≠ nullMsg

THEN
msgdest := messageDestination (msg)
END
END

A.6 Machine PoolForCommunication

MACHINE
PoolForCommunication
SEES
CtxMemory , Ctx

ABSTRACT_VARIABLES
usedPagesCommunication ,
pagesCommunicationMapping

INVARIANT
usedPagesCommunication ⊆ PAGESCOMMUNICATION ∧
pagesCommunicationMapping ∈
usedPagesCommunication → TASKS

INITIALISATION
usedPagesCommunication := ∅ ||
pagesCommunicationMapping := ∅

OPERATIONS

/∗Maps a new free page for a process to communicate∗/
mapNewFreePage (tt , fp) =
PRE
tt ∈ TASKS ∧
fp ∈ PAGESCOMMUNICATION ∧
fp /∈ usedPagesCommunication ∧
fp /∈ dom (pagesCommunicationMapping)
THEN
usedPagesCommunication := usedPagesCommunication ∪ { fp } ||
pagesCommunicationMapping (fp) := tt
END ;

/∗Map a used page to a task∗/
mapPage (pg , tt) =
PRE
pg ∈ usedPagesCommunication ∧
tt ∈ TASKS

THEN
pagesCommunicationMapping (pg) := tt
END ;

/∗unmap a page from a task∗/
unmapPage (pg) =

PRE
pg ∈ usedPagesCommunication ∧
pg ∈ dom (pagesCommunicationMapping)
THEN
usedPagesCommunication := usedPagesCommunication − { pg } ||
pagesCommunicationMapping :=
pagesCommunicationMapping − { pg ↦ pagesCommunicationMapping (pg) }
END
END

A.7 Machine KernelCommunication

MACHINE
KernelCommunication
SEES
Ctx , CtxMemory

INCLUDES
PoolForCommunication , Messages

ABSTRACT_VARIABLES
ports ,
portSender ,
portReceiver ,
portsSize ,
com

INVARIANT
ports ∈ usedPagesCommunication → iseq (message) ∧
portSender ∈ usedPagesCommunication → TASKS ∧
portReceiver ∈ usedPagesCommunication → TASKS ∧
dom (portSender) = dom (portReceiver) ∧
∀ (us1 , us2) . (us1 ∈ usedPagesCommunication
∧ us2 ∈ usedPagesCommunication ∧ us1 ∈ dom (portSender) ∧
us2 ∈ dom (portReceiver) ∧ us1 = us2
⇒ portSender (us1) ≠ portReceiver (us2)) ∧
portsSize ∈ usedPagesCommunication → NAT ∧
dom (portsSize) = dom (ports) ∧
∀ (us) . (us ∈ usedPagesCommunication
⇒ ∀ (mm) . (mm ∈ message ∧ mm ∈ ran (ports (us))
⇒ SIGMA (xx) . (
xx ∈ message ∧ xx ∈ ran (ports (us)) | messageSize (xx)) = portsSize (us))) ∧
∀ (up) . (up ∈ usedPagesCommunication ⇒
portsSize (up) ≤ maxPortSize) ∧
∀ (us) . (us ∈ usedPagesCommunication
⇒ ∀ (mm1 , mm2) . (mm1 ∈ message ∧ mm1 ∈ ran (ports (us))
∧ mm2 ∈ message ∧ mm2 ∈ ran (ports (us))
⇒
messageDestination (mm1) = messageDestination (mm2))) ∧
∀ (us) . (us ∈ usedPagesCommunication ⇒
∀ (mm1 , mm2) . (mm1 ∈ message ∧ mm1 ∈ ran (ports (us))

∧ mm2 ∈ message ∧ mm2 ∈ ran (ports (us))
⇒
messageSource (mm1) = messageSource (mm2))) ∧
com ∈ TASKS ↔ TASKS

INITIALISATION
ports := ∅ ||
portSender := ∅ ||
portReceiver := ∅ ||
portsSize := ∅ ||
com := ∅

OPERATIONS

sendMessage (t1 , t2 , sz) =
PRE
t1 ∈ TASKS ∧
t2 ∈ TASKS ∧
t1 ≠ t2 ∧
sz ∈ NAT1 ∧
∃ (up) . (up ∈ usedPagesCommunication
∧ portSender (up) = t1 ∧ portReceiver (up) = t2 ∧
portsSize (up) + sz ≤ maxPortSize
∧ pagesCommunicationMapping (up) = t1)
THEN
ANY newMessage , up **WHERE**
newMessage ∈ MSG ∧
newMessage /∈ message ∧
newMessage ≠ nullMsg ∧
up ∈ usedPagesCommunication ∧ portSender (up) = t1
∧ portReceiver (up) = t2
∧ portsSize (up) + sz ≤ maxPortSize
THEN
addNewMessage (newMessage , t1 , t2 , sz) ||
ports (up) := ports (up) ↩ newMessage ||
portsSize (up) := portsSize (up) + sz
END
END ;

sendNewMessage (t1 , t2 , sz) =
PRE
t1 ∈ TASKS ∧
t2 ∈ TASKS ∧
t1 ≠ t2 ∧
sz ∈ NAT1 ∧
sz ≤ maxPortSize ∧
(card (usedPagesCommunication) <
card (PAGESCOMMUNICATION)) ∧
(¬ (∃ (up) . (up ∈ usedPagesCommunication
∧ portSender (up) = t1 ∧ portReceiver (up) = t2 ∧
portsSize (up) + sz ≤ maxPortSize

```
    ∧ pagesCommunicationMapping ( up ) = t1 ) ) )
THEN
    ANY newMessage , up WHERE
        newMessage ∈ MSG ∧
        newMessage /∈ message ∧
        newMessage ≠ nullMsg ∧
        up ∈ PAGESCOMMUNICATION ∧
        up /∈ usedPagesCommunication ∧
        up /∈ dom ( pagesCommunicationMapping )
    THEN
        mapNewFreePage ( t1 , up ) ||
        addNewMessage ( newMessage , t1 , t2 , sz ) ||
        ports ( up ) := [] ↩ newMessage ||
        portSender ( up ) := t1 ||
        portReceiver ( up ) := t2 ||
        portsSize ( up ) := sz ||
        com := com ∪ { t1 ↦ t2 }
    END
END ;

res ⟵ receiveMessage ( tto , tfrom ) =
PRE
    tto ∈ TASKS ∧
    tfrom ∈ TASKS ∧
    tto ≠ tfrom ∧
    ∃ ( up ) . ( up ∈ usedPagesCommunication
    ∧ portSender ( up ) = tfrom
        ∧ portReceiver ( up ) = tto ∧ ports ( up ) ≠ []
        ∧ pagesCommunicationMapping ( up ) = tto
    )
THEN
    ANY msg , up WHERE
        msg ∈ message ∧
        messageSource ( msg ) = tfrom ∧
        messageDestination ( msg ) = tto ∧
        up ∈ usedPagesCommunication ∧
        pagesCommunicationMapping ( up ) = tto ∧
        portSender ( up ) = tfrom ∧
        portReceiver ( up ) = tto ∧
        ports ( up ) ≠ [] ∧
        msg = last ( ports ( up ) )
    THEN
        res := msg ||
        IF size ( ports ( up ) ) = 1 THEN ports ( up ) := []
        ELSE ports ( up ) := front ( ports ( up ) )
         END ||
        portsSize ( up ) := portsSize ( up ) − messageSize ( msg ) ||
        removeMessage ( msg )
    END
END ;
```

```
    processTaskToReceive ( tt ) =
    PRE
        tt ∈ TASKS
    THEN
        ANY pg WHERE
            pg ∈ usedPagesCommunication ∧
            portReceiver ( pg ) = tt ∧
            pagesCommunicationMapping ( pg ) ≠ tt
        THEN
            mapPage ( pg , tt )
        END
    END ;

    cleanEmptyPorts =
    PRE
        ∃ ( up ) . ( up ∈ usedPagesCommunication ∧ ports ( up ) = [] )
    THEN
        ANY up WHERE
            up ∈ usedPagesCommunication ∧
            ports ( up ) = []
        THEN
            portSender := portSender − { up ↦ portSender ( up ) } ||
            portReceiver := portReceiver − { up ↦ portReceiver ( up ) } ||
            ports := ports − { up ↦ ports ( up ) } ||
            portsSize := portsSize − { up ↦ portsSize ( up ) } ||
            unmapPage ( up )
        END
    END
END
```

A.8 Machine Scheduling_FIFO

```
MACHINE
    Scheduling_FIFO ( CC )

ABSTRACT_VARIABLES
    fifo
INVARIANT
    fifo ∈ iseq ( CC )
INITIALISATION
    fifo := []

OPERATIONS

    add ( cc ) =
    PRE
        cc ∈ CC ∧
        cc /∈ ran ( fifo )
    THEN
        fifo := fifo ↩ cc
    END ;
```

```
    res ⟵ isEmpty =
    BEGIN
        IF  fifo  = [] THEN res := TRUE ELSE res := FALSE END
    END ;

    res ⟵ topFifo =
    PRE
        fifo  ≠ []
    THEN
        res := first  (  fifo  )
    END ;

    remove =
    PRE
        fifo  ≠ []
    THEN
        fifo  := fifo \|/ 1
    END ;

    change =
    BEGIN
        IF  fifo  = [] THEN fifo := [] ELSE
            IF size  (  fifo  ) = 1 THEN fifo := fifo ELSE
                ANY newFifo WHERE
                    newFifo ∈ iseq  (  CC  ) ∧
                    newFifo = tail  (  fifo  ) ↩ first  (  fifo  )
                THEN
                    fifo  := newFifo
                END
            END
        END
    END

END
```

A.9 Machine InterfaceMachine

```
MACHINE
    InterfaceMachine
SEES
    Ctx ,
    CtxMemory

INCLUDES
    Scheduling_FIFO ( TASKS ) ,
    Clock ,
    KernelCommunication ,
    MemoryManagement ,
    FlowPolicy
```

ABSTRACT_VARIABLES

elapseTime ,
actualTask ,
actualThread ,
tperiod ,
tasks ,
threads ,
tthreads ,
reSched ,
configure ,
kernelRemainTime,
per

INVARIANT

configure $\in \mathbb{B} \wedge$
reSched $\in \mathbb{B} \wedge$
elapseTime $\in$ NAT $\wedge$
tasks $\subseteq$ TASKS $\wedge$
threads $\subseteq$ THREADS $\wedge$
actualTask $\in$ tasks $\wedge$
actualThread $\in$ threads $\wedge$
tperiod $\in$ tasks $\rightarrow$ NAT1 $\wedge$
tthreads $\in$ threads $\rightarrow$ tasks $\wedge$
tthreads (actualThread) = actualTask $\wedge$
kernelRemainTime $\in$ NAT $\wedge$
per $\in$ NAT

INITIALISATION

configure := TRUE ||
reSched := FALSE ||
elapseTime := kernelTaskPeriod ||
tasks := { kernelTask } ||
threads := { kernelThread } ||
actualTask := kernelTask ||
actualThread := kernelThread ||
tperiod := { kernelTask $\mapsto$ kernelTaskPeriod } ||
tthreads := { kernelThread $\mapsto$ kernelTask } ||
kernelRemainTime := 0 ||
per := 0

OPERATIONS

createTask (tt , time, bb) =
PRE
tt $\in$ TASKS $\wedge$
tt $/\in$ tasks $\wedge$
time $\in$ NAT1 $\wedge$
configure = TRUE $\wedge$
bb $\in$ NAT $\wedge$
bb $\geq$ 4

THEN
createTaskM (tt , bb , per , time)
tasks := tasks ∪ { tt } ||
tperiod (tt) := time ||
add (tt) ||
per := per + 1
END ;

addThreadToTask (tt , th) =
PRE
tt ∈ tasks ∧
th ∈ THREADS ∧
th /∈ threads ∧
tt ≠ kernelTask ∧
configure = TRUE
THEN
threads := threads ∪ { th } ||
tthreads (th) := tt
END ;

endConf =
PRE
configure = TRUE
THEN
configure := FALSE ||
generateConf
END ;

changeToKernelTask =
PRE
elapseTime = 0 ∧
reSched = FALSE ∧
configure = FALSE ∧
actualTask ≠ kernelTask ∧
actualThread ≠ kernelThread ∧
kernelRemainTime = 0
THEN
change ||
setTaskMMode(actualTask,)
actualTask := kernelTask ||
actualThread := kernelThread ||
IF ∃ (up) . (up ∈ usedPagesCommunication ∧ ports (up) = []) **THEN**
cleanEmptyPorts
ELSE
skip
END ||
kernelRemainTime := kernelTaskPeriod
END ;

res ⟵ kernelTaskExecution =
PRE

```
    actualTask = kernelTask ∧
    actualThread = kernelThread ∧
    configure = FALSE ∧
    kernelRemainTime ≠ 0
THEN
    IF ∃ ( pg ) . ( pg ∈ usedPagesCommunication ∧ portReceiver ( pg )
    = first ( fifo ) ∧ pagesCommunicationMapping ( pg ) ≠ first ( fifo ) )
    THEN
        processTaskToReceive ( first ( fifo ) ) ||
        res := first ( fifo )
    ELSE
        res := first ( fifo )
    END ||
    ANY t1 WHERE
        t1 ∈ NAT1 ∧
        t1 ≤ kernelRemainTime
    THEN
        tick ( t1 ) ||
        kernelRemainTime := kernelRemainTime − t1
    END
END ;

kernelTaskEndExecution =
PRE
    actualTask = kernelTask ∧
    actualThread = kernelThread ∧
    configure = FALSE ∧
    kernelRemainTime = 0
THEN
    actualTask := first ( fifo ) ||
    elapseTime := tperiod ( first ( fifo ) ) ||
    ANY th WHERE
        th ∈ threads ∧
        tthreads ( th ) = first ( fifo )
    THEN
        actualThread := th
    END ||
    reSched := FALSE
END ;

run =
PRE
    elapseTime ≠ 0 ∧
    reSched = FALSE ∧
    configure = FALSE ∧
    actualTask ≠ kernelTask ∧
    actualThread ≠ kernelThread
THEN
    ANY t1 WHERE
        t1 ∈ NAT1 ∧
        t1 ≤ elapseTime
```

```
    THEN
        tick ( t1 ) ||
        elapseTime := elapseTime − t1 ||
        ANY th WHERE
            th ∈ threads ∧
            tthreads ( th ) = first ( fifo )
        THEN
            actualThread := th
        END
    END
END ;

res ⟵ sendMessageCurrentTask ( t1 , t2 , sz ) =
PRE
    reSched = FALSE ∧
    configure = FALSE ∧
    actualTask ≠ kernelTask ∧
    actualThread ≠ kernelThread ∧
    t1 = actualTask ∧
    t2 ∈ TASKS ∧
    t2 ≠ kernelTask ∧
    t2 ≠ t1 ∧
    sz ∈ NAT1 ∧
    sz ≤ maxPortSize ∧
    elapseTime ≥ communicationTime ∧
    ( ∃ ( up ) . ( up ∈ usedPagesCommunication ∧ portSender ( up ) = t1 ∧
    portReceiver ( up ) = t2 ∧ portsSize ( up ) + sz ≤ maxPortSize ∧
    pagesCommunicationMapping ( up ) = t1 )
    ∨ ( card ( usedPagesCommunication ) < card ( PAGESCOMMUNICATION ) )
     ∧ ( ¬ ( ∃ ( up ) . ( up ∈ usedPagesCommunication ∧ portSender ( up ) = t1 ∧
       portReceiver ( up ) = t2 ∧ portsSize ( up ) + sz ≤ maxPortSize ∧
        pagesCommunicationMapping ( up ) = t1 ) ) ) ∧
    ( flowPolicy ( t1 , t2 ) = write ∨ flowPolicy ( t1 , t2 ) = readWrite )
THEN

    IF ( card ( usedPagesCommunication ) < card ( PAGESCOMMUNICATION ) ) ∧
     ( ¬ ( ∃ ( up ) . ( up ∈ usedPagesCommunication ∧
      portSender ( up ) = t1 ∧ portReceiver ( up ) = t2 ∧
       portsSize ( up ) + sz ≤ maxPortSize ∧
        pagesCommunicationMapping ( up ) = t1 ) ) )
    THEN sendNewMessage ( t1 , t2 , sz ) || res := TRUE
    ELSE IF ∃ ( up ) . ( up ∈ usedPagesCommunication ∧
     portSender ( up ) = t1 ∧ portReceiver ( up ) = t2 ∧
      portsSize ( up ) + sz ≤ maxPortSize ∧
       pagesCommunicationMapping ( up ) = t1 )
       THEN sendMessage ( t1 , t2 , sz ) || res := TRUE
       ELSE res := FALSE
       END
    END ||
    tick ( communicationTime ) ||
    elapseTime := elapseTime − communicationTime
```

END ;

res ⟵ receiveMessageCurrentTask (t1 , t2) =
PRE
reSched = FALSE ∧
configure = FALSE ∧
actualTask ≠ kernelTask ∧
actualThread ≠ kernelThread ∧
t1 = actualTask ∧
t2 ∈ TASKS ∧
t2 ≠ kernelTask ∧
t2 ≠ t1 ∧
elapseTime ≥ communicationTime ∧
(flowPolicy (t1 , t2) = read ∨ flowPolicy (t1 , t2) = readWrite)
THEN
IF (∃ (up) . (up ∈ usedPagesCommunication ∧
portSender (up) = t2 ∧ portReceiver (up) = t1
∧ ports (up) ≠ [] ∧ pagesCommunicationMapping (up) = t1))
THEN
res ⟵ receiveMessage (t1 , t2)
ELSE
res := nullMsg
END ‖
tick (communicationTime) ‖
elapseTime := elapseTime − communicationTime
END
END

A.10 Machine FlowPolicy

MACHINE
FlowPolicy
SEES
Ctx

ABSTRACT_VARIABLES
flowPolicy ,
conf

INVARIANT
flowPolicy ∈ ((0 .. nmaxTasks) × (0 .. nmaxTasks)) → MODE ∧
∀ (t1 , t2) . (t1 ∈ (0 .. nmaxTasks) ∧ t2 ∈ (0 .. nmaxTasks)
∧ (t1 , t2) ∈ dom (flowPolicy) ∧ t1 = t2 ⇒ flowPolicy (t1 , t2) = noflow) ∧
conf ∈ 𝔹

INITIALISATION
flowPolicy := (0 .. nmaxTasks) × (0 .. nmaxTasks) × { noflow } ‖
conf := FALSE

OPERATIONS

generateConf =
PRE
conf = FALSE
THEN
ANY newConf **WHERE**
newConf ∈ ((0 .. nmaxTasks) × (0 .. nmaxTasks)) → MODE ∧
∀ (t1 , t2) . (t1 ∈ (0 .. nmaxTasks) ∧ t2 ∈ (0 .. nmaxTasks)
∧ (t1 , t2) ∈ dom (newConf) ∧ t1 = t2 ⇒ newConf (t1 , t2) = noflow)
THEN
flowPolicy := newConf
END ||
conf :∈ 𝔹
END ;

res ⟵ lookup (t1 , t2) =
PRE
t1 ∈ (0 .. nmaxTasks) ∧
t2 ∈ (0 .. nmaxTasks)
THEN
IF flowPolicy (t1 , t2) ≠ noflow **THEN** res := TRUE **ELSE** res := FALSE **END**
END ;

res ⟵ send (t1 , t2) =
PRE
t1 ∈ (0 .. nmaxTasks) ∧
t2 ∈ (0 .. nmaxTasks) ∧
flowPolicy (t1 , t2) ≠ noflow
THEN
IF flowPolicy (t1 , t2) = write ∨ flowPolicy (t1 , t2) = readWrite
THEN res := TRUE
ELSE res := FALSE **END**
END ;

res ⟵ receive (t1 , t2) =
PRE
t1 ∈ (0 .. nmaxTasks) ∧
t2 ∈ (0 .. nmaxTasks) ∧
flowPolicy (t1 , t2) ≠ noflow
THEN
IF flowPolicy (t1 , t2) = read ∨ flowPolicy (t1 , t2) = readWrite
THEN res := TRUE
ELSE res := FALSE **END**
END

END

A.11 Machine Matrix

MACHINE

Matrix
SEES
Ctx
ABSTRACT_VARIABLES
matrix
INVARIANT
matrix ∈ ((0 .. nmaxTasks) × (0 .. nmaxTasks)) → MODE
INITIALISATION
matrix := (0 .. nmaxTasks) × (0 .. nmaxTasks) × { noflow }
OPERATIONS

add (t1 , t2 , mm) =
PRE
t1 ∈ 0 .. nmaxTasks ∧
t2 ∈ 0 .. nmaxTasks ∧
mm ∈ MODE
THEN
matrix (t1 , t2) := mm
END ;

val ⟵ get (t1 , t2) =
PRE
t1 ∈ 0 .. nmaxTasks ∧
t2 ∈ 0 .. nmaxTasks
THEN
val := matrix (t1 , t2)
END
END

A.12 Machine BASIC_IO

MACHINE
BASIC_IO
OPERATIONS
bb ⟵ INTERVAL_READ (mm , nn) = **PRE**
nn ∈ NAT ∧
mm ∈ NAT ∧
mm ≤ nn
THEN
bb :∈ mm .. nn
END ;

INT_WRITE (vv) = **PRE**
vv ∈ NAT
THEN
skip
END ;

bb ⟵ 𝔹_READ = **BEGIN**
bb :∈ 𝔹
END ;

```
  𝔹_WRITE ( bb ) = PRE
    bb ∈ 𝔹
  THEN
    skip
  END ;

  cc ⟵ CHAR_READ = BEGIN
    cc :∈ 0 .. 255
  END ;

  CHAR_WRITE ( cc ) = PRE
    cc ∈ 0 .. 255
  THEN
    skip
  END ;

  STRING_WRITE ( ss ) = PRE
    ss ∈ STRING
  THEN
    skip
  END

END
```

A.13 Machine FlowPolicy_Imp

```
IMPLEMENTATION
    FlowPolicy_Imp
REFINES
    FlowPolicy
IMPORTS
    Matrix ,
    BASIC_IO
SEES
    Ctx
CONCRETE_VARIABLES
    confC
INVARIANT
    confC ∈ 𝔹 ∧
    confC = conf ∧
    flowPolicy = matrix
INITIALISATION
    confC := FALSE

OPERATIONS
    generateConf =
    BEGIN
        VAR cont1 , cont2 , ch IN
            cont1 := 0 ;
            cont2 := 0 ;
```

```
        IF confC = TRUE THEN
            STRING_WRITE ( "\tCONFIGURATION ALREADY DONE\n" )
        ELSE
            WHILE cont1 ≤ nmaxTasks DO
                WHILE cont2 ≤ nmaxTasks DO
                    IF cont1 ≠ cont2 THEN
                        STRING_WRITE ( "\tSET MODE FOR∈\n" ) ;
                        INT_WRITE ( cont1 ) ;
                        STRING_WRITE ( "\t→\t" ) ;
                        INT_WRITE ( cont1 ) ;
                        STRING_WRITE ( "\n" ) ;
                        STRING_WRITE ( "\t 1 − NO FLOW \n" ) ;
                        STRING_WRITE ( "\t 2 − READ \n" ) ;
                        STRING_WRITE ( "\t 3 − WRITE \n" ) ;
                        STRING_WRITE ( "\t 4 − READ AND WRITE \n" ) ;
                        ch ⟵ CHAR_READ ;
                        CASE ch OF
                            EITHER 1 THEN add ( cont1 , cont2 , noflow )
                            OR 2 THEN add ( cont1 , cont2 , read )
                            OR 3 THEN add ( cont1 , cont2 , write )
                            OR 4 THEN add ( cont1 , cont2 , readWrite )
                            ELSE add ( cont1 , cont2 , noflow ) END
                        END
                    ELSE
                        add ( cont1 , cont2 , noflow )
                    END ;
                    cont2 := cont2 + 1
                INVARIANT
                    cont2 ∈ 0 .. nmaxTasks + 1
                VARIANT
                    nmaxTasks + 1 − cont2
                END ;
                cont1 := cont1 + 1
            INVARIANT
                cont1 ∈ 0 .. nmaxTasks + 1
            VARIANT
                nmaxTasks + 1 − cont1
            END
        END ;
        confC := TRUE
    END
END ;

res ⟵ lookup ( t1 , t2 ) =
BEGIN
    VAR aux IN
        IF t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0 ∧ confC = TRUE THEN
            aux ⟵ get ( t1 , t2 ) ;
            IF aux = noflow THEN res := FALSE ELSE res := TRUE END
        ELSE
            STRING_WRITE ( "\tERROR ↪ TASKS DO NOT EXIST IN THE SYSTEM\n" ) ;
```

```
            res := FALSE
        END
    END
END ;

res ⟵ send ( t1 , t2 ) =
BEGIN
    VAR aux IN
        IF t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0 ∧ confC = TRUE THEN
        aux ⟵ get ( t1 , t2 ) ;
            IF aux = write ∨ aux = readWrite THEN res := TRUE ELSE res := FALSE END
        ELSE
            STRING_WRITE ( "\tERROR ↪ TASKS DO NOT EXIST IN THE SYSTEM\n" ) ;
            res := FALSE
        END
    END
END ;

res ⟵ receive ( t1 , t2 ) =
BEGIN
    VAR aux IN
        IF t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0 ∧ confC = TRUE THEN
            aux ⟵ get ( t1 , t2 ) ;
            IF aux = read ∨ aux = readWrite THEN res := TRUE ELSE res := FALSE END
        ELSE
            STRING_WRITE ( "\tERROR ↪ TASKS DO NOT EXIST IN THE SYSTEM\n" ) ;
            res := FALSE

        END
    END
END

END
```

A.14 Machine Matrix_Imp

```
IMPLEMENTATION
  Matrix_Imp
REFINES
  Matrix
SEES
  Ctx
CONCRETE_VARIABLES
  matrixC
INVARIANT
  matrixC ∈ ( ( 0 .. nmaxTasks ) × ( 0 .. nmaxTasks ) ) → MODE ∧
  matrixC = matrix
INITIALISATION
  matrixC := ( 0 .. nmaxTasks ) × ( 0 .. nmaxTasks ) × { noflow }
OPERATIONS
```

```
add ( t1 , t2 , mm ) =
BEGIN
    IF ( mm = noflow ∨ mm = read ∨ mm = write ∨ mm = readWrite )
    ∧ t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0 THEN
        matrixC ( t1 , t2 ) := mm
    ELSE
        skip
    END
END ;

val ⟵ get ( t1 , t2 ) =
BEGIN
    IF t1 ≤ nmaxTasks ∧ t1 ≥ 0 ∧ t2 ≤ nmaxTasks ∧ t2 ≥ 0 THEN
        val := matrixC ( t1 , t2 )
    ELSE
        val := matrixC ( 0 , 0 )
    END
END
```

Appendix B

Example Configuration File

B.1 Config File

```
start
        flow<<1,2,read>>;
        flow<<2,3,write>>;
        flow<<2,1,noFlow>>
end
```

B.2 Result file

```
#include <prex/prex.h>
#include <stdio.h>
#include <string.h>

 /* 1 = No Flow
 2 - Read
 3 - Write
 4 - Read and Write
 */

void flowPolicyCreation ()
{
 addFlowPolicy(1,2,2);
 addFlowPolicy(2,3,3);
 addFlowPolicy(2,1,1);

 }
```