An Example of Use of Formal Methods to Debug an Embedded Software

André Arnold[1], Didier Bégay[1], Jean-Pierre Radoux[2]

[1] LaBRI, Université Bordeaux I
351, cours de la Libération
F-33405 Talence, France
[2] SERLI-Informatique,
avenue du Téléport
F-86960 Futuroscope, France

Abstract. This article releases an industrial experiment of using formal methods to analyze and to debug a system that was shown erroneous by testing. After presenting the industrial context of the experiment, it details the modelling process and the interpretation of the results, in conjunction with the designers of the system.

1 Introduction

1.1 The EUROTRI project

The EUROTRI project was aimed by Schlumberger-Industries to define, conceive, design and develop a static electricity meter with large abilities of tariff programming and distant measuring. For commercial and industrial reasons, short time of development and low cost of production (through the use of components at frontier of their technical limits) were the challenges. It was also the first time a software-based product was designed and developped in this industrial center: the previous products had been conceived as electronical, even if they used some software.

The main idea was to avoid iteration of any development step, and to meet the deadlines by a straightforward way. This implied to use at every step of design and development all possible means to validate it definetely. This process was usual for the electronics engineers, and we convinced them easily that an equivalent for the software would be formal methods. So it has been decided at the very beginning of the project to use formal methods in most steps of it, from feasability study to reception testing.

For different reasons, including readability of transition systems by electronic engineers and availability of a powerful model-checker, the Arnold-Nivat model and the MEC tool have been selected, in conjunction with the language Ada and the Rate Monotonic Analysis.

The "Concurrency Semantics" team of LaBRI became involved in the design and development process as an extra team among the usual industrial ones: hardware, software, marketing and industrialization, with a general interest on the use of formal methods along the project.

The different aspects of this collaboration, and peculiarly the different ways formal methods have been combined and applied in the project, have been presented in [4].

The study we present here is illustrative of the power of the formal methods. The application described here had been thought simple enough not to deserve formal methods in its design (furthermore, this would spare academy resource in the project). On the bench the application did not work properly, although not clearly enough to point out where the default was lying. The lack of rational explanation made the engineers report the phenomenon to our team. Happily we happened to explain "intellectually" what was going on, as it is described hereby, but *during the project, this has been the only solution to this important problem related to security.* This emphasizes the ability of formal methods to intervene successfully and a posteriori in an application developped without them, in the tests step.

So this is a real industrial application, solved by academy people.

A part of the project, related to electronics, the study of the physical layer of the two processors, at the interface of hardware and software, has been released in [5].

The interesting way the Synchronized Transition Systems approach and the Rate Monotonic Analysis have been coupled to study the temporal aspects of the embedded system has been explained in [1].

The whole work has been presented as a thesis by Jean-Pierre Radoux [8].

The consequences of this project on the way Schlumberger-Industries considers system development will certainly impact its R&D structure.

1.2 The OUCABO system

The detailed design of this system, and some informations related to implementation details, were the basis of our approach of the OUCABO system.

Dealing with an electricity meter, it is a basic feature to iteratively measure N numerical values (such as voltage, intensity and phase) and to process them in the flow. At a given time t, one among the N analog signals is input in the measuring software, by the mean of a 1-in-n multiplexer, driven by an external (external to the software) counter. So we have two counters referring to the same value: an *external* (hardware) counter, selecting the input signal, an *internal* (software) counter, processing the input signal, both of them being controled by the same clock.

The first problem we have to deal with is that parasites can impact the links between the common clock and both devices, and thus desynchronize both counters, making the software process erroneously input values.

This can be handled either by identifying each sample transmitted to the software or by periodically synchronizing both counters. The first solution was not relevant for technical reasons, and it has been designed that an output port would be used by the software counter to reset the hardware counter.

In such an embedded system as the electricity meter, various processes have to be activated to ensure the fiability of the apparatus. One of them is the OUCABO system. This system requires a device to be periodically activated and tested. *The second problem is that the solution to first problem left no output port available to activate this secondary device, nor input port to test it.* The situation on the output port has been handled using the leftmost carry bit of the hardware counter to activate the device : if the software counter does not reset the hardware counter in due time, there occurs an arithmetic overflow and the device is activated.

This solution had been conceived and implemented ignoring formal methods, as an electronic hack to answer a very simple, low-level problem.

1.3 Debugging the OUCABO system

This part of the system had been developped and implemented, and tested using laboratory equipment: same signals were simultaneously measured by the system and the testers. Unfortunately, it happened that a very seldom measure difference occurred between both measures; at first glance, the origin could be hardware, software or a *testing protocol error.*

Testing such rare behaviours may last a long time, and in order to spare time, a model-checking approach was accepted. Then, using the detailed design and some implementation details, * the system has been modelled using synchronized labeled transition systems. The error and the error rate could be confirmed on a study of synchronized transitions, thus identifying the source of the erroneous behaviour. The bug was fixed in the model and after verification, the system could be efficiently and safely corrected.

1.4 Content of the paper

Instead of reporting the experience as it was, we choose in this paper to construct step by step, in a pedagogical order, the system to model.

2 Synchronized products of transition systems

We model any behaviour using labeled transition systems (LTS), ie: uniquely named states linked by transitions, each of them wearing a non-null string of characters called its label, and due to model what triggers the transition from source state to target state. As same phenomenon can trigger different transitions depending of current state, different transitions can wear same label.

A parameterized LTS is a LTS where boolean properties are defined on states and transitions, and operators allow computation of such properties as deadlocks of loops, etc... using an adapted version of Tarjan's algorithm and a μ-calculus engine ([6, 7]).

Interactions between behaviours is expressed in our synchronous model using correlation of labels: behaviours triggers freely, provided they respect a set of synchronization constraints, expressed as a set of vectors of firing configurations

of labels. So we construct the synchronized product of original LTS's as a LTS where states are vectors of original states, and transitions are labeled by vectors of original labels, restricted to the synchronization constraints. This synchronized transition system models the global behaviour under study. We can apply on it the usual computing of boolean properties.

Both construct of the synchronized product and computing of boolean properties are implemented in the MEC tool.

A complete description of our model and of the power of expression of the properties will be found in [2], and a shorter one illustrated by some demonstrative examples using MEC is given in [3].

3 Hard and soft counters

The system contains a hardware counter modulo N activated by ticks of a clock and able to be reset to 0 by an input signal. If N ticks occur without any reset, a carry bit is set. Each tick of the clock generates an interruption: the interruption routine increments a counter, called **channel** and initialized to 0, when its value is strictly less than $N-1$, and resets it to 0 and sends a reset signal to the hardware counter when it has value $N-1$. The code of this interruption routine is thus

if channel $< N-1$ **then increment(channel) else send(reset); channel:=0; endif**

3.1 Modeling the hardware counter

The first thing to do is to give a definite value to N. Recall this value is inversely related to the error ratio. We assume that N has a lesser value than the actual value, say 4, and it will not be difficult to convince oneself that is a quite admissible simplification with respect to the verifications to be done. The hardware counter can have its carry bit set or not. In order to make things easier, we model activity of the carry bit during $3 \times N$ ticks.

Therefore, the hardware counter can be modeled by a transition with thirteen states. A state of the counter is encoded by a two-digit number: the second digit (0,1,2, or 3) represents the value of the counter and the fist digit counts the number of carries since its last reset. Two actions can change the state of the counter:

- a tick of the clock, denoted by `t`,
- a reset signal, denoted by `reset`.

We also use the null action denoted by `e` to express that something else may occurs that has no effect on the counter.

In MEC, such a transition system is described by

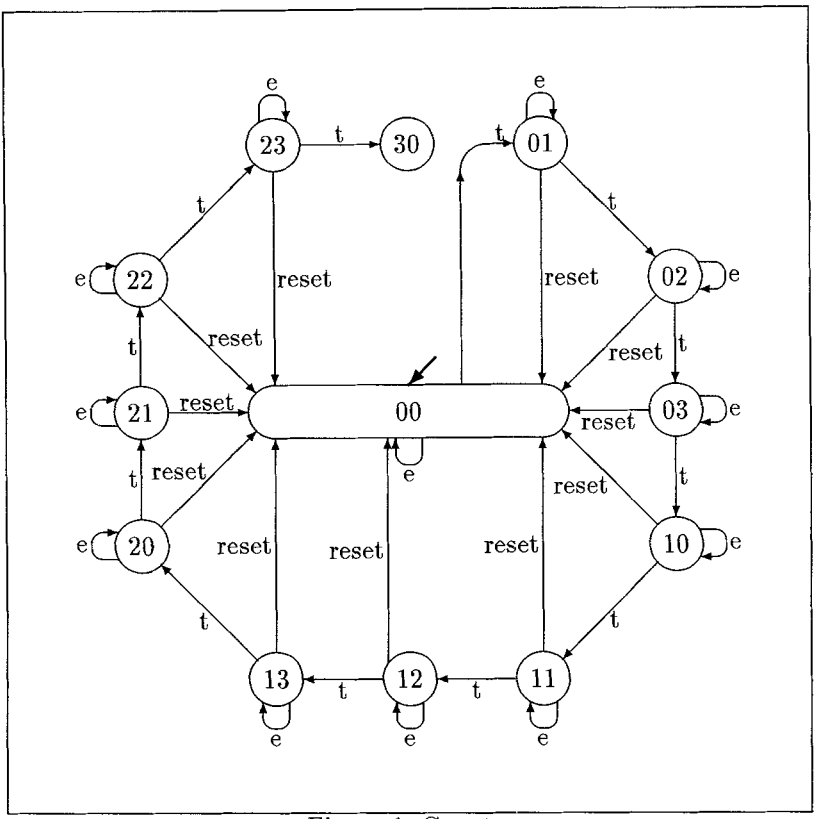

Figure 1: Counter

3.2 Modeling the software counter

The software counter is similar to the hardware counter: its states are the values of **channel** and when a tick occurs (action denoted by `t`) this value changes. However, when this value is 3, a reset signal is immediately sent to the hardware counter before the value becomes 0 (action denoted by `reset`). Therefore, we get the following system.

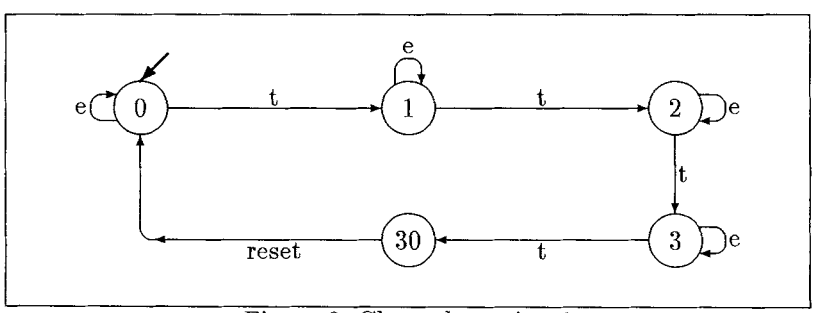

Figure 2: Channel, version 1

Remark that when the channel receives a tick in state 3, it goes into an intermediate state 30. In this state it is not allowed to wait (there is no "e-loop"), it has to immediately execute **reset** and to go into state 0.

3.3 Synchronization constraint

We consider the system composed of the counter and the channel. When a tick occurs, both components react simultaneously by executing a transition labeled by the action t. When the interruption routine sends a reset signal by **reset**, the counter simultaneously receives this signal and executes **reset** too. Finally both components can stay idle together. This synchronization constraint is described :

counter	*channel_v1*
t	t
reset	reset
e	e

The synchronized product of this system is

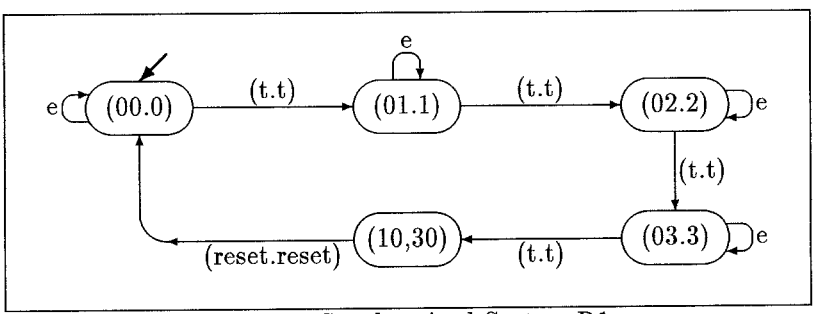

Figure 3: Synchronized System R1

and one can see that, excepted when the channel is in its intermediate state 30, the values of the counter and of the channel are the same.

4 Introduction of a mode

Actually, for some reasons relating to the activity of the secondary device via the carry bit, the above system can be in two modes, special and non special. The mode becomes special when an external and uncontrollable event, called **start**, is generated by the environment of the system. The only difference between the two modes is that no reset signal is emitted when in special mode, and thus the carry bit will be set to activate the secondary device:
if channel $< M - 1$ **then increment(channel)**
else if mode is not special then send(reset); channel:=0; endif

We model the mode by the transition system:

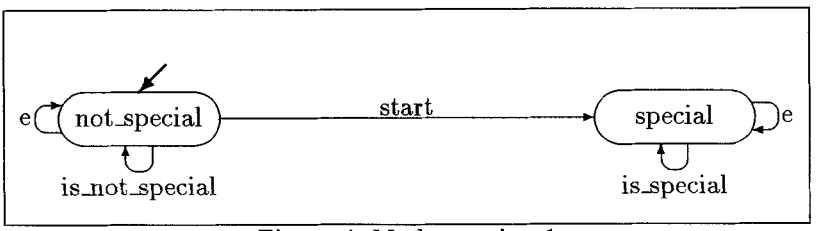

Figure 4: Mode, version 1

The action `start` switches the mode from non special to special. The actions `is_not_special` and `is_special` allows to know what the mode of the system is.

We now have a system with three components: `counter`, `channel_v1`, `mode_v1`. Let us describe the interactions between these components.

When a tick occurs it is received by `counter` and `channel` and has no effect on `mode`; this is expressed by the synchronisation vector `(t.t.e)`. When the mode becomes special, nothing happens to the counter and the channel: `(e.e.start)`. When the channel is in state `30` it emits a reset signal if and only if the mode is not special. Instead of modelling this situation, we model an equivalent situation: this signal is emitted in any case but it is received by the counter only if the mode is not special, that is expressed by the two vectors `(reset.reset.is_not_special)` and `(e.reset.is_special)`. Thus we get the system:

counter	*channel_v1*	*mode_v1*
t	t	e
e	e	start
reset	reset	is_not_special
e	reset	is_special
e	e	e

whose synchonized product is:

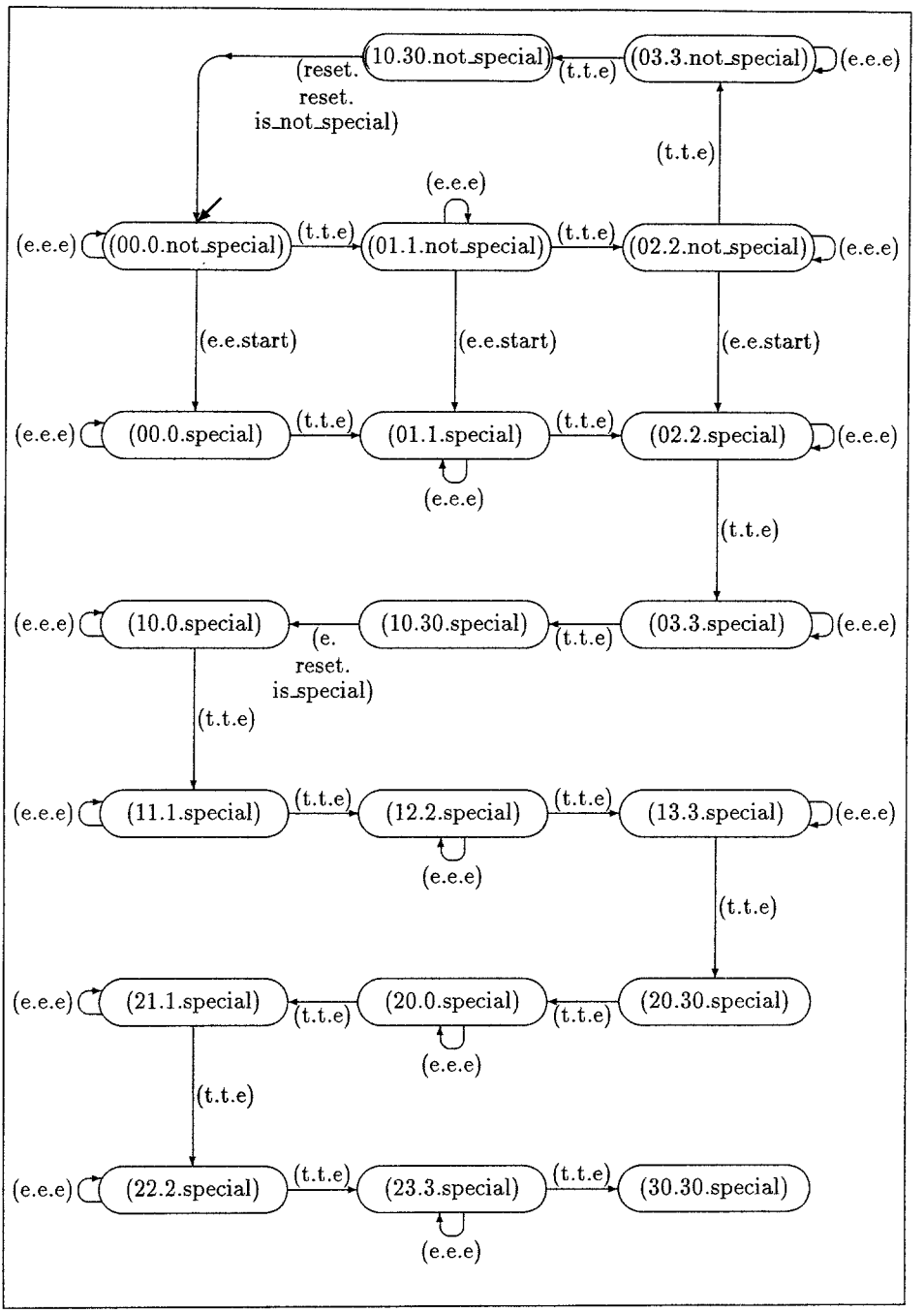

Figure 5: Synchronized Product R2

5 Activation of a secondary device

When the mode is special, the channel 1 of the multiplexer is used to read a value produced by the device that has been activated by the carry bit of the counter. We use an action "read" to explicit the fact that a value is read on this channel coming from the device.

The interruption routine becomes
case
$0 <$ **channel** $< N - 1 \Rightarrow$ **increment(channel);**
channel $= 0 \Rightarrow$ **channel:=1; if mode is special then read endif;**
channel $= N - 1 \Rightarrow$ **if mode is not special then send(reset) endif; channel:=0;**
end
that is modeled by slightly modifying the previous model: adding a new intermediate state `01` between `0` and `1`.

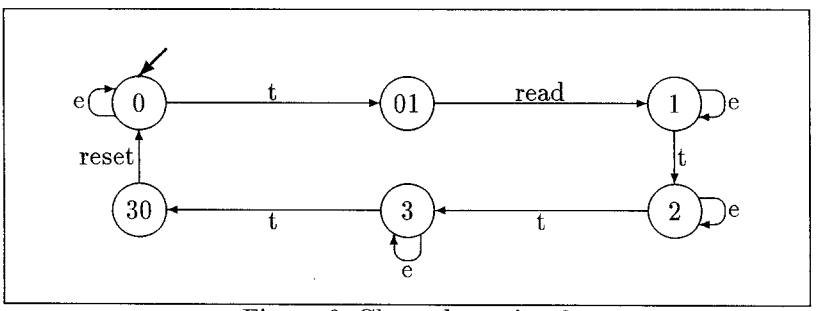

Figure 6: Channel, version 2

Indeed when is special mode the secondary device is activated twice and read twice. We introduce a transition system that counts how many times the device has been read. This transition system is:

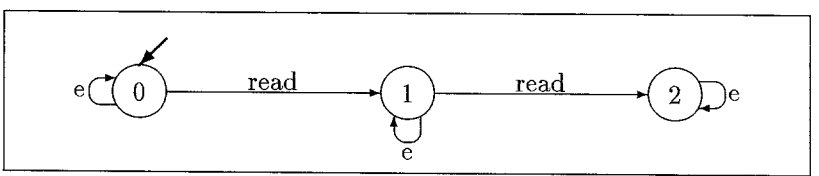

Figure 7: Device

As for the reset signal, the "read" action will be taken into account depending on the mode is special or not. Thus our system now is:

device	*counter*	*channel_v2*	*mode_v1*
e	t	t	e
e	e	e	start
e	reset	reset	is_not_special
e	e	reset	is_special
e	e	read	is_not_special
read	e	read	is_special

The corresponding synchronized product has 36 states and 38 transitions.

6 Fixing a bug

To behave correctly the system must meet the following property:

> Just after any reading of the device, the number of activations of the device (i.e., the first digit of the state of the hardware counter) must be equal to the number of readings (i.e., the state of `device`).

The designers of the code were almost sure that this property was true. Experiments showed that indeed it was not, and we introduced the modelling above just to understand how and why it happened.

6.1 Finding the cause of the errors

First, we look at the states reached after any reading of the device by computing

```
s := tgt(!label[1]="read");
```

This set contains four states: `e(1.01.1.special)`, `e(1.11.1.special)`, `e(2.11-.1.special)`, and `e(2.21.1.special)`. The second and the fourth ones satisfy the above requirement, while the first and the third do not. This is indeed the bug in the system and is consistent with the experimental observations.

Looking at a path leading to one of those states, one finds the one:

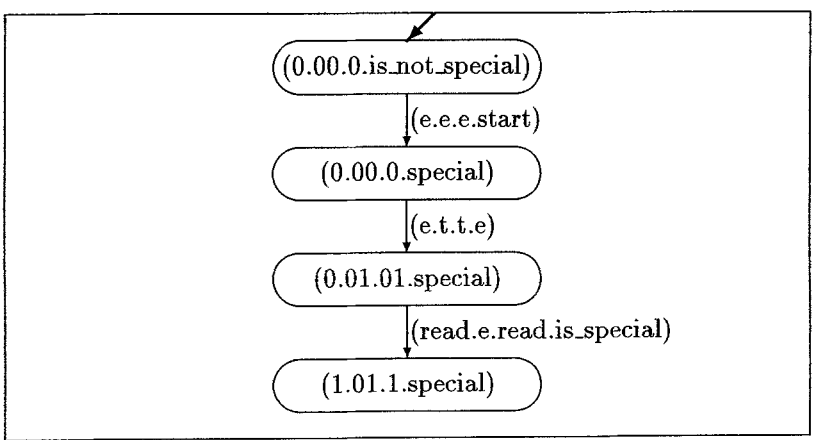

Figure 8: Path

Then one guesses that the problem is probably caused by the mode switching from non special to special while the channel has value 0. To check this guess we define the set of transitions where the mode changes and the channel is 0,

```
cause := !label[4]="start"/\rsrc(!state[3]="0");
```

and we compute the set of states that can be reached without using these transitions (indeed, there is only one)

```
reachable:=reach(initial,*-cause);
```

The intersection of the sets `s` and `reachable` contains only two states, `e(1.11.1.-special)` and `e(2.21.1.special)`, and the two incorrect states become unreachable.

6.2 Some predictions

We know that the observed errors are due to the fact that the system switches to special mode when the channel has value 0. Since the start event is randomly distributed, the probability it occurs when channel has value 0 is 1/4 and the misbehaviour of the system should be observed with a frequency of 1/4. The frequency $1/N$ was experimentally observed, with respect to the actual value N of the software counter. What is more, the fact that the incorrect result of the actual system is caused by one additionnal reading of the inactivated device (when start event arrives on channel 0) was also experimentally confirmed.

6.3 The correction

A solution for avoiding the problem is to force the system to switch to special mode when the channel has not value 0. But there is no possible control on the start event. On the other hand, it is possible to control the time when the mode switches to special. Thus, we "desynchronize" the two actions: when the start signal is received, the mode goes into a special state informing the system that this event has happened, and the system goes from this state to the special mode under control of the interruption routine by the action named `switch`. Remark that this action can be executed only when the mode is in state `request`.

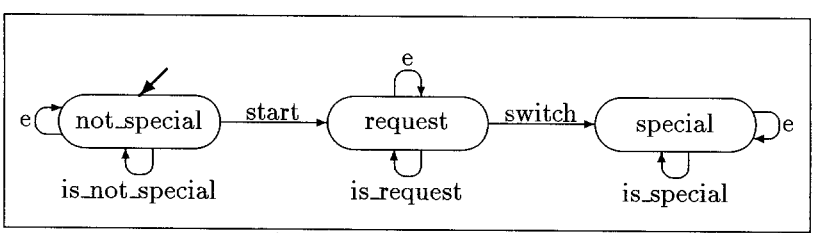

Figure 9: Mode, version 2

When setting the channel from 0 to 1 the interruption routine makes the device to be read if the mode is special and do nothing if the mode is not special. If the mode is in the intermediate state `request` it switches the mode to `special`. That new value of the mode will have effect at the next `activate`, as if the start event has occurred when the channel has value 1, 2, or 3! Of course, this cause a delay of four additionnal ticks before switching to the special mode, but engineers found it quite acceptable.

The new constraint is:

device	*counter*	*channel_v2*	*mode_v2*
e	t	t	e
e	e	e	start
e	reset	reset	is_not_special
e	e	reset	is_special
e	reset	reset	is_request
e	e	read	is_not_special
read	e	read	is_special
e	e	read	switch

The corresponding system has 28 states and 31 transitions. Now, the set

```
s := tgt(!label[1]="read");
```

contains only the two correct states `e(1.11.1.special)` and `e(2.21.1.special)`.

Finally it is easy to rewrite the interruption routine so that it conforms to the model above:

case
$0 <$ **channel** $< N - 1 \Rightarrow$ **increment(channel)**
channel $= 0 \Rightarrow$ **channel** $:=$ **1;**
if mode = special then read;
elsif mode = request then mode := special;
endif;
channel $= N - 1 \Rightarrow$ **if mode is not special then send(reset)**
endif;
channel $:=$ **0;**
end

7 Conclusion

We presented a use of formal methods in an unusual context: at testing time, to find out the cause of an error in a system, with respect to the difficulty to show up some very seldom behaviours on the bench. As model-checking aims to exhibit all possible behaviours, whatever their individual probability may be, this approach is extremely powerful in applications showing very seldom misbehaviours. The

fault of the system was confirmed, and a precise diagnostic set. This would not have been possible (and they tried to do so) in a reasonable delay using testing techniques.

Notice however we did not intend to model and verify an implementation: we specified *a posteriori* the system by abstracting relevant elements with respect to the observed misbehaviour.

The efficiency of the use of synchronized labeled transition systems for modelling and checking electronic components has been widened to debugging the system in a way efficient for the developpers.

Using formal modelling techniques in the development can not guarantee systems are error-free, for this is impeded by the necessary abstraction made to build the model. This experience shows that formal methods can be used at any stage of development to reinforce the quality of the product.

This has been the conclusion of Schlumberger-Industries as well; and the R&D center is under works to internalize this new way of engineering.

References

1. M. Alabau, D. Bégay, J.-P. Radoux. *Formal Methods and Real-Time: design and validation of a Real-Time embedded System.* Real-Time Systems conference, Paris, january 1996.
2. A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.
3. A. Arnold, D. Bégay, P. Crubillé. *Construction and analysis of transition systems with MEC.* World Scientific Pub., 1994.
4. A. Arnold, D. Bégay, J.-P. Radoux. *The embedded software of an electricity meter: An experience in using formal methods in an industrial project.* In a Special issue of Sci. Comp. Prog., to appear 1996.
5. D. Bégay, J. Dormoy, P. Félix. An experiment in developing real-time systems using Mec. In Teodor Rus and Charles Rattray, editors, *Theories and experiences for real-time system development*, volume 2 of *AMAST series in Computing*, chapter 14, pages 363–388. World Scientific Pub., 1994.
6. P. Crubillé. *Réalisation de l'outil Mec : spécification fonctionnelle et architecture.* PhD thesis, Université de Bordeaux I, novembre 1989.
7. A. Dicky. *Une approche algébrique et algorithmique de l'analyse des systèmes de transition.* PhD thesis, Université de Bordeaux I, février 1985.
8. J.-P. Radoux. *Utilisation de systèmes de transitions finis pour la conception et le développement d'un système embarqué.* PhD thesis, Université de Bordeaux I, mars 1995.

This article was processed using the LaTeX macro packages with LLNCS style,
and with automata graphical style by A. Griffault from LaBRI.