On the Application of Model Checking Techniques to Real-Time Hypervisors in the Context of Integrated Modular Avionics Systems

Inês Jorge Pedro
inesjpedro@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2017

Abstract

The safety-critical nature of most avionics systems, such as aircraft and spacecraft, require a scrutinizing examination of their correctness before they are released for use. In order to study the correctness of such systems, more robust techniques other than testing are required. Formal methods such as model checking, an automatic verification technique for finite state concurrent systems, arise as an alternative solution for providing verification of system properties. The goal of our work is to use model checking systems, such as Spin and TLC, for verifying parts of GMV's hard real-time hypervisor. Concurrency problems of this real-time distributed system, namely time partitioning, interprocess communication with shared memory and memory models, are abstracted and model checked throughout this dissertation. The major contribution of this work is the discovering of an implementation error, by the TLC model checker, that caused concurrent processes to have access to the same shared data simultaneously and, consequently, obtaining corrupted information. Moreover, we also present a solution that assures mutual exclusion between a single writer and multiple readers, using the Non-Blocking Write protocol [4], in which additional information of the processes' periods, known a priori, allows us to eliminate "read-and-check" loops from the original algorithm.
Keywords: model checking, formal verification, real-time systems, distributed systems

1. Introduction

Avionics, whose term originates from the composition of the words "aviation electronics", are the electronic systems used on aircraft, artificial satellites, and spacecraft. Due to both safety and financial reasons, it is crucial for the companies that produce avionics systems that such systems are bulletproof.

Examples of disastrous software failures are NASA's Mars Climate Orbiter (1998), whose trajectory missed the landing on Mars, and the radiation therapy machine Therac-25 (1982), whose extreme doses of radiation caused the death or serious injuries of several patients. Software errors like these could have been prevented if a more effective correctness analysis had been performed. Typical software verification techniques like peer reviewing and testing cannot prove the absence of errors, given the system's complexity. An alternative approach is using formal methods, which is a mathematically based technique, for proving the correctness of soft- or hardware systems. In particular, model checking is a formal method that, given a model of a system and a correctness property, exhaustively verifies if the model meets the respective property.

An avionics system contains different functionalities such as navigation, autopilot and communication. In order to study the safety of the system, it is easier to analyze each functionality, in terms of safety, individually. One way of imposing this independence between functionalities is to distribute them geographically in different processors, thus allocating dedicated resources for each functionality. This is called the federated architecture and has the disadvantage of requiring at least one hardware component for each function. On the other hand, we can enforce independence of functions through software without the extensive use of resources as in the latter case, following the Integrated Modular Avionics (IMA) architecture. These two approaches are represented in Figure 1.

What we want to guarantee is that both architectures satisfy the same requirements. The study of this "safety equivalence" motivates the use of formal methods in avionics systems in order to verify if the model of the IMA architecture satisfies the

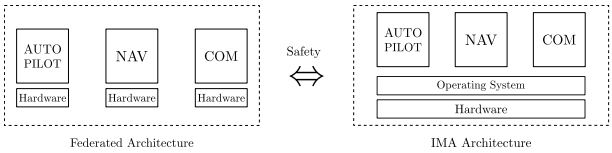

Figure 1: Safety equivalence between federated and IMA architectures.

same correctness properties of the federated one.

The goal of this work, in collaboration with GMV, is to use model checking techniques to study correctness properties of parts of XKY's hard real-time hypervisor, Partition Management Kernel (PMK), namely scheduling analysis of time partitioning, interprocess communication with shared memory and memory models.

2. Model Checking and Real-Time Distributed Systems - Background

Our research work concerns two topics of the computer science field: distributed computing and model checking. In particular, we will apply model checking techniques to a real-time distributed case study system.

2.1. Model Checking Overview

Contrarily to sequential programs, that have a single thread of control and in which instructions are executed in a consecutive and ordered fashion, *concurrent* programs contain a thread of control for each process and, hence, the order in which processes execute is non-deterministic. There are two forms of concurrency, depending on the allocation of processes. Two processes may execute concurrently in an *interleaved* fashion if they belong to the same processor, in which a process may interrupt another, but only one can execute at a time; or they can run in *parallel* (they may execute at exactly the same time), if allocated in different processors.

The non-determinism inherent to concurrent systems exponentially increases the complexity of their correctness analysis. One way of modeling concurrent systems is through *transition systems*, which are very useful in describing their behavior. These are directed graphs, whose nodes are the system states and whose edges represent system transitions. Model checking verifies if a given property holds in a system by analyzing all possible states in the corresponding transition system.

We will only verify safety *linear-time properties*, formalized with Linear-Time Logic (LTL), which informally reflect the idea that "nothing bad ever happens".

In order to perform model checking analysis, we will use two different model checkers, Spin [3] in Section 3 and TLC [5, 7] in Sections 4 and 5, whose respective modeling languages are Promela and TLA+.

2.2. Real-Time Distributed Systems

The term *distributed* was originally used to designate algorithms that ran in several processors physically distributed [8]. Currently, it also refers to shared memory multiprocessors algorithms. Distributed systems consist of multiple processes, each of which with an independent thread of control, and, therefore, such systems are also concurrent. Despite process independence, they are all working towards a common goal and, consequently, there must be an interprocess communication mechanism to assure their interaction.

Besides distributed, our system is also *real-time*. The correctness of a real-time system depends not only on its functional behavior but also on the time of response. Such systems are expected to respond to requests from their environment with specific time constraints, called deadlines. A system is called *hard* real-time if there is at least one deadline that suffers severe consequences if missed. PMK is a hard real-time system and, thus, it is crucial that time partitioning is enforced in order to avoid missed deadlines.

3. Scheduling Analysis

This section reports the modeling and verification of time partitioning in PMK's scheduling, with an IMA architecture.

3.1. System Description

Each functionality represented in Figure 1 is called a *partition*, and the workset for this section consists of a set of partitions sharing the same processor, whereby a static schedule must be provided to assure that each partition has access to the same computing resource. PMK supports functions, i.e., partitions, with different timeliness requirements and all should complete in time (this is called timeliness property). This is achieved through time partitioning within a partition, where a function is divided into several software components, that we will call *tasks*, sharing the time budgeted for the respective function, in which only one task can run at a time.

In this way, time partitioning must be performed at two levels: between partitions, through what we call the higher-level schedule, and among each partition's set of tasks, determined by a lower-level schedule (one for each partition).

Partition scheduling is static, non-preemptive and is defined through time windows. A time window is characterized by the starting moment, s, and the duration, d, meaning that the respective partition will run in the interval $[s, s+d]$. A partition can have multiple time windows but we can only define a finite number of them. Since time increases indef-

initely, the schedule must be cyclic, which leads us to introduce the Major Time Frame (MTF), which is the moment that marks the scheduling cycle: the schedule repeats itself every time the clock reaches a multiple of MTF. So, in practice, there can be infinite time windows, but we only need to specify the ones in the interval $[0, MTF]$.

Tasks, on the other hand, are scheduled according to their timeliness, which is translated in priorities. A more critical task will be given a higher priority, and all priorities are defined off-line and are not subject to changes during runtime. Contrarily to the higher-level schedule, tasks' schedule is preemptive, in the sense that a higher priority task can interrupt an executing task with a lower priority. This lower priority task is resumed once the higher priority task completes its execution. Each task is characterized by a period, p, and an execution time, e, meaning that the task must run for e time units in each p units. Additionally, each task is also associated with a priority, which will be represented by an identifier, ID, inversely proportional to its priority, inspired in [2]. In other words, the higher the priority, the lower the identifier.

Each partition's lower-level schedule must be combined with the corresponding partition's time partitioning. Tasks have now not only running time restrictions related to their attributes (period, execution time and priority) but also to their partition's time windows. To illustrate this, consider two tasks, T_0 and T_1, defined by $p_0 = 10$, $e_0 = 3$, $ID = 0$ and $p_1 = 5$, $e_1 = 1$, $ID = 1$. Assuming that these tasks belong to partition P_0, with time windows $[1, 5]$ and $[8, 10]$, then we would obtain the schedule represented in Figure 2.

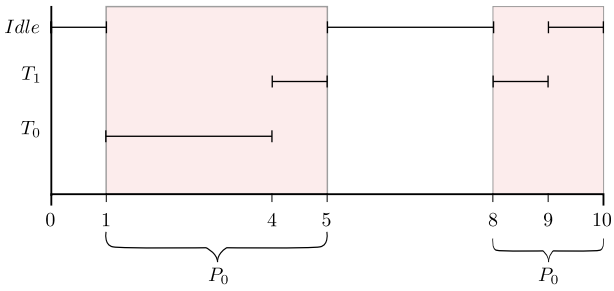

Figure 2: Schedule of tasks T_0 and T_1 of partition P_0.

When a partition's time window ends and another partition starts executing, there is an often called *context switch*. The same happens within a partition: when an executing task gives place to another task, there is also a context switch in the middle of this transition. During the switching mechanism, the state of the interrupted process (either a task or a partition) must be saved, so it can be loaded when it is resumed, while the state of the process that is ready to execute is loaded. This operation takes a given amount of time and may, eventually, have an impact on the schedule, since it delays the starting moment of process execution.

3.2. System Properties

We intend to verify if time partitioning of the system assures system safety, i.e., if both schedules are *feasible*. A schedule is feasible if all the respective processes meet their deadlines. Thus, we are interested in verifying when the system responds, rather than what it does, whereby the properties we want to verify are real-time.

More specifically, we want to ensure that every task in every period has access to its full budget (execution time). As a consequence, every time the clock meets the task's deadline, it must have already executed for the respective amount of time. Denoting by d_i and e_i the deadline and execution time, respectively, of task i, the feasibility of a schedule can be described by the following LTL property, for every task i:

$$P_1: \quad \Box\ ((clock \geq d_i) \Rightarrow (executed\_time = e_i)). \tag{1}$$

The same property can be adapted to the higher level schedule: every time the clock reaches the final instant of the partition's current time window, the partition must have already stopped running.

A schedule may not be feasible, consequently violating property P_1, when

- the design of the schedule is not feasible a priori, i.e., not considering context switches. For example, the schedule may not be feasible because the CPU's usage is higher than 100%;
- the schedule is feasible but, when considering context switching, it consumes a substantial amount of time of a partition so that some tasks do not have access to their full budget.

The other time property that must be fulfilled is related to the partitions' static scheduling. What we want to verify here is that partition context switches never reach the duration of each time window, otherwise no progress would be made, since the partition would not have access to the CPU during the corresponding time window. Denoting by *window* a time window of a partition, and by $PART\_CS$ the partition context switch duration, we can translate this property in the following LTL formula

$$P_2: \quad \Box\ (PART\_CS < window.duration). \tag{2}$$

3.3. Modeling Time Partitioning

Since our system has a significant complexity associated, we decided to follow a bottom-up approach, where we start by modeling the partition and task

schedules individually, and then combine them. Afterwards, we add another layer of complexity, by inserting context switches (at two levels, one at each step), obtaining the complete version of the model. In each version of the model, we perform verification of the system properties, to avoid building a more complex model upon a defective one.

We selected PROMELA as the modeling language for this system given that the higher and lower-level schedules were highly inspired on Ben-Ari's work [2], which uses PROMELA. Moreover, this language's *timeout* predefined variable, that contains the value true if and only if all system's processes are blocked, allowed us to directly model the idle process, which runs when no other process is available.

Time was discretized in our abstraction and the time unit is the finer grain of the schedule's timings, in order to reduce the model's complexity. In this way, time will be modeled as a discrete variable, *clock*, initialized to 0, and is incremented through clock ticks. Since there is always an active task, we will consider that processes are responsible for advancing time.

The modeling of tasks and partitions was highly inspired on the work of Ben-Ari [2]. Property P_1, in 1, is verified in the model by means of a watchdog [2] that, whenever the clock passes the process's deadline, checks if the process has executed, with an assert statement.

Relatively to the context switches, we introduced a process responsible for simulating the duration of switches between partitions, and, for each partition, a process was added in order to increment the clock every time the respective partition swaps active tasks. Property P_2, in 2, concerning the duration of partition context switching, is verified after each such switch is performed, again with an assert statement.

Let us now analyze a concrete example to demonstrate how our model simulates time partitioning. Suppose we have two partitions, P_0 and P_1, where P_0 has a single task, T_0, and runs in the time windows $[0, 9]$ and $[40, 52]$, while P_1 contains tasks T_1 and T_2, and runs in the intervals $[9, 21]$ and $[28, 40]$. Tasks are specified as follows.

$$\left.\begin{array}{llll} T_0 : p_0 = 43; & e_0 = 3; & ID_0 = 0 & \rightarrow P_0 \\ T_1 : p_1 = 32; & e_1 = 3; & ID_1 = 0 & \\ T_2 : p_2 = 52; & e_2 = 6; & ID_2 = 1 & \end{array}\right\} P_1 \quad (3)$$

Defining $MTF = 52$, $PART\_CS = 3$ and $TASK\_CS = 1$, SPIN encounters an error trail, outlined in Figure 3, in which the lower priority task T_2 does not have access to its full budget. Now, if we change the execution time of T_1 from 3 to 2 time units, i.e., if we reset $e_1 = 2$ in partition P_1, SPIN is not able to find an error trail. This scenario is depicted in Figure 4, and hence we can conclude that the schedule with this alteration is feasible, contrarily to the one specified in 3.

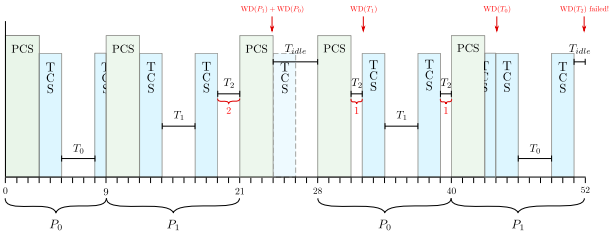

Figure 3: Error trail found by SPIN while evaluating the feasibility of the schedule specified by 3.

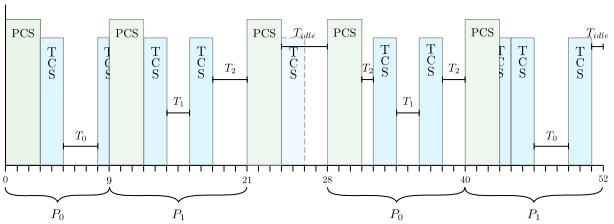

Figure 4: Feasible schedule obtained from 3, resetting $e_1 = 2$.

4. Asynchronous Communication Mechanism (ACM) in Interpartition Communication

Given that the system is composed of a set of partitions, each of which responsible for the computation of a certain function, it is expected that they will need to communicate in order to achieve the final goal of the respective system. Typically, processes communicate through messages: one process writes a message and passes it through a connector, that we will call *channel*, to other processes that may read it. In our case, communication is asynchronous, whereby another connector, called *port*, responsible for storing messages, is necessary at each endpoint of the channel. Figure 5 illustrates the structure of the communication mechanism among processes.

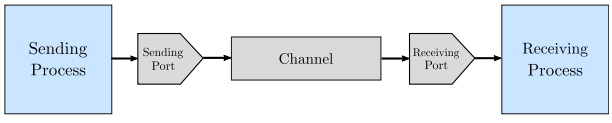

Figure 5: Structure of connectors in message passing between processes.

We will consider two possible modes of transferring a message in a port: *queuing* and *sampling*. In a queuing port, messages are stored in a bounded FIFO (first-in/first-out) queue, meaning that new

messages are stored in the end of the queue and the next message to be read is the oldest one, which is in the head of the queue. Since the queue must have a finite size, there is the possibility of overflow. In that case, new messages will be discarded until there is enough space to store new upcoming data. Whenever a partition receives a message, it consumes it from the channel, providing more space to new messages. Channels with source and destination queuing ports only have one destination process, so we have a $1:1$ correspondence between partitions.

In sampling ports, messages are not buffered. Instead, when a partition wants to send a new message, it stores it directly in the corresponding port, regardless of whether the previous message is still there. If that is the case, the previous message is overwritten by the most recent instance. The same happens in the source ports. So, there is only space for one message at a time and there is no possibility of overflow, given that a new message always overwrites the previous one, contrarily to queuing ports. In this way, the sender can send a message at any moment, and hence some messages may never get to be read. Moreover, in this transfer mode, whenever a receiving partition receives a new message, it does not consume it. In opposition to queuing ports, in sampling ports there can exist multiple destination partitions, therefore presenting a $1:N$ correspondence.

4.1. Implementation Aspects

As the communication mechanism is described, partitions never operate in the same physical space and, therefore, message coherence is always verified. However, due to performance and efficiency reasons, communication between partitions is not implemented though message passing, but by shared memory. The data structure shared by communicating partitions will be called *RAM channel*, and contains the following data

(i) *message buffer*, the bounded data structure in which messages are written. Each message is stored in a data area called *slot*;

(ii) *control variables*, which are shared variables used to determine given aspects of messages, such as whether the writer is active or not, which slot it is writing, etc.

Relatively to the message buffer, it is a FIFO queue in queuing ports and, according to the system's description, should be a single buffer when dealing with sampling ports. However, another implementation technique was used to store messages in sampling mode, called *multiple buffering*. This technique supports more than one message slot, in which the writer keeps writing new messages in the available slots, but only the one containing the most recent message is visible to the readers.

4.2. Properties of the Communication Mechanism

Since we have two distinct modes of transfer, we can already expect that communication with different ports verifies different properties. Nevertheless, there is a property that is common to both port types, which is the asynchronous communication.

> **Asynchronism:** During the communicating operations, no process can affect the timing of any other.

An immediate consequence is that the implementation cannot contain any lock primitives in order to achieve synchronization in reading and writing in the same message slot.

Since, in queuing transfer mode, the data structure that stores messages shared between processes is a FIFO queue, we can only expect that the properties that should be verified by our system concern such structure. They are as follows.

> **Mutual Exclusion:** The reader and the writer can never access the same message slot simultaneously.
>
> **FIFO:** The reader always reads the oldest message in the message buffer.

The mechanism in communicating messages with sampling ports is a little more sophisticated. Contrarily to queuing ports, the writer must always be able to write new data. This is a *non-blocking* property, in the sense that the progress of the writer is not influenced by the progress of any other process. Consequently, we will not impose that our ACM verifies mutual exclusion, but only *data coherence*, in which the reader is able to detect a corrupted message. Moreover, the reader should always read the *freshest* message, in opposition to queuing ports, in which the reader always obtains the oldest one. The last property imposes that the order in which the writer produces new instances is maintained by every reader, even though message loss (when the writer writes so fast such that the reader cannot keep up the pace) and repetition (reversed situation) are allowed. The properties that must be verified in sampling transfer mode are summarized below. The nomenclature of the last two properties is based on the work of Simpson in [12, 13] and Rushby [11].

> **Non-Blocking Writer:** The writer can never be blocked by the execution of any reader.
>
> **Data Coherence:** In every successful reading operation, the data attained must be coherent.

Data Freshness: Any message read by a given reader must not be older than the latest message written in the moment the reader starts the respective reading operation.

Sequencing: A reader should never return data older than which it has returned before.

4.3. System Model

In order to verify the properties stated above, we modeled the interprocess communication mechanism in TLA+, a very expressive mathematical language, which was later model checked using TLC. The writer and readers are modeled as processes, indexed by 0 and $i \in \{1, \ldots, N\}$, where N is the number of readers. Let $max\_nb\_message$ be the total number of slots, also corresponding to the size of the message queue, modeled by the variable $queue$. The original C code uses two pointers, namely $first$ and $last$, which point to the head and the tail of the message buffer, respectively, and are naturally initialized to 0 and $max\_nb\_message - 1$. The writer always operates in slot $(last + 1) \mod max\_nb\_message$. The slot that the reader will read from differs according to the transfer mode. In queuing ports, it will be the head of the queue, which is pointed to by $first$, while in sampling ports, it reads from the slot pointed to by $last$.

Letting $windex$ and $rindex[i]$ denote the index of the slot in which the writer and reader i are currently operating, respectively, the mutual exclusion property can be written in TLA+ as follows

$$\begin{aligned} MutualExclusion \triangleq\ & \forall i \in 1 \,..\, N : \\ & \wedge pc[0] = \text{“write”} \\ & \wedge pc[i] = \text{“read”} \\ & \Rightarrow windex \neq rindex[i], \end{aligned}$$

where pc stands for program counter and “write” and “read” are the labels corresponding to the writer and reader being in the critical section, respectively. With queuing ports, we have obtained that our model does not violate the mutual exclusion property. Since in the current specification of sampling ports, the readers always read the messages they obtain, i.e., all readings are successful, the data coherence property is, in this case, equivalent to the mutual exclusion property. After running the model with sampling ports, a single reader and a message buffer of size 2, the TLC model checker reported an error trace, outlined in Figure 6. The content of $queue$ is binary: 0 represents an empty slot, while 1 indicates it contains a message. $W_i()$ denotes the i^{th} write operation, while $R()$ a read operation.

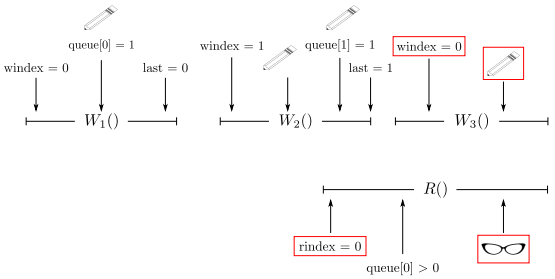

Figure 6: Error trace found by TLC violating the Mutual Exclusion property with sampling ports.

4.4. The Non-Blocking Write Protocol

Now that model checking techniques helped us identify that the implementation does not respect the system specification, we must correct our model in order to satisfy the mutual exclusion property (and all the others) in the case of sampling ports. In other words, we need to introduce an ACM between the writer and the readers when using the sampling transfer mode. This type of problem has already been studied in the past century by several computer scientists and mathematicians, with a historical overview presented in Figure 7.

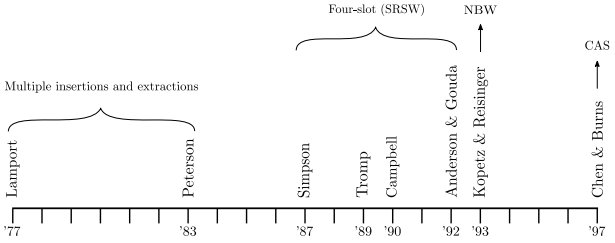

Figure 7: Some of the most relevant algorithms developed in the twentieth century for ACM displayed in chronological order.

Due to the context of these solutions, we decided that Kopetz and Reisinger's Non-Blocking Write (NBW) protocol [4] was the most adequate to our system. The idea of this protocol is that all processes are non-blocking, with the readers being able to detect reading of corrupted data. If that is the case, then the reader retries the operation until obtaining a consistent message. The protocol uses a single control variable, denoted CCF_i, which stands for concurrency control field, for each message i. These variables are initialized to 0. Each time the writer wants to write an instance of the i^{th} message, it increments CCF_i, and, when it is done, it increments it again. Since the writer is the only process that can modify this variable, the writer is writing a new instance of the message i if and only if CCF_i is odd. Let R_i be the range of CCF_i and $bcnt_i$ be the number of slots (in the paper the authors use the term *buffers*) reserved for storing message i.

The control field is used to infer if there was interference by the writer while reading, which happens when, during a read operation, $bcnt_i$ or more slots were written, as well as to determine which slots the reader and writer operate in. The protocol is only guarantee to work if the range of CCF_i, R_i is a multiple of $2 \times bcnt_i$, and in case the maximum number of interferences by the writer in a single read operation is $R/2-1$. The pseudo-codes of the writing and reading processes for the NBW protocol are shown in Algorithms 1 and 2, respectively.

```
CCF_old := CCF_i;
CCF_i += 1;
(* write in slot ⌊CCF_old / 2⌋ mod bcnt_i *)
CCF_i := CCF_old + 2;
```

Algorithm 1: Write pseudo-code for the NBW protocol.

```
start:  CCF_begin := CCF_i;
    (* read slot ⌊CCF_begin / 2⌋ − 1 mod bcnt_i *)
    CCF_end := CCF_i;
†   if CCF_end < CCF_begin
      then CCF_end := CCF_end + R_i;
    if CCF_end − 2 * ⌊CCF_begin / 2⌋ > 2 * (bcnt_i − 1)
      then goto start;
```

Algorithm 2: Read pseudo-code for the NBW protocol.

Before inserting the NBW protocol in our implementation of the communication mechanism, we modeled its original version in the PlusCal language (automatically translated to TLA+) and determined that it verifies the three properties stated above, namely data coherence, freshness and sequencing. The positive results from the TLC model checker gave us the confidence to introduce this protocol in our communication mechanism with sampling ports.

4.5. Strengthen the NBW Protocol: from Data Coherence to Mutual Exclusion

Rather than inserting the NBW protocol as it is described in [4], we observed that if the periods of the writer and readers are known a priori, we can compute the minimum number of buffers required so that the writer is never simultaneously operating in the same slot as any reader. In this way, our system satisfies a property stronger than data coherence, which is mutual exclusion. We understand by a period the interval of time between two executions of the same process. Let p_w and p_r denote the period of the writer and the maximum period of the readers, respectively. More concretely, if the writer started executing in instant t, then its next execution will start in instant $t + p_w$, and the analogous for the readers.

The authors of the original NBW protocol state, in [4], that *"These additional buffers will be used to set up periods in time in which a message is guaranteed not to change"*. Let us denote this period (in which messages are not changed) by p_{mnc}. After the write completion in slot i, this message will remain intact until the writer performs $bcnt-1$ operations. When it writes the $bcnt^{\text{th}}$ message, it will overwrite the current message in slot i. Thus, this period is given by

$$p_{mnc} = (bcnt - 1) \times p_w. \tag{4}$$

If we want our implementation to verify the mutual exclusion property, the message that the reader is accessing cannot be changed during the respective reader's execution. In the worst case, the reader starts reading a message just before the writer releases a new message, say, in slot $i+1$. Thus, the reader will read the previously released message, which is in slot i. Therefore, the period in which a message is not changed must be at least as large as the sum of the writer and reader's period, as illustrated in Figure 8. We can translate this into an inequality and compute the minimum slots needed to guarantee exclusive access to messages.

$$\begin{aligned} p_{mnc} \geq p_w + p_r &\Leftrightarrow (bcnt-1)p_w \geq p_w + p_r \\ &\Leftrightarrow (bcnt-2)p_w \geq p_r \\ &\Leftrightarrow bcnt \geq \frac{p_r}{p_w} + 2 \\ &\Rightarrow bcnt_{min} = \left\lceil \frac{p_r}{p_w} \right\rceil + 2. \end{aligned} \tag{5}$$

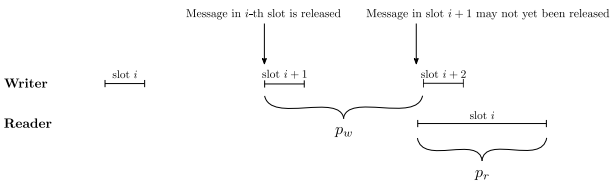

Figure 8: Worst case scenario in which the reader cannot access the same message slot that the writer.

Moreover, we can also derive the minimum value for the range of the CCF variable based on the periods of the writer and reader. The replacement of the $bcnt$ and R parameters by the writer and readers periods allows us to compute the former two values in a way that the protocol satisfies mutual exclusion, and not only data coherence. Consequently, all reads will be successful and, thus, the reader will never be looping trying to read a message, while the writer is overwriting such instances. Lastly, we obtain a more efficient protocol, given that no checks from the reader are necessary in order to verify if the message read is coherent.

In addition to modeling the new version of the system communication, using the NBW as the ACM for sampling ports, we also verified if our calculations of the minimum number of slots and range of the CCF variable were correct, in which the readers and writer could execute simultaneously, which requested the introduction of the concept of time. This was also abstracted using the TLA+ language, whose models yielded no error traces.

5. Memory Model over the ACM

In the previous section, model checking techniques were applied to verify if a given ACM enjoyed a given set of LTL properties, assuming that the instructions of the algorithm were executed in program order. However, looking at our problem in a more lower-level perspective, we see that the processors may not execute program statements by the order in which they appear, and thus, we cannot guarantee that the same properties are still verified, in this case. This is called *memory reordering*, and is performed by most processors at runtime.

Lamport introduced the notion of *sequentially consistent* multiprocessor, in [6].

> A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

According to Gharachorloo, in [1], such multiprocessor must satisfy the two following properties

1. **Program order:** each processor executes instructions in program order, completing each memory access instruction before advancing to the next one.
2. **Write atomicity:** every write operation becomes visible to all processors at the same time.

The sequential consistency correctness property severely restricts the use of parallelism and, consequently, reduces the system's performance. Therefore, most of the modern processors do not guarantee sequential consistency. Instead, they have a what is called *relaxed memory model*, that relaxes constraints in program order and/or write atomicity. We will only consider the first type of relaxation, which concerns the order of operations within a given processor.

5.1. Case Study Memory Model

All processors of the system under study are equal and correspond to Freescale P1010 processors. Their single relaxation is that a write occurring after a read in program order can bypass the read (denoted $W \longrightarrow R$ bypass), i.e., be observed as happening after the read, in which the memory addresses of the two operations must be distinct. All addresses of shared variables are stored in the writer's processor memory. In this way, the writer has direct access to them, while the readers do it remotely through an interconnection. In order to communicate, each processor has a PCI Express device, often referred to as PCIe, which is then connected by a PCI Express link, where the memory requests are passed. The PCI Express has also a relaxed memory model, where a write operation is allowed to bypass read operations.

Let us now summarize how the ordering in which operations are completed in the communication mechanism, taking into account the system's memory model. In the writer's case, the processor may reorder reads that are prior to writes. The reader's case is slightly more complex. The processor starts by issuing the memory requests according to the same orderings rules, given that both processors are of the same type. Then, the PCI Express remotely accesses the writer's memory by forwarding the requests according to its memory model, which allows $R \longrightarrow W$ bypass. Therefore, a request issued by the writer can only be reordered once, while a request issued by a reader may suffer reordering from two different memory models. The architecture of the communication mechanism is illustrated in Figure 9.

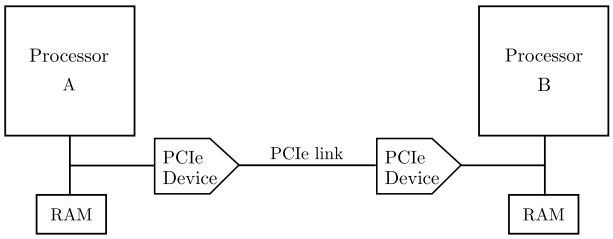

Figure 9: Architecture of the communication mechanism.

5.2. Model Checking the System's Memory Model

Lamport presents several models of multiprocessors with different memory specifications in [7, Chapters 5, 11.2]. Although Lamport's solutions do not concern the exact same problems than ours, it brings us insight relatively to reason about how to abstract memory models and how to model the behavior of a system with an underlying relaxed memory model. Therefore, our specification will be inspired in Lamport's approach.

The relevant hardware components in our system are the processors, the main memory, which, in this case, is the writer's memory, and the PCI Express link that allows the communication between the reader and writer. We have already modeled the

behavior of the processors, which consists of writing or reading messages. What remains to specify is how processors communicate with the other components, through an *abstract interface*. Figure 10 depicts the communication flow, where the hardware components are wrapped in a continuous line box, while the abstract interfaces are in a dashed line box.

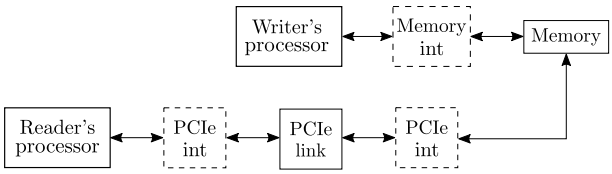

Figure 10: Hardware components connected by abstract interfaces, inspired in [7, pg. 45].

The main difference from the model of a reader or writer process in Section 4, is that, instead of reading and writing a shared variable atomically, the process will issue a read and a write request, respectively. Since we are now in a lower level environment, we will use the memory address of a shared variable (that uniquely identifies it), i.e., the place where the variable is stored in memory, to issue requests.

A request is represented in TLA+ as a record, uniquely identified by the process, *proc*, that issued it, the operation, *op*, (either a read or a write), the memory address, *adr*, of the respective operation and, lastly, the value to be written, *val*, in the case of a write. As resembling the memory models produced by Lamport in [7], memory requests will be stored in a tuple and are removed from immediately after being complete. $reqWrQ$ denotes the set of pending requests issued by the writer in program order, $reqRdQ$ contains all ordered pending requests issued by readers and, which are then passed, according to the PCI Express memory model, to the PCI Express list of requests, $PCIeQ$.

In order to model the order in which writer's requests are responded, we introduced a process, named $RespondToWr$, that is responsible for it. It can only execute when $reqWrQ$ is non-empty. In that case, it starts by defining the set of indexes of requests that can be responded, according to the $W \longrightarrow R$ bypass. If it is a write operation to be completed, it must be the first element of the queue, since writes are not allowed to bypass any other operation. In contrast, the only requirement for a read operation to complete is to not have any preceding read operation (since reads cannot bypass other reads) nor any other operation with the same memory address (remember that program order relaxations solely concern pairs of requests with different memory addresses). Such set can then be written in PlusCal as

$$
\begin{aligned}
S\_wr := \{i \in 1 \,..\, Len(reqWrQ) :\\
&\lor i = 1\\
&\lor \land reqWrQ[i].op = \text{“Rd”}\\
&\quad \land \forall\, j \in 1 \,..\, (i-1) :\\
&\qquad \land reqWrQ[j].adr \neq reqWrQ[i].adr\\
&\qquad \land reqWrQ[j].op \neq \text{“Rd”}\}.
\end{aligned}
$$

Afterwards, an element of this set is non-deterministically chosen, resorting to the PlusCal *with* construction, so that the request that is about to be responded is the record in that position in the $reqWrQ$ queue.

A memory access from a reader is divided in two phases, with respect to the memory models. Firstly, the process named $IssuePCIeRequest$, selects a request from the reader, respecting the processor's memory model, to be moved to the $PCIeQ$ queue. This is done by defining the set of indexes, S_rd, similarly to S_wr, of enabled requests to be responded, from the $reqRdQ$ queue, and select one of them non-deterministically again with the *with* construction. The difference from the $RespondToWr$ process, is that, instead of completing the request, it is redirected to the $PCIeQ$ queue and removed from $reqRdQ$.

Now that the request is in the PCI Express, it is responded according to the respective memory model, adopting the same technique as before. It starts by defining the set, S_pci, of indexes of operations in $PCIeQ$ that can be currently responded. Since the only relaxation in the PCI Express is $R \longrightarrow W$ bypass, a read request can only be responded if it is the first element of the queue. On the other hand, a write is only allowed to bypass read requests with different address, whereby such set can be written as

$$
\begin{aligned}
S\_pci := \{i \in 1 \,..\, Len(PCIeQ) :\\
&\lor i = 1\\
&\lor \land PCIeQ[i].op = \text{“Wr”}\\
&\quad \land \forall\, j \in 1 \,..\, (i-1) :\\
&\qquad \land PCIeQ[j].adr \neq PCIeQ[i].adr\\
&\qquad \land PCIeQ[j].op = \text{“Rd”}\}.
\end{aligned}
$$

Now that we are considering relaxations in program order, the moment in which processes are accessing the critical section, i.e., are reading or writing a message, do not correspond to the instant in which they issue the corresponding request, since these are not necessarily responded atomically. Thus, it was needed to redefine the labels, in the code, where processes are in the critical section.

6. Conclusions

The core work of this dissertation consisted in using model checking techniques to analyze a set of correctness properties of a fairly complex real-time

distributed system integrated into GMV's XKY project.

Our work focused on three safety-critical aspects of a real-time distributed system: scheduling analysis, interpartition communication, and the processor's memory model in the scope of communication. Our major contribution was the discovery of an implementation error in the communication of partitions, in which readers obtained corrupted messages. The communication mechanism was modeled in a 40-lines PlusCal algorithm and the TLC model checker found an error trace in a matter of seconds. In order to correct this, we introduced the known NBW protocol [4], where we strengthened the data coherence property to mutual exclusion, by knowing, a priori, the periods in which the writer and readers operate.

This is not the first attempt to use formal methods in industry. On the contrary, big and influential organizations such as Amazon [9] and NASA [10] have used and continue to use formal methods in order to validate their soft or hardware products. We hope that our work encourages GMV and other safety-critical software companies to resort to formal methods in either design, implementation or verification phases to study the correctness of their systems.

Model checking has some limitations: it only verifies the model of the system rather than the actual system and it suffers from the state-space explosion problem and, hence, realistic systems are often too large considering the available amount of memory. Therefore, model checking is used to provide the user confidence in their system and cannot fully guarantee the system's correctness. Nevertheless, most implementation errors can be found in small instances of the problem, and thus our verification work is still valid and useful.

We would like to conclude by pointing out that expertise in developing abstractions may significantly reduce the amount of time required for the model to be checked. Thus, we can expect that the increasing experience in using formal methods may help us produce models of more realistic systems that can be verified in a reasonable amount of time, whereby the use of model checking techniques should be encouraged in the computer science/engineering community responsible for developing such safety-critical systems.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. Technical report, Western Research Laboratory, September 1995.

[2] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer-Verlag London, 2008.

[3] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2006.

[4] H. Kopetz and J. Reisinger. The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem. *Real-Time Systems Symposium*, 1993.

[5] L. Lamport. Principles and Specifications of Concurrent Systems.

[6] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[7] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 1st edition, 2002.

[8] N. Lynch. *Distributed Algorithms*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 1996.

[9] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, Mar. 2015.

[10] J. Penix, W. Visser, S. Park, C. Pasareanu, E. Engstrom, A. Larson, and N. Weininger. Verifying Time Partitioning in the DEOS Scheduling Kernel. *Formal Methods in System Design*, 26(2):103 – 135, March 2005.

[11] J. Rushby. Model Checking Simpson's Four-Slot Fully Asynchronous Communication Mechanism. Technical report, Computer Science Laboratory, Menlo Park CA USA, July 2002.

[12] H. Simpson. Four-slot fully asynchronous communication mechanism. In *Computers and Digital Techniques*, volume 137, pages 17–30. IEEE, January 1990.

[13] H. Simpson. Freshness specification for a class of asynchronous communication mechanisms. In *Computers and Digital Techniques*, volume 151, pages 110–118. IEEE, March 2004.