Modeling and Verification of the RUBIS μ–Kernel with SPIN

Gregory DUVAL [1,3]
Jacques JULLIAND[2]

[1] *Laboratoire de TéléInformatique, Ecole Polytechnique Fédérale, CH-1015 Lausanne, Switzerland*
[2] *Laboratoire Informatique de Besançon, Université de Franche-Comté, 16 route de Gray, F-25030 Besançon Cedex, France*

This experience was lead at the Laboratoire Informatique de Besançon and its aim was the modeling and verification of the RUBIS kernel, a multitasking multiprocessor operating system. An other aim of this study was to find out a method to formalize and to verify such systems. The first results are given. A modeling approach is used and we formalize the models of the system, the scenarios and the properties in Promela and linear temporal logic. Two kinds of models are produced, the so-called *"abstract models"* and *"detailed models"* which are model-checked with the SPIN tool. An exhaustive verification of the intertask communication features of RUBIS was carried out. It revealed several problems in the management of error return codes. This paper gives hints of a method to construct and formalize the models of an operating system such as RUBIS.

1 - Introduction

Formal specifications of application software are becoming more and more common. However it is not a common practice for operating systems, some attempts have been made for UNIX or real-time kernels such as HARMONY [2].

Our objective was to find a method to model and verify the entire intertask communication features of RUBIS [12]. We used formal methods to verify some properties of the RUBIS kernel. The complexity of concurrent systems led us to prefer an automatic verification approach. There are many common points with the domain of communication protocol design and validation. Specification and verification of communication protocols is a common practice. Communicating finite state machines seemed to be the best choice to model our system. So we decided to use PROMELA as specification language and the SPIN tool to check the large state spaces of the system. Even if Promela's expressive power is rather low and provide only a few basic data types and only very primitive type constructors, it offers the possibility for expressing safety or liveness requirements by logical assertions or linear temporal logic expressions. A model can be animated by the Spin simulator or verified by the validator with different coverage degrees depending on the problem size. The properties are checked during the simulation or the verification step and if an error occurs it is possible to run the simulator again and look at each state to find the problem.

In the next sections we briefly present the RUBIS system, we explain the approach used to produce the models of RUBIS and the scenarios used for the verification, finally we show how these models and scenarios were used to verify properties.

---

[3] This research was done at the Laboratoire Informatique de Besançon, Université de Franche-Comté, 16 route de Gray, F-25030 Besançon Cedex, France

2 - Rubis Kernel

RUBIS [12] is a multitasking multiprocessor operating system developed by TELMAT company and is installed on multinode machines, based on Transputers processors. RUBIS is an operating system based on a micro-kernel and system servers. The processes can be located on the same or on distinct processors.

A port can be created by a process using the *PortCreate* primitive and must be published through the system, using *PortPublish* in order be known of the other tasks on all the processors. When a process wants to send data through the system, it has to request a free Port to the appropriate port manager using the *PortLookup* primitive. Synchronous or asynchronous communications are possible with *PortSend*, *PortReceive*, *PortAsend* and *PortAreceive*. When the communication is over, the Port can be unpublished and deleted using the *PortUnpublish* and *PortDelete* primitives. For more details about RUBIS see [12]

3 - General approach

Such a work may be divided into several steps. First of all, we have defined properties of the system which had to be checked. Then models were constructed with different degrees of abstraction. The first models are called abstract models. They just are made from specifications and they do not take the system implementation details into account. Abstract models allowed some verification of the system specifications and a better understanding of the system functionality's. When this kind of model is verified, an other kind of model may be built. They are called detailed models and, this time, they take implementation of the system into account. They have to be an exact reflection of the system behavior. Finally, scenarios have to be defined to check properties.

4 - Abstract models

This kind of models is built by using behavioral and functional specifications of the system and its user guide manual. Abstract models are a user's view of the system. It is impossible to give a systematic approach for constructing an abstract model because the support documents used are informal.

For instance, a port can be defined as in Fig. 4.1. An abstract representation just uses a channel and two boolean variables to synchronize the sender with the receiver when processes are using synchronous communications.

```
typedef  Port_table{
        chan messages = [MAX_MESSAGE] of {byte};
        bool sender_ready;
        bool receiver_ready;
};

Port Port_table[NB_PORT];
```

Fig.4.1 — Abstract Representation for Communication Ports

In Fig 4.2 we can see how the communication primitive *PortSend* which allows to send data synchronously is modeled. The primitive parameters are the port number, the message to send and an error return code. The boolean variables *sender_ready* and *receiver_ready* are used to synchronize the receiver and sender processes.

```
#define PortSend(port_num,message,error)\
        if\
        ::(empty(port[port_num].p) && port[port_num].receiver_ready) ->\
                port[port_num].sender_ready=TRUE;\
                port[port_num].p!message;\
                port[port_num].sender_ready=FALSE;\
                error=OK;\
         :: error=ERROR;\
        fi
```

Fig. 4.2 — Abstract Representation for the PortSend Communication Primitive

5 - Detailed models

This kind of models must faithfully reflect the system behavior. Detailed models are built by using the C code of the system and the high level specifications. For this step it is possible to give transformation rules which can be an assistance to produce models. For instance, if we want to extract an interaction model by rendezvous from an Ada program, we can give transformation rules such as :

$$\frac{e \mid\!- B \rightarrow B'}{e \mid\!- taskbody(T,B) \rightarrow proctype(T,B')}$$

$$\frac{e \mid\!- P \rightarrow P'}{e \mid\!- taskentry(T,E(P)) \rightarrow chan(T_E,0,P')}$$

The first rule states that the body an Ada task *T* may be modeled as a Promela process whose body *B'* is the modeling of body *B*. Similarly, a task entry *E* with parameters *P* may be modeled with a synchronous Promela channel carrying messages of type *P'* (*P'* being the representation in Promela of *P*).

As many rules as possible Ada structures need to be defined. The set of all these rules is the specification of a support tool for automatically extracting an interaction model of a program. For RUBIS it is not so easy to give transformation rules because it is implemented in C and Assembler. Even if it is more difficult, it should be possible to find such rules, the point being the identification of the primitive synchronization mechanisms. In RUBIS this is achieved with two specific assembler statements used for establishing a rendezvous between two processes.

Refinement relation between abstract and detailed models can be shown by using processes equivalence techniques [14].

6 - Scenarios and Verification

For building scenarios, two kinds of combinatory are applied successively. First topological combinatory are investigated. Fig 6.1 shows a particular scenario where synchronous and asynchronous communications are mixed between four tasks on two processors. Thus it is possible to check at the same time mono and multiprocessors communications with the two kinds of transmission (synchronous and asynchronous). If such a scenario does not produce any error, we may think that the communication mechanism is error free.

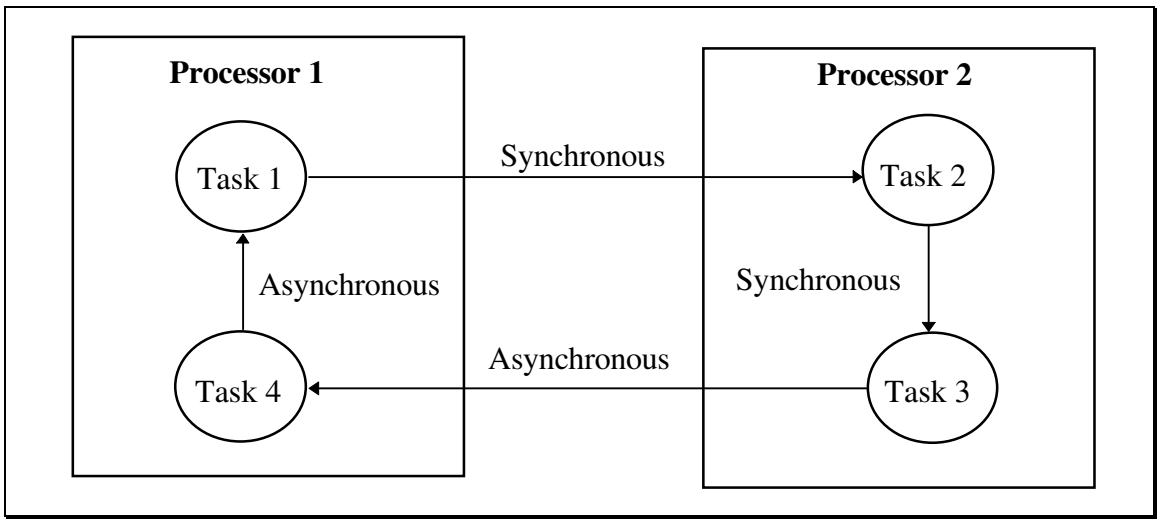

Fig 6.1 - Mixed Synchronous and Asynchronous Communications

Second, combinatory of communication situations are studied. For building the scenarios one combines all primitives of the system so as to get the most exhaustive tests set as possible for checking defined properties and to attempt to detect unreachable code. We can list the different communication situations in which a task may be involved. A task may attempt to communicate with a non existing task, an existing task that is not ready to communicate, an existing task that is ready to communicate, an existing task that is already communicating, or it may attempt to communicate through a port which is already in used or which does not exist.

All scenarios which have been verified with abstract models are checked with detailed models too. Properties which can be checked with abstract models are only those which are known by users.

As an example of safety requirement of abstract models, we may consider the fact that Ports need to be in a sound state before being reused for another communication. For the verification of this requirement, some extra variables (*Port_state*) are added to the specification that record the ports state and the following linear temporal logic property is expressed. It intuitively means that for all Ports *p* numbered from, say *0* to *m*, if *p* is created (with the *PortCreate* primitive) then the related channel needs to be empty.

$\Box(P(0) \Rightarrow Q(0) \wedge (P(1) \Rightarrow Q(1)) \wedge ... \wedge (P(m) \Rightarrow Q(m))$

where :

$P(p) = (Port_State[p] = CREATED)$

$Q(p) = (empty(Port[p].messages))$

Fig.6.2 shows the Promela definitions which implement that claim for *m=5*.

```
#define  P(p)  (Port_State[p] = CREATED)
#define  Q(p)  (empty(Port[p].messages)
#define R      ( (P(0)->Q(0)) ∧ (P(1)->Q(1)) ∧... ∧ (P(5)->Q(5)) )
/* ![]R */
never{
        do
        :: R
        :: !R
        od
}
```

Fig 6.2 - Never Claim for Ports Verification

Many other properties have to be checked, in particular the preservation of message or port destruction sequences.

For detailed models, many other scenarios depending on the implementation of the system have to be verified. These scenarios serve, for instance, to check the correct behavior of system servers in charge of actual data exchanges.

Fig 6.3 shows the Port manager code and we can see that there are two possible states for a Port Manager. The *INIT* state means that the manager is waiting for a request from any task and the *WORK* state means that the manager is treating a request.

```
proctype Portman(byte pid; int processor){
        T_Message message;
        bool request;
        ...
        State = INIT;
        do
         :: link[processor]?message;
                request=TRUE;
                ...
                if
                :: (message.request == PORT_SEND) ->
                        State = WORK;
                        ...
                        State = INIT;
                :: (message.request == PORT_RECEIVE) ->
                        State = WORK;
                        ...
                        State = INIT;
                :: else -> assert(0)
                fi;
                request=FALSE;
                ...
        od;
}
```

Fig 6.3 - Port Manager

The properties we want to verify are :

P1 : The manager always eventually does a treatment when a request is send to it

P2 : The manager always return in its initial state after a treatment, namely it will never stay blocked in a treatment sequence.

So we have two properties to check. The first one is that after having received a request in an *INIT* state we must eventually have a *WORK* state. And the second one is that after a *WORK* state we must eventually have an *INIT* state.

$P1 : \Box(((State = INIT) \wedge request) \Rightarrow \Diamond(State = WORK))$

$P2 : \Box((State = WORK) \Rightarrow \Diamond(State = INIT))$

The conjunction of *P1* and *P2* is :

$\Box(((State=INIT)\wedge request) \Rightarrow \Diamond(State=WORK)) \wedge \Box((State = WORK) \Rightarrow \Diamond(State=INIT))$

which is equivalent to :

$\Box((((State=INIT) \wedge request) \Rightarrow \Diamond(State=WORK)) \wedge ((State = WORK) \Rightarrow \Diamond(State=INIT)))$

We know that *State* may only take two values, i.e. $State \in \{INIT, WORK\}$, so we have :

$(State = INIT) \Leftrightarrow \neg (State = WORK)$

and thus the conjunction of *P1* and *P2* becomes :

$\Box(((State=INIT) \wedge request) \Rightarrow \Diamond\neg (State=INIT)) \wedge (\neg (State=INIT) \Rightarrow \Diamond(State=INIT)))$

This claim can be expressed in Promela by using the following simplified *never* claim :

```
never{
        do
        :: skip;
        :: ((State == INIT) && request) -> goto accept_S;
        :: !(State == INIT) -> goto accept_no_S;
        od;
accept_S :
        do
        :: (State == INIT);
        od;
        accept_no_S :
        do
        :: ! (State == INIT);
        od
}
```

Fig 6.4 - Liveness Property checked on Port Manager

Many others properties have to be checked in detailed models. In particular properties for FIFO buffers which have to be empty at the end of a communication. Others scenarios are constructed to fill all buffers of the system. Thus it is possible to check the system's behavior when it is overflowed.

7 - Results

Different kinds of error have been detected. Lots of error were related to error return codes. These error codes were not returned to the calling task and therefore the task did not know that a problem occurred. So the system could continue its work without taking the error into account and from that could result a system crash. Errors concerning memory management were detected too.

8 - Conclusion

This work shows that operating systems are good candidate for formal methods. The complexity of operating systems necessarily breeds conception errors which are difficult to detect manually but are rather easily found with automatic methods. Several models of RUBIS were produced with different levels of abstraction. The more detailed models allowed an exhaustive verification of the intertask communication primitives of the RUBIS kernel.

Because the system already existed, we had a re engineering approach. It could be useful to have tools which can automatically produce models by using the source code of such a system. Even if this tool could only produce interaction models, it would be a great help.

This work has shown once more that SPIN was very well adapted for verifying operating systems and that a high-level formalism such as temporal logic may be fruitfully used in that context since it is fully supported by SPIN. Even if, and perhaps because, Promela expressive power is limited, it can produce very high verification possibilities.

9 - Acknowledgements

We are very grateful to Thierry Cattel of Laboratoire de TéléInformatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland for his comments about this work and contribution to improve the final version of this paper.

10 - References

1. Burch E. M., Clarke K.L., McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 1020 *states and beyond*. Information and Computation, 98(2) : 142-170, june 1992
2. Cattel T. : *Modelling and Verification of Multiprocessor Realtime OS Kernel*. Proc. of FORTE 94, Berne, October 1994.
3. Clarke O. Grumberg and D. Long : *Verification Tools for Finite-State Concurrent Systems*, Carnegie Mellon University.
4. Derek Coleman and David Skov : *Analysis and Desing for Concurrent Object Systems*. Hewlett Packard Laboratories. Bristol BS12 6QZ, UK.
5. Duval Gregory : Contribution à la définition d'une méthode de vérification de logiciels réactifs appliquée au système RUBIS. DEA report, Laboratoire Informatique de Besançon - France.
6. Holzmann : *Basic Spin Manual*. AT&T Bell Laboratories, Murray Hill, New Jersey.
7. Holzmann : *What New in Spin 2.0 (Draft)*. AT&T Bell Laboratories, Murray Hill, New Jersey.
8. Holzmann : *Design and validation of protocols. Prentice Hall Int., 1991*.
9. Z. Manna and A. Pnueli : *The Temporal Logic of Reactive and Concurrent Systems - Specifications*. Spinger - Verlag.
10. McMillan. *Symbolic model checking : An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.
11. Pnueli : *Applications of temporal logic to the specification and verification of reactive systems : A survey of current trends*. Department of Applied Mathematics. The Weizmann Institute of Science. Rehovot 76100, ISRAEL.
12. RUBIS - Run time Basic Interface Software - Reference Manual & User's Guide. Société TELMAT Multinode. 1993-1994.
13. J Rumbauch & al. : *O.M.T - Modélisation et Conception Orienté Objet*. Masson - Prentice Hall.
14. Van Glabeek, R.J. : *Comparative Concurrency Semantics and Refinement of Actions*, PhD thesis, Free University of Amsterdam, May 1990.