Presented at FORTE'94, 4-7 Oct. 1994, Berne, Switzerland.

Modelling and Verification of a Multiprocessor Realtime OS Kernel

Thierry CATTEL[1,2]

[1]Laboratoire de Téléinformatique, Ecole Polytechnique Fédérale
CH-1015 Lausanne, Switzerland

This paper reports the experience of the Software Engineering Laboratory of the National Research Council of Canada with the modelling and verification of the kernel of Harmony, a portable real-time multitasking multiprocessor operating system. In this paper we explain the aim of this study and give the first results. We use a modelling approach and formalize the models of the system, the scenarios and the properties that are to be checked in PROMELA using the SPIN tool. Several models of the systems were produced with various degrees of abstraction and completeness. The most recent is a tractable one that enables the expression, simulation and verification of any scenario that consists of a bounded number of tasks that may use all the services of the kernel. An exhaustive verification of the intertask communication features of Harmony was carried out by model-checking. It revealed a bug that has been in the system for more than ten years. The first verifications of the dynamic task management primitives lead to the discovery of other bugs and serious critical races. This paper shows that it is possible to detect more than deadlocks when using formal methods for the study of a real medium-sized operating system that encompasses complex internal management.

Introduction

Formal specifications of application software are appearing in many areas [8]. However it is not yet a common practice for operating systems. Some attempts have been made, in particular for UNIX [5], but they mainly have provided an unambiguous documentation. Some complete verifications of limited parts of systems [4-22-23-24] were done. Some attempts of abstract architectural specification have also been done [21] but have not led to significant results yet.

When we began this work, our objective was clearly to illustrate of the applied use of formal methods by proving or verifying some interesting properties of the Harmony kernel. The sole production of a specification was not considered sufficient.

The complexity of concurrent systems led us to choose a framework favoring proof or verification power over expressive power. We would not only address parts of the Harmony kernel but, as much as possible, its totality. Manual proof techniques, even supported by proof environments, did not seem to be realistic tools to use because of the complexity of the task and because of the very error prone nature of such activities, even for the smallest systems. An automatic or semiautomatic approach seemed to be a better choice. We initially studied the state of the art and could see the common points with the domain of communication protocol design and validation [25]. Specification and verification of real-sized communication protocols is a common practice [17]. Communicating finite state machines appeared as the obvious choice to model our system. Two kinds of approaches tend to be used in studying finite state machines. Transition-oriented techniques [11] avoid the issue of state explosion,

---

[2] This research was done at the Software Engineering Laboratory, National Research Council of Canada, Ottawa, Ontario, Canada K1A OR6, and is registered as technical report NRC 38309.

but usually require manual proofs. State-oriented techniques such as reachability analysis or model-checking may rely on automated tools. State explosion is always an issue, but we found a tool that addressed this problem. We decided to use PROMELA as a specification language, and to use the SPIN[14-15] tool to check the large state spaces for different kinds of properties. PROMELA's expressive power is rather low, since it provides only a few basic data types and only very primitive type constructors, but it offers the possibility for expressing safety, liveness or precedence properties by logical assertions, valid end states, progress states, or linear temporal logic expressions. A model may be animated by the SPIN simulator, or submitted for verification. Full state or partial state verification may be used, depending on the problem size. During simulation or verification, the properties specified are checked, and if an error is found, it is possible to run the simulator again and look at the state values at each step.

In the next sections we briefly present the problem, we explain the approach used to produce the models of Harmony and we show how these models were used to verify some interesting properties. In this paper we mainly focus on deadlocks, livelocks, and other safety properties for the primitives of Harmony kernel.

1. Harmony Kernel

Harmony is a portable real-time multitasking multiprocessor operating system being developed at the NRC since the early 80's. Version 4.1 was released in 1994. For details about Harmony see [6-7]. Basically, Harmony is an operating system based on a micro-kernel and system servers. The features of the kernel are interrupt management, task scheduling, intertask communication, dynamic task management, memory management and I/O.

There are three interrupt management primitives. *_Disable* masks all the interrupts on the current processor and *_Enable* reenables those interrupts that can occur while the calling task is executing. *_Await_interrupt* puts a task to sleep until a given interrupt is raised.

Task scheduling is based on a priority preemptive scheme. There are *N* priority levels. On a given processor, the ready task having the highest priority runs until it blocks, or until it is preempted by a higher priority task that has become ready to run, or until it is killed. Priorities are fixed for the whole life of a task; only when it dies does it change temporarily. Four functions implement task scheduling. *_Dispatch* finds the task of the highest priority among the ready queues and gives the processor to it. *_Add_ready* makes a task ready and inserts it in the ready queues according to its priority. *_Block* removes the running task from the ready queues and dispatches another, *_Block_signal_processor* has the same effect as *_Block*, but it also signals the processor of the task given as a parameter. These four functions are not part of the kernel interface; they are private to the kernel.

The intertask communication primitives of Harmony, implement the *send-receive-reply* scheme. *_Send* is a blocking primitive. It sends a message to a specified task and returns when the correspondent has not only received the message but has replied to it. *_Receive* receives a message either from a specific task or from any task. It blocks until a message is sent to the receiver. *_Try_receive* has the same functionality as *_Receive* but is non blocking. *_Reply* is a non blocking primitive that returns a message in response to a sender. In comparison to the Ada rendez-vous that may be nested according to the lexical scope and that offers an implicit reply, Harmony is not a language but a collection of OS services whose reply is explicit and may be postponed indefinitely. This allows simpler tasking structures.

Dynamic task management is provided by the *_Create*, *_Destroy* and *_Suicide* primitives. Tasks can be created dynamically on any processor but cannot migrate during their life. A task may commit suicide or kill any other task even over processor boundaries or among its ancestors. Destruction is immediate even for a task that is waiting on another task. This leads to uniform semantics at the cost of a more intricate implementation. Tasks waiting on the victim must be unblocked with an adequate return code. Before a task is destroyed, its offspring are destroyed and the resources it possesses are released. Task management is implemented thanks to a system server called the local task manager (LTM) on each

processor. The Harmony kernel provides cache and non cache memory management and I/O facilities that we do not consider here.
Fig.1.1 shows a logical view of an application running under Harmony on a multiprocessor. Application tasks communicate with other tasks over processor boundaries or with local system servers, or may wait for an interrupt.

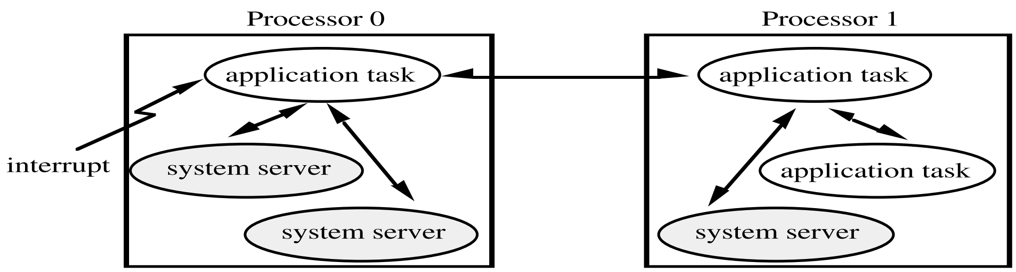

Fig.1.1 — Logical view of a multiprocessor application

Mutual exclusion across processor boundaries is implemented in the kernel by an ownership protocol rather than by locks. The implementation of this protocol is distributed among the communication primitives, the LTMs, and a procedure, *_TD_service* , that is invoked by the interprocessor interrupt handler. At most one processor can own a task descriptor[1] (TD) at a given time. Only the processor that owns a TD may access the fields that are shared between processors. Interprocessor interrupts are used to signal the transfer of ownership of a TD.

Several interprocessor interrupts coming from remote processors may arrive before the current one is treated. Therefore, a way is needed to memorize the pending interrupts. On some platforms, this is achieved by a hardware device called a hardware FIFO. On other platforms, software mailboxes [19] are used. The actual communication between tasks is done by copying the message from the sender's memory space to the receiver's.

2. Tools and modelling approach

PROMELA [14] models communicating sequential processes which may be created dynamically and communicate asynchronously or synchronously by messages through channels. The available data types are booleans and integers with the usual logical and arithmetic operators. Type constructors are restricted to channels and one dimension arrays. Control statements are the sequence, non deterministic choices, loops, unconditional branchings, and atomic sequence. Structuring facilities do not exist except the ones provided by the C preprocessor.

PROMELA models may be simulated with SPIN. It is possible to insert trace messages in the model, and to look at the model state after each step, namely the variables values and the current instruction. From a PROMELA model, SPIN generates a validator that may either attempt a complete space search or a partial one using the supertrace algorithm[12]. The validator detects all deadlock or livelock situations, violations of logic or linear temporal logic propositions[2-9-10], and unreachable code.

PROMELA has a very restricted expressive power. In particular, there is no direct means of representing data of type real, multiple dimensions arrays, pointers, structures and union type. Nor is it possible to structure models with subprograms. But the experience confirmed that this is not a problem and that it is always possible to address such issues.

A model is a simplification of a real system. The model only needs to be equivalent to the full system with respect to the properties that are being validated [13]. Verifying some properties of a system assumes two preliminaries: the modelling of the system and the modelling of the

---

[1] A task descriptor (TD) or task control block (TCB) is a record that holds information about a task: its name or id, its current state, its descendants, its ancestor, the memory blocks it has allocated, etc.

properties. The absence of deadlocks and livelocks was our minimal requirement, but we knew that some other properties would have to be checked. For instance task destruction is known to suffer from a possible but highly unlikely critical races.

To write the models, we took into account interrupts, intertask communication, task management and scheduling by producing partial models. We separately modelled each of these concerns in an abstract way and in a more concrete one. We thus had an incremental approach based on a *"divide to conquer"* principle that moved along two axes, as shown in Fig.2.1, horizontally from abstract to more detailed, and vertically from partial to more complete, the basic features appearing first.

Features \ **Abstraction**		*abstract*	*concrete*
Interrupts	***interprocessor***	hardware FIFO	software mailboxes
	physical	several priorities	several priorities
Intertask communication		channels	_TD_service FSM
Task mngt	***creation***	synch. + offsprings	synch. + offspr. +queues
	destruction	synch. + offsprings	synch. + offspr. +queues
Scheduling		nondeterministic	priority preemptive

Fig.2.1 — Criterias for incremental modelling approach

The best abstraction we found for interprocessor interrupts management is to have a FIFO on each processor that memorizes the pending interrupts; this is exactly what a hardware FIFO does. Communication through PROMELA channels is the obvious abstraction for intertask communication. Apart from looking at a particular aspect of it, (e.g. synchronization and offspring-tree management), it is difficult to produce an abstraction for task management. Non deterministic scheduling is a generalization of priority preemptive scheduling; it does not determine which task to run among the ready ones, but let any choice possible.

We successively produced several types of models. From abstract-partial to complete-detailed. With the first type, we attempted to produce a very abstract specification of the *send-receive-reply* scheme in terms of communications through PROMELA channels. We obtained satisfying results only for some restricted communication situations and topologies (e.g. *_Send-_Receive_any-_Reply*) but if we tried to produce a model addressing all situations, it became artificially complicated. We started using some overall specifications of Harmony written with Petri nets [18]. It was quite easy to simulate these nets or even to produce equivalent and clearer models using PROMELA channels. We did not gain very much with this first type of models for two reasons. First it could not provide an abstract specification for the detailed models we produced later, thus preventing any benefit from the specification refinement methodology [1-16]. Second the obtained model was not suitable for proving properties, in particular deadlock absence in the system. Another type of model focused on dynamic task management with some restrictions (e.g. a task cannot kill one of its ancestors). Since the communication situations involved in task creation and destruction occurs always on the one hand between the creator/killer and the local task manager (LTM) and between the LTM and the killed task (there is no communication between the LTM and the created), we could use PROMELA channels for abstracting intertask communications. This type of model relies on a nondeterministic scheduling totally implicit, namely no code is written for it, it relies on the scheduling of the simulator/validator. This type of model was very useful to better understand Harmony task management. Another type, described in §3., addresses the intertask communication as implemented in *_TD_service* and relies on a hardware FIFO and an explicit nondeterministic scheduling that lets run any ready task. The last type, described in §4., addresses intertask communication and dynamic task management with no restriction and relies on a hardware FIFO and a priority preemptive scheduling, namely each task has a priority and is allowed to run according to it when it is ready. This type of model really corresponds to the concrete system and is used in our current work for the actual verification of the whole kernel. The same type of model with software mailboxes instead of hardware FIFO, we did not write yet, but could serve in the future for performing more specific verifications.

3. Intertask communication

Before looking at the model used for the verification of intertask communication, we explain how it was obtained. In order to increase the confidence one has of a real system, it is possible to produce a model of this system, to express and check formally some of its properties. That means that if we succeed in checking or proving a property for the model, we are sure its true for the model, but for the real system it is hard to say, since unfortunately there exists no way of proving that a model corresponds to a real system. In the case of Harmony, the real system was studied through its C and assembler code. We tried to keep the detailed model as close as possible to the code, only abstracting by pruning irrelevant things for the verification we had in mind. We made this choice for easing model management and for convincing people that the model corresponds to the real system. A drawback is that it probably limits the abstraction one can produce of the system, perhaps disabling the actual verification of complex properties. Nevertheless, we succeeded in producing a detailed model using the full intertask communication and dynamic task management primitives, running according to a priority preemptive scheduling. Before reaching this point, we had already produced partial models for detailed intertask communication and roughly validated them. However, we found a more sensible practice to consist in taking the complete detailed model, pruning the irrelevant details out of it (task management, scheduling) in order to get a consistent restricted model suitable for the validation of intertask communication. We use this model in §3.1. It only addresses intertask communication synchronization, the actual data exchange through message is irrelevant for the properties we want to check, but could easily be taken into account if needed.

3.1. Modelling

The model of Harmony intertask communication is decomposed into several modules. At the top level, module *kernel* includes parameters, such as the number of processors, the definition of each part of the system and scenarios. It initializes the system by starting the interrupt handlers modelled as independent processes, and the user program. Module *td* models task descriptors (TD), defining data such as the state of a task, its correspondent, its processor and its send queue. It defines a function *_I_td* to initialize the TD of task, and *_Convert_to_td* to convert a task identifier of an existing task into its TD. Since we do not have pointers, in fact we do not manage task descriptors but only task identifiers. The send queue is limited to one-element queue, because the scenarios we will validate do not require longer queues. The receive and reply queues serve only in the case of task destruction, so we forget them here. Module *taskstates* is a set of constant definitions for all the possible states of a task. Module *processor* models interrupt masking primitives. We should note that in Harmony, there is no global mutual exclusion mechanism over processor boundaries, only local mutual exclusion mechanism relying on interrupt masking. A solution consisting in using the PROMELA *atomic* statement would be too strong, and could lead to forget some problematic interleavings. We thus implemented this mechanism at the model level thanks to spin-locks – but there is no use of spin-locks in Harmony though. When a process runs on its processor, and wants to mask, it takes the processor for itself only. The treatment of interprocessor interrupts is modelled into module *ipinterrupt*. It defines the hardware FIFOs as PROMELA channels, the primitives used to signal processors, *_Block_signal_processor* and *_Signal_processor*, and the interrupt handlers modelled as processes. The handling consists in calling *_Td_service* masked and then dispatching. Most of the kernel mechanics is encoded in the *_Td_service* procedure. This model uses a quite straightforward nondeterministic *scheduling*. Any task is allowed to run provided its state is READY. Module *queues* provides usual functions for queue manipulation. *_Send*, *_Receive* and *_Reply* almost exactly look like the ones in the real system, without all what concerns things we do not take into consideration, in particular cache memory management.

We do not consider dynamic task management in this model, but we have to *_Create* the application tasks. This initializes the task descriptor, makes it *READY* and actually runs the corresponding proctype. We implemented a very pragmatic startup that statically creates the application tasks of the scenario in the *init* process of the PROMELA model. For instance a sender with identifier 1, a receiver with identifier 2 running both on processor 0 :

```
#define _Start_up()\
    _Create(Sender,1,0);\
    _Create(Receiver,2,0)
```

The modelling of interrupt masking imposes a few adaptations. Any sequence of the system of the form *_Disable();S;_Enable();* may be left as it is in the model, except if *S* includes a blocking primitive such as *_Block* or *_Block_signal_processor*, it must be rewritten by unmasking just before calling the blocking primitive and by remasking right after. For instance *_Disable();*S_1*;Block();*S_2*;_Enable();* needs to be rewritten in the following manner *_Disable();*S_1*;_Enable();Block();_Disable();*S_2*;_Enable()*. This could strongly shock any Harmony designer who demands to call *_Block_signal_processor* or *_Block* disabled. The task possessing the processor, through the lock, when calling the blocking primitive, may be preempted but has not the possibility to give back the lock, which leads always to a deadlock on the processor. Our solution could certainly encompass an issue if *_Block_signal_processor* or *_Block* would modify some critical data, but it is not the case in this model. *_Block_signal_processor* sends a message to the channel FIFO, which is an atomic operation and blocks till the current task gets *READY*.

We now dispose of all what is necessary for writing scenarios that model application programs. We will see some examples of them in the section dedicated to verification.

3.2. Verification

Having the model, the problem it to know what to verify. We are first interested in checking the absence of deadlock or livelock . We could do more specific verifications such as checking that in a situation of successful communication the two communicating tasks return with an appropriated return code. We are certainly interested in checking that all the code in used too. Now the problem is to know how to verify the previous properties. This may be done by trying to find a set of scenarios as complete as possible that would exhibit all possible communication schemes. We can enumerate a list of all the possible situations that a communication task may face :

A task tries to communicate with :

(1) a non existing task
(2) an existing task that does not attempt to communicate with any task
(3) an existing task that communicates with this particular task in the right way (e.g. receive then reply)
(4) an existing task that communicates with this particular task but in a wrong way (e.g. no receive before a reply, a receive instead of a send, ...)
(5) an existing task that does not communicate with this particular task but another one

Cases (1) to (4) involves only two distinct tasks that may run on one or two processors. Case (5) involves three distinct tasks that may run on one, two or three processors. A scenario composed of T tasks may thus run on one to T distinct processors. Taking into account, obvious symmetry considerations based on the fact that we consider all the processors identical, there are P distinct affectations of T distinct tasks on T distinct processors, where P is the number of partitions of a set of T elements. For instance if two tasks t_1 and t_2 may run on p_1 and p_2, there are two distinct affectations: $\{[t_1,t_2]\}$ *and* $\{[t_1], [t_2]\}$. That means that either both tasks may run on a single processor (either *p1* or *p2*, equivalently) or that t_1 runs on p_1 and t_2 on p_2 (or equivalently t_1 runs on p_2 and t_2 on p_1). If three tasks may run on three processors there are five distinct affectations.

To be exhaustive, we should draw a three dimensions array, with all the possible relevant behaviors of the first task in columns, all the possible behaviors of the second task in raw and all the possible behaviors of the third task in the third dimension. We then have to analyze each cell by anticipating the different possible states each task could reach, e.g. first task returns with code 0 and second tasks gets *SEND_BLOCKED*,... (See Fig.3.2.1).

		1	2	3
	Sender (id=1) **Receiver (id=2)**	***_Send(2)***	***_Send(99/0)***	∅
1	***sender=_Receive(1) [;_Reply(sender)]***	2-1	0-RSB	_-RSB
2	***sender=_Receive(0) [;_Reply(sender)]***	2-1	0-RB	_-RB
3	***sender=_Try_receive(1) [;_Reply(sender)]***	SB-0 | 2-1	0-0	_-0
4	***sender=_Try_receive(0) [;_Reply(sender)]***	SB-0 | 2-1	0-0	_-0
5	***_Receive(99)***	SB-0	0-0	_-0
6	***_Try_receive(99)***	SB-0	0-0	_-0
7	***_Reply(1)***	SB-0	0-0	_-0
8	***_Reply(99/0)***	SB-0	0-0	_-0
9	∅	SB-_	0-_	_-_

Legend :
- ***1*** *task identifier of Sender*
- ***2*** *task identifier of Receiver*
- ***0*** *dummy task identifier for (try)receive ANY*
- ***99*** *task identifier of a non existent task*
- ***Ø*** *empty task*
- ***SB*** *SEND_BLOCKED*
- ***RSB*** *RCV_SPECIFIC_BLOCKED*
- ***RPB*** *REPLY_BLOCKED*
- ***RB*** *RCV_BLOCKED*
- ***_*** *non significant*
- ***x-y*** *Sender reaches state x and Receiver reaches state y*
- ***a | b*** *a or b depending on relative timing of Sender and Receiver*

Fig.3.2.1 — Elementary scenarios with two tasks

It is easy to write the code related to a given cell, submit it to verification, and interpret the results. In each error case we need to check if its a predictable deadlock, an assertion violation or something else. Do we need to exhaustively test the model this way ? Is the minimal complete set of scenarios smaller than the exhaustive one produced by this three dimensions array ? We believe so, though we cannot provide a completely formal justification. We can prune cases out of the cube that are redundantly treated. Taking into account the symmetry considerations argued above, we may reduce the cube to a pyramid. There are classes of cells having the same content. It is then possible to group the corresponding scenarios and verify them altogether. It chiefly saves time if the verification of the class produces no error. We are convinced that it is useless to test some particular two-task scenarios and most of the three tasks ones, if not all. We illustrate this point a bit further but first, we consider some two-task scenarios of Fig.3.2.1. We think that this table sums up all the interesting situations.

In cell 1-1 for instance, the sender whose identifier is 1, sends to the receiver whose identifier is 2. The receiver receives from this specific sender, and then replies to it. This is a regular case of successful communication, the sender should return with code 2 (its correspondent's identifier) and the receiver should return with code 1. The only difference in cell 2-1 with cell 1-1 is the *_Receive* any, but the resulting situation is the same. Both scenarios may then be grouped into a same class. In cell 4-1, depending on the relative timing of both processes, there may be two states reached. First, another regular case of successful communication, the sender has initiated its *_Send* before the receiver executes the successful *_Try_receive* and then the *_Reply*, both processes return with their correspondent's identifier as return code. Second, the receiver executes the *_Try_receive* before the sender has executed the *_Send*, the communication attempt fails and the receiver returns with code 0 and do not reply. When the sender does the *_Send* it gets *SEND_BLOCKED* and does not return.

Fig.3.2.2 shows the code for the class containing both scenarios of cells 1-1 and 2-1. Notice the *assert* clauses that identify impossible situations. Cells 5-1 to 9-1 (Fig.3.2.3) represent another class of scenarios. The sender always gets *SEND_BLOCKED* and thus never returns. The receiver returns with 0. Notice the assertion in *Sender* that *_Send* does not return.

```
proctype Sender(byte _Active){
    byte receiver;
    _Send(receiver,2);
    assert(receiver==2)}
proctype Receiver(byte _Active){
    byte c,sender,replyee;
    if :: c=0 :: c=1 fi;
    _Receive(sender,c);
    if
    ::  sender==1 ->
        _Reply(replyee,sender);
        assert(replyee==1)
    ::  sender!=1 -> assert(FALSE)
    fi}
```

Fig.3.2.2 — Scenario for cells 1-1 to 2-1

```
proctype Sender(byte _Active){
    byte receiver;
    _Send(receiver,2);
    assert(FALSE)}
proctype Receiver(byte _Active){
    byte c,sender,replyee;
    if
    ::  _Receive(sender,99)
    ::  _Try_receive(sender,99)
    ::  if :: c=0 :: c=1 :: c=99 fi;
    _Reply(sender,c)
    ::  goto FINISHED
    fi;
  CHECK:
    assert(!sender);
  FINISHED:
    skip}
```

Fig.3.2.3 — Scenario for cells 5-1 to 9-1

There are some *a priori* interesting scenarios that do not need to be verified because they may be decomposed into a set of elementary two-task scenarios. Let us take two examples. First, a two-task scenarios involving two senders that send to each other. Obviously the resulting situation is both senders *SEND_BLOCKED*. We think that verifying such a scenario or verifying the scenario corresponding to cell 9-1 of Fig.3.2.1 is equivalent, at least for the properties we check here. Second, a three-task scenario, a priori more problematic ; both first tasks correspond to cell 1-1 of Fig.3.2.1, the third attempts to reply to the first in place of the second. We think that it is equivalent to verify scenarios of cells 1-1 and 7-3. We claim that Fig.3.2.1 presents all the elementary scenarios. It even includes some redundancy. For instance it is certainly equivalent to verify only cell 9-1 instead of cells 5-1 to 9-1. We have coded and verified the scenarios of Fig.3.2.1 taking into account one or two processors, first separately and then grouped into classes. We tried some scenarios with three tasks too. In every cases, the results were consistent with what we have said above.

Another aspect is unreached code. Which scenario do we need to write to check if some code is never reached and thus is useless ? Intuitively, a scenario that verifies all the elementary situations at the same time would fit. We argued that Fig.3.2.1 contains all these elementary situations and it is even possible to suppress last raw and last column, it then is easy to write the corresponding scenario that verifies all the situations (Fig.3.2.4).

```
proctype _Sender(byte _Active){
    byte receiver,c;
    if :: c=0 :: c=2 :: c=99 fi;
    _Send(receiver,c)}

proctype _Receiver(byte _Active){
    byte c,sender;
    if :: c=0 :: c=1 :: c=99 fi;
    if
    :: _Receive(sender,c)
    :: _Try_receive(sender,c)
    :: goto REPLY
    fi;
  REPLY:
    _Reply(rid,sender)}
```

Fig.3.2.4 — Scenario for identification of unreached code

Now that we have, model, properties and scenarios, how do we proceed to verification and how do we interpret the results ? First we separately validated all the scenarios of Fig.3.2.1. Before we added a *end:* label in front of the loop of *_Int_handling*, so that when both the sender and receiver finish, the handler is blocked on an interrupt reception but this is considered as a valid end state. For instance for cell 9-1 of Fig.3.2.1, verification confirmed that the only end states were the expected deadlocks where the sender is *SEND_BLOCKED*. It is possible to stop the validation after a given number *N* of errors encountered (*pan -cN*) and get a trace of the execution that lead to error number *N* ; it is thus possible to have a close look at each error. This was what we did, which confirmed the expected results of Fig.3.2.1. Then we grouped scenarios by classes and got the same results. We also tried a few scenarios with three tasks, for confirmation.

We found a bug during this validation. The problem concerns primitive *_Receive* that, on the opposite of *_Send*, *_Try_receive* and *_Reply*, does not check if the correspondent exists before signaling the correspondent's processor. This caused an error in the model, which could be interpreted as signaling a non existent processor. In the best case the result was a deadlock that left the receiver blocked, instead of returning with code 0, in the worst case an exception was raised leading to a system crash. We could fix it, implemented it in the model and returned to validation. This time no error was found. Namely there was no livelocks and only predictable deadlocks occurred, but none in case of predictable successful communication and no assertion was violated.

Regarding the identification of unreached code, we proceeded exactly the same way. In all the cases, we tried the scenario on one or two processors. The verification covered all the code of the model except at some places. There are two reasons to justify this very limited unreached code. First, when reabstracting the complete model, only keeping things related to intertask communication, we only suppressed entire cases of *_Td_service*, but never modified the remaining cases that contains some code useful to test if a task has died or if a message exchange was aborted. This code is obviously useless since we take task management out of consideration. We did not modify the primitive code neither and some similar code remained. Second, we use macros to simulate functions or procedures that have conditional bodies. If we use such a macro several times, we are ensured that some of its code will not be reached in at least one place. In order to get meaningful output from the verification, we need to use each macro once at the most. This constraint is rather easy to respect while using the macros that implement the primitives of Harmony, when writing scenarios (See *REPLY*, Fig.3.2.4) but it is quite impossible when writing the model. The fact that the validator indicates that all the code is reached does not prove that every situation in the code was reached, though. For a conditional statement such as :

if :: c1 && c2 -> S1 :: !c1 || !c2 -> S2 fi

it is enough that c_1 or c_2 be false to execute S_2, but, if we do not rewrite the code with exclusive conditions, we are not ensured that the case were both c_1 and c_2 were false was encountered. Obviously, we did not bother to write the model this way :

if $:: c_1 \,\&\&\, c_2 \rightarrow S_1 \;\; :: !c_1 \,\&\&\, c_2 \rightarrow S_2 \;\; :: c_1 \,\&\&\, !c_2 \rightarrow S_2 \;\; :: !c_1 \,\&\&\, !c_2 \rightarrow S_2$ *fi*

4. Task management

4.1. Modelling

The detailed model of Harmony complete kernel contains all the modules of the model related to intertask communication of §3.1, possibly completed or modified, plus specific modules that correspond to task management primitives, and local task managers.

Module *ltmmsg* in particular defines constants used by the LTM and the task management primitives *_Create*, *_Destroy*, *_Suicide*. Each primitive sends an appropriate message to the LTM in charge of the particular task type to be manipulated. Module *td* is much bigger than in the previous model, because a TD is composed of all the fields necessary to manage the offspring creation tree, the various queues, and some extra fields that are detailed further. Module *queues* is modified to treat real queues, module *trees* manages the offspring trees. Module *tdservice* is completed with the extra cases that correspond to task management plus an initial control for avoiding to treat several times a task being destroyed. Module *copymsg* takes into account exchange of messages of various lengths. Module *ltm* models the local task manager. This is a system server that runs on each processor and is in charge of managing the tasks that are running on the processor. It accepts three types of messages : *CREATE*, *DESTROY* and *SUICIDE*. The *CREATE* and *DESTROY* messages are sent by the primitives *_Create* and *_Destroy*. The *SUICIDE* message is sent by a task being killed as explained below. Task creation may be dynamic in PROMELA (*run*). It is okay for simulation, but it introduces infinite state space at validation time. The solution consists in prerunning a bounded number of application tasks at the initialization of the model, the actual creation of each task consists in synchronizing the beginning of its execution on the value of field *state* of its TD. This is slightly complicated by the possibility to create several tasks types (templates). This is solved by introducing an extra field in each TD, called *index*, that indicates the task type. We then only need a generic application task type (See Fig.4.1.1). By convention task of index TASK1 is the user program that initializes the application, it corresponds to the *main* task of a real system. Task identifier attribution is centralized in our model on the opposite of what is done in the real system, where each processor generates structured task identifiers, but this choice did not introduce any issue so far.

```
proctype Task(byte _Active){...
START:
 (state[_Active]==READY && valid[_Active]) ->
  do
  :: atomic{
     ( _Active==running[processor[_Active]] &&
      !treatment[processor[_Active]] &&
      (!len(fifo[processor[_Active]]) || masked[_Active])) ->
     if
     :: killed[_Active] ->
        /* _Infanticides() */
        ...
     :: !killed[_Active] ->
        if
       :: index[_Active]==TASK1 ->
#include "scenariotask1"
       :: index[_Active]==TASK2 -> ...
      fi
    fi}
  od}
```

Fig.4.1.1 — A generic application task

Task destruction imposes a difference in the control over application task execution. This control needs to be finer, to be able to take into account a task destruction as soon as possible. In the real system, when a task is killed, first, if it is involved in intertask communication, it is retrieved from its correspondent's queues. Then the LTM changes the task application code by the procedure *_Infanticide* and sets its priority to the same as the LTM's. This code serves for killing the victim's offspring and ends by sending a *SUICIDE* message to the LTM. The LTM then invalidates the victim's TD and unblocks all the possible victim's correspondents by walking through the victim's queues. When it is done the victim is really dead. The *_Send* in *_Infanticide* never returns, because the LTM never replies to this king of message. A call to *_Destroy* always return with a zero code. This is because it is implemented as the emission of a *DESTROY* message to the proper LTM. When the LTM receives the message, the requestor (killer) is enqueued on the LTM' s reply queue. The LTM retrieves it from its own reply queue and enqueues it on the victim's reply queue. When the victim reaches the very last step of its destruction, its LTM will unblock all its correspondents, the killer being normally the last one.

First, we have the issue of how to represent this context change between the task application code and the *_Infanticide* procedure. Second, how do we represent the fine execution control ? The solution imposes to regularly check if a task is killed or not, between the execution of its instructions. A boolean *killed* field is added to the TDs, which is set to true when the LTM changes the task procedure code.

Besides each application task has to manage a program counter (PC). Module *pc* provides the minimal features to execute a sequential instruction, a conditional instruction, an unconditional jump and also subprogram calls. Consider now an application task that calls a system primitive. The code of this primitive must include the fine control too.
That means that we have to write all the primitives to take this into consideration. The caller must take into account the length of the primitive called, in terms of instruction number, for managing the PC. This imposes to do kind of manual assembly.

Fig.4.1.2 shows the code of a two-tasks scenario. The main task creates a child and loops on sending messages to it, and the child receives messages from its father and replies to it.

```
/* main */
if
:: CAL2(0,      _Create,child,TASK2)
:: COND(20,     child==0,0,21)
:: SEQI(21,     request[SIZE]=MAXMSGLENGTH;request[RESULT]=99)
:: CAL4(22,     _Send,rid,request,response,child)
:: GOTO(35,     21)
fi
/* child */
if
:: SEQI(0,      request[SIZE]=MAXMSGLENGTH)
:: CAL3(1,      _Receive,requestor,request,_Father_id())
:: COND(55,     requestor==0,0,56)
:: SEQI(56,     reply[SIZE]=MAXMSGLENGTH;reply[RESULT]=127)
:: CAL3(57,     _Reply,rid,reply,requestor)
:: COND(88,     rid==0,56,0)
fi
```

Fig.4.1.2 — A two-tasks scenario

The priority preemptive scheduling is also introduced by adding data related to ready queues and running processes into module *processor*. A *priority* field is added to TDs too. The most differences appear in module *scheduling*, and *ipinterrupt*. The *_Add_ready* function non only sets the task state to READY but also does it insert the task into the ready queue of

appropriate priority on the task's processor. *_Dispatch* on a given processor *p*, sets the running process for processor *p* by searching the task of highest priority in the ready queues of *p*. Only *_Block* and *_Block_signal_processor* change in module *ipinterrupt*, they take the executing task out of the ready queue.
Interruption masking is totally changed. Each task has an extra boolean field *masked* than *_Enable* sets to false and *_Disable* to true. All the tasks of the system are subject to priority scheduling, even LTMs and tasks executing *_Infanticide*. The overall control on application tasks needs to take that into account, *_Infanticide* and LTMs has to be written with a fine control but LTMs are not concerned by task destruction.
Besides interprocessor interrupt handling needs to be prioritary over the running tasks if there are not masked. The handling does not need the fine control over its execution since it is not interruptible. In summary, on a given processor, here are the running conditions for the various items :

Interprocessor Interrupt Handler :
an interprocessor interrupt is under treatment or
an interprocessor interrupt is pending and
the running task did not mask
LTMs and application tasks :
the task is the running task and
no interprocessor interrupt is in treatment and
no interprocessor interrupt is pending or
the task has masked

Fig.4.1.3 shows the skeleton of the LTMs and Fig.4.1.4 the skeleton of the interprocessor interrupt handlers.

```
proctype _Local_task_manager(byte _Active){
    byte requestor; message(request); ...
    do
    ::  atomic{
            ( _Active==running[processor[_Active]] &&
            !treatment[processor[_Active]] &&
            (!len(fifo[processor[_Active]]) || masked[_Active])) ->
            if
            :: SEQI(0,          response[SIZE]=MAXMSGLENGTH)
            :: CAL3(1,          _Receive,requestor,request,0)
            :: COND(55,         (request[TYPE]==CREATE), 56, 200)
            ...
            :: COND(200,        (request[TYPE]==DESTROY), 201,400)
            ...
            :: COND(400,        (request[TYPE]==SUICIDE), 401,500)
            ...
            :: SEQI(500,        assert(FALSE))
            fi
    od}
```

Fig.4.1.3 — Local task manager

Startup now better conforms to reality. The variable *_Ltm_for_template* is a table that returns the identifier of the LTM in charge of a given task template. The variable *template_priority* returns the priority of a given template. In order to keep the startup pragmatic we prevent the user program from running before all the LTMs on all the processor are blocked on the reception of their first message. User program is started by initializing the first task, that will run on processor 0 (Fig.4.1.5).

```
proctype Int_handling(byte p){...
```

```
do
:: (len(fifo[p]) && !masked[running[p]]) ->
   /* _IP_int(p) */
          treatment[p]=TRUE;
          fifo[p]?candidate;
          /* _Td_service(candidate) */
             ...
          _Dispatch(p);
          treatment[p]=FALSE
od}
```

Fig.4.1.4 — Interprocessor interrupt handler

```
init{...
   atomic{
   Proc_init(); ...
      _Avail_task_number = NB_PROC+1;
      _Ltm_for_template[TASK1]=...;...
      template_priority[TASK1]=...;...
      /* wait for all the LTMs be blocked on the reception of first message */
      (state[1]==RCV_BLOCKED && ...
         state[NB_PROC]==RCV_BLOCKED);
   /* start user program */
         _I_td(NB_PROC+1,0,0,template_priority[TASK1],TASK1);
      _Add_ready(NB_PROC+1)}}
```

Fig.4.1.5 — System startup

4.2. Verification

The verification of task management in Harmony is certainly the most interesting but the most challenging part of this work. With the model presented in §4.1, we are able to write, simulate and verify any scenario involving a bound numbers of application tasks that may use any primitives of the kernel, either for intertask communication or task management running according to a priority based scheduling. It is obvious that it is quite impossible to cover all the possible scenarios, but at least is it possible to write the most basic ones and verify them. We have defined a set of elementary application tasks and we have used them in combination to produce more or less complicated scenarios. Some were already presented in §4.1. As another example, here is a task that immediately suicides :

```
if
:: CAL0(0,        _Suicide)
:: SEQI(18,       assert(FALSE))
fi
```

Fig.4.2.1 presents a two-tasks scenario. The main task creates a child and immediately destroys it. The child attempts to kill its father. When beginning this work, this kind of scenarios were suspected to encompass critical races.

These scenarios allowed us for finding several issues. First, the program restricted to the main task that immediately suicides, revealed that the termination of any application could be not very clean. When a victim has released all its children it sends a SUICIDE message to its LTM and this one requests from the father of the victim to remove it of its offspring tree, but the main task has no father and no control is done for that. At this point, the main task is the only remaining task of the application, all the children are killed, their resources released and it could seem not very important to end in a clean way or not, even crashing the machine. The problem is that the illegal treatment may imply wild stores, affect device registers and result in physical damage.

```
/* main task */                                /* TASK2 */
if                                             if
:: CAL2(0,    _Create,child,TASK2)             :: CAL1(0,    _Destroy,_Father_id())
:: CAL1(20,   _Destroy,child)                  :: ...
:: ...                                         fi
fi
```

Fig.4.2.1 — A two-task scenario for detecting critical races

Second, the scenario of Fig.4.2.1, revealed a lot of problems that were mostly detected by assertion violations. The *_Receive any* primitive should be atomic, namely it should always return when a message is received and with a non zero code, but in fact it was not the case. An assertion to check the requestor in the LTM code was violated, showing that *_Receive any* could return with zero. The problem was easily fixed and the new version of *_Receive any* proved to be atomic. Unfortunately the circumstances under which the bug was found suggest that it may exist much more serious issues inside the LTM. Study is underway.

Several other weird situations were found with this scenario, that are consequences of races between both simultaneous destructions, more precisely between the DESTROY or SUICIDE cases of the LTMs, the *_Infanticide* procedure and the part of *_Td_service* which retrieves the victim out of its queue. The cause was mostly a race between two concurrent executing items that corrupted the state of a task before it could be correctly treated by *_Td_service*, letting a pending interrupt that resulted either in a call of *_Td_service* for a task having five illegal possible states, or in progressing prematurately a task state. This could be considered as a light problem, but it leads to dramatic situations such as the call to *_Suicide* returns, a task is retrieved from a queue where it is no longer in, possibly causing wild stores, a task is queued in two different queues at the same moment, a victim task executing *_Infanticide* never succeeds in sending the SUICIDE message to the proper LTM and thus never really dies, or never sends a DESTROY message for killing one of its children, or never executes *_Infanticide* at all and thus never releases its offspring tree and never dies.

The explanation of these errors would have be very hard without a graphical debugger that we developped with Tcl/Tk [20] as an extension of the X interface of SPIN. During simulation, a window is created for every process of the model. The current instruction for each process is highlighted. Debugging may proceed step by step or in running mode. The execution stops if the simulation ends or if a breakpoint is encountered. There are unconditional, conditional breakpoints and stop conditions. A window allows for the print of variable values.

Conclusions

In this paper we have reported our experience in modelling and verifying a real size operating system kernel whose internal high complexity makes it an ideal candidate for formal methods. Most of the time was spent on producing the models, and only a little on verification. It takes a long time to get acceptable models for validation. Even at this point, we are not sure if they are too simplified or approximate in comparison to the real system, either hiding possible issues or bringing out artificial ones, or if some simplifications could be done, that would increase the model abstraction and thus allow a deeper validation. With this study we succeeded in producing a complete tractable model of Harmony kernel. We are thus able to write any scenario involving a bounded number of application tasks that may use the whole functionality of the system. We only touched upon the verification of the entire system but already succeeded in finding problems in the system. We have already exhaustively verified the intertask communication primitives of the kernel and found a bug that did not showed up during ten years of utilization. Some basic scenarios using the task management primitives revealed at least two other bugs and several critical races. The model we have written also provides a prototype of the system that may be simulated and could serve as a convenient pedagogical tool for Harmony users.

We had an reengineering approach, because the system existed already. We only studied it through its code. The first serious issue we met was to produce an adequate model. The ideal way would have consisted in proceeding by refinements from a very global and abstract model. Unfortunately we could not do that, first because that would have requested a total understanding of the system, which we had not, and second it is easier to apply such an approach to layered systems such as communication protocols. Harmony is a layered system but its kernel is the first layer and it is certainly impossible to apply this kind of decomposition here. For instance, task destruction, which is the most critical issue, uses the features of the whole kernel. It sometimes was possible to isolate concerns and model them separately, such as the relative priority between application tasks, LTMs and interprocessor handlings. We have the feeling that an approach by parts is easier than by refinement. Very often, we realized at validation time that the model was imperfect. It is certainly more realistic to abstract detailed models to check overall properties than produce detailed models form abstract ones. Only after a few iterations, might the ideal method be followed. The second serious issue was to design the tests. Is it possible to produce a complete set of separately validated scenarios that would not miss problematic situations ? When verifying the four intertask communication primitives, we were exhaustive. But it is certainly quite impossible to do an exhaustive verification when we take into account the task management primitives too. The only thing we can do is to define the most interesting scenarios and verify them.

How to validate big models like the ones produced for Harmony is a puzzling question. It is intuitively better to do complete validations of partial models instead of partial validations of complete models, but even with a partial model and a very restricted scenario, we may be faced with a forbidding state explosion. New techniques based on partial order reduction will be soon integrated into SPIN and they will help removing the limits. Nevertheless, to be efficient, such techniques require that the model to be checked present a minimum rate of communication interaction. Not surprisingly, since one of the basic services of Harmony is intertask communication, the resulting models presented a high rate of computation, thus nullifying the benefit of the reduction techniques. We certainly will gain in trying to abstract the detailed models obtained. Some parts of the system do not involve task destruction or only use restricted communication. It is thus possible to produce more abstract models using more interactions. Another serious issue arises when an error is found during validation. If the problem size is small, it is not too difficult to interpret the results. But if it is large, even if SPIN provides all the necessary information, it is really tough to follow the path that lead to the error, especially if it is long and if the model has a lot of variables. We had difficulties to explain the real interesting errors of the task management primitives, but we have begun the development of a graphical debugger on top of SPIN. It already helps in interpreting the validation output. Eventually we have a prototype of a really powerful tool.

Though it was not the initial aim of this study, it was proven that SPIN was very well suited for addressing this problem. Its low expressive power was only an apparent limitation that could always easily be circumvented. More important was the high verification power of validators produced by SPIN. We are convinced that designers would even gain more in using it as a design help and not only in *a posteriori* validations.

Future work should address some part of the questions raised above and some other areas of investigation such as the cache memory management in the kernel. We only used a restricted subset of SPIN's features for our validation and temporal logic should be taken into account for checking more elaborated properties. Most of the errors found tend to suggest they occur for processors with different relative speeds. Nevertheless they give good indications of possible errors for homogeneous processors. Since Harmony runs on boards with homogeneous processors, the errors may sometimes be discarded, but it is often tricky to say without the introduction of time. One could do queueing system modelling with QNAP2 [3] for instance, for doing not only qualitative but also quantitative validations.

Acknowledgments

I am very grateful to Charles-Antoine Gauthier and Morven Gentleman of Software Engineering Laboratory for their comments about this work and contribution to improve the first versions of this paper. I also would like to thank Gregory Duval who extended the X interface of SPIN to allow the graphical debugging of models.

References

1. Andrews D. & Ince D., *Practical Formal Methods with VDM* , McDrawhill Int., 1991.
2. Audureau-Enjalbert-Fariñas, *Logique temporelle, semantique et validation des programmes paralleles*, Masson, 1990.
3. Badel M, Duong D., Eyraud S., *The New Simulation Facilities in QNAP2*, Proc of ESS91, Het Pand, Belgium, November 1991.
4. Birrell A.D., Guttag J.V., Horning J.J. & Levin R., *Synchronization Primitives for a Multiprocessor*, A Formal Specification, Operating Systems Review, 21(5), 1987.
5. Doeppner T.W. & Giacalone A., *A Formal Description of the UNIX Operating System*, Proc. of 2nd ACM Symp. on Principles of Distrib. Syst., Montreal, Canada, August 1983.
6. Gauthier C., *Design and Implementation of Realtime Operating System on Thinwire Multiprocessor*, PhD candidacy paper, Ottawa University, June 1993.
7. Gentleman W.M. , MacKay S.A., Stewart D.A. & Wein M., *Using the Harmony Operating System, Release 3.0*, NRC/ERA-377, National Research Council of Canada, Ottawa, February 1989.
8. Gerhart S., Craigen D. & Ralston T., *Experience with Formal Methods in Critical Systems*, IEEE Software, January 1994, pp.21-39.
9. Glade B.B., *A temporal logic to Promela "never" clause converter*, EE594 Final Project, Cornell University, Electrical Engin. Department, 3 May 1991.
10. Gotzhein R., *Temporal Logic and applications - a tutorial*, Computer Networks and ISDN Systems 24(1992) 203-218.
11. Gouda M.G., *Protocol verification made simple : a tutorial*, Computer Networks, 25(9), April 93, pp. 969-980.
12. Holzmann G.J., *Algorithms for automated protocol verification*, AT&T Technical Journal Jan/Feb, 1990
13. Holzmann G.J., *Basic Spin Manual*, AT&T Bell Laboratories, March 1994.
14. Holzmann G.J., *Design and Validation of Computer Protocols*, 512 pgs, ISBN 0-13-539925-4, Publ. Prentice Hall, (c) 1991 AT&T Bell Laboratories.
15. Holzmann G.J., *Design and validation of protocols : a tutorial,* Computer Networks, 25(9), April 93, pp. 981-1017.
16. Jones C.B., *Systematic Software Development Using VDM*, Prentice Hall Inter., 1990
17. Lai R. & Jirachiefpattana A., *Verification of ISO ACSE protocol specified in Estelle,* Computer Communication 17(3), March 1994.
18. Li Y., *The Harmony operating system described by Petri nets*, Master thesis, Carleton University, 1987.
19. NRC, *Harmony Operating System, Release 4.0*, Application Note 18, National Research Council of Canada, Ottawa, Ont., March 1993.
20. Ousterhout J., *Tcl and the Tk toolkit*, Addison-Wesley, 1994.
21. Pêcheur C., *Using Lotos for specifying the CHORUS distributed operating system kernel*, Computer Communication 15(2), Mars 1992.
22. Pike R., Presotto D., Thompson K. & Holzmann G., *Process Sleep and Wakeup on Shared-memory Multiprocessor*, EurOpen'91 - Tromso
23. Ramsey N., *Correctness of Trap-Based Breakpoint Implementations*, Bell Communications Research, November 1993.
24. Spivey J.M., *Specifying a Real-Time Kernel*, IEEE Software, 7(5), 1990.
25. West C.H., *Protocol Validation - principles and applications*, Computer Networks and ISDN Systems 24, 1992, pp. 219-242.