Modelling and Analysis of a Commercial Field Bus Protocol

Alexandre David
adavid@DoCS.UU.SE

Wang Yi
yi@DoCS.UU.SE

Uppsala University
Department of Computer Science
Box 325, 751 05 Uppsala Sweden
Fax: +46 18 55 02 25

Abstract

We report on an industrial application of UPPAAL, *in which a commercial field bus protocol (AF100) is modelled and analysed using the tool. During the case study, a number of imperfections in the protocol logic and its implementation are found and the error sources are debugged based on abstract models of the protocol; respective improvements have been suggested. In this paper, we shall summarize our experiences in dealing with the complexity of the protocol using various modelling and abstraction features provided in* UPPAAL. *As an example, we study the bus coupler of AF100, which serves as the data link layer of the protocol.*

1. Introduction

In recent years, a number of modelling and verification tools for real-time systems [HHWT95, BLL+95, BLL+96, DOTY95] have been developed based on the theory of timed automata [AD94]. They have been successfully applied in various case-studies (e.g. [BGK+96, JLS96, SMF97]). However, the tools have been mainly used in the academic community, namely by the tool developers. It has been a challenge to apply these tools to the development and debugging of industrial products.

In this paper we report on an industrial application of the UPPAAL tool to model and debug a commercial field bus communication protocol, AF100 (Advant Fieldbus 100) developed and implemented in ABB for safety-critical applications e.g. process control. The protocol has been running in various industrial environments over the world for the past ten years. During its seven years on the market a number of errors have been detected which result in time–outs and retransmissions. Due to the complexity it has been very time and resource consuming to troubleshoot these errors. It is a great potential to become more efficient by using the tool UPPAAL and this is part of the motivation for initializing a project to model and analyse the protocol. A clear interest is to improve the methods, decrease the maintenance time/costs and to increase quality of the product. However the goal of the project is not to verify the correctness of the protocol in any sense of *completeness*, which is basically impossible due to the size and complexity of the system, but to localize the error sources in both the protocol logic and the implementation at the source level.

To our knowledge, the case study is the largest reported so far, where the UPPAAL tool has been applied, which involves hundreds of pages of protocol specification and thousands of lines of source code. During the case study, a number of errors in the protocol logic and its implementation have been found and debugged based on abstract models of the protocol; respective improvements have been suggested. It turns out that many of the problems are due to incorrect usage of synchronization and timing mechanisms in the implementation of the protocol, in particular, semaphores and timeouts. In this paper we shall summarize our experience in dealing with the complexity of the protocol using various abstraction features provided in UPPAAL. As an example, we study the bus coupler of AF100, which serves as the data link layer of the protocol. However the high level functions of VFI are believed not to be affected by these low level transient errors due to retransmission and protection of data. We are investigating this issue.

UPPAAL is a tool box for modelling and verification of safety and bounded liveness properties of networks of timed automata developed jointly by Uppsala University and Aalborg University. It contains a number of tools including a graphical interface, automatic generator of diagnostic traces, and a model-checker based on on-the-fly state-space examination and constraint solving techniques [YPD94, BLL+95]. It provides several features for modelling system behaviours on different levels of abstraction.

In modelling AF100, it turned out that "urgent state" is a useful notion for modelling race condition and "committed location" is a convenient means for abstraction. Both are implemented in UPPAAL. As an example, we have studied in details the bus coupler of AF100, which corresponds to the data link layer of the protocol. Several abstract models have been developed, offering different levels of abstraction on the behaviour of the bus coupler. A number of properties related to the abstraction levels have been checked and the sources of found errors have been exhibited by generating the diagnostic traces. In addition, dead codes are found in the protocol implementation.

The paper is organized as follows: Section 2 gives an informal description of AF100. Section 3 presents the modelling and the abstraction. Section 4 the verification and debugging. Section 5 concludes the paper.

2. An Informal Description of AF100

2.1. Overview

AF100 is a field bus communication protocol (Advant Fieldbus). It is designed for 80 stations communicating over a bus. A station acting as a "master", may initiate a dialog with up to 79 other stations acting as "slaves" in this dialog. The master requests information from a slave which only responds to it, thus the names master and slave. In fact the dialog is established between applications on stations and each station may have several applications running (acting as masters or slaves).

The protocol has two main layers which are: *VFI* (*Virtual Field Interface*) and the *Bus Coupler* corresponding to the transport and data link layers respectively in the ISO protocol standard [Tan81]. Typically a client application will use the master part of VFI to send requests to another station where a server application will respond through the slave part of VFI. VFI communicates with the bus via the Bus Coupler which uses a low level bus queue.

We are interested in the service calls from the application, which is the API offered by AF100 (*Application Programmer Interface*); the VFI protocol between the VFI peers over the Bus Coupler and the bus; the interface between VFI and the Bus Coupler; and the protocol between the Bus Coupler peers that we call the Bus Coupler protocol.

Message passing through VFI is as follows: First, an application sends messages to VFI, VFI cuts messages into packets and sends them to the Bus Coupler via an interface; the Bus Coupler receives packets and sends them over the bus to the next Bus Coupler. The receiving Bus Coupler accepts packets and sends them to VFI via an interface; VFI assembles messages from packets and signals the application when they are ready to be fetched. Finally an application blocked by VFI is signaled and gets messages.

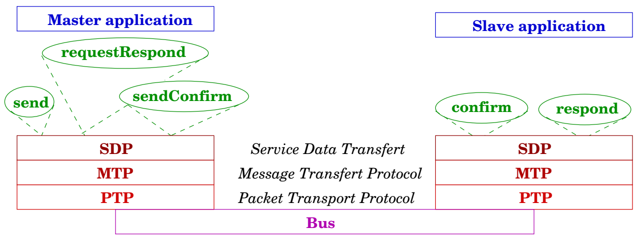

Figure 1. Layers of AF100.

On top of this message passing protocol we have the master/slave concept: a client application sends a VFI-master request message to a server which responds with a VFI-slave answer message. An acknowledgment mechanism ensures that messages are transmitted correctly.

Figure 1 gives the layered structure of the protocol of AF100. The API services are on top of the structure, VFI covers the *Service Data Transfer* and the *Message Transfer Protocol* layers and the Bus Coupler is the *Packet Transport Protocol*.

2.2. The Transport Layer: VFI

As mentioned earlier, VFI corresponds to the transport layer in the ISO standard. It offers three different services on the master side, which have a counterpart on the slave side: *request respond*, *send confirm* and *send message*. The last one is simple: it does not require an answer. The first two are similar and both require an answer provided by the services *respond* and *confirm* on the slave.

The VFI protocol is in short an alternating bit protocol at the message level and a sliding window protocol at the packet level. Packets are sent transparently, i.e. without acknowledgment within the window and they require acknowledgment between each window. The bit used to mark the packet is called the *transparent bit* and it is actually checked at every layer, which is the exception in the layered structure of the whole protocol.

We will now focus on the Bus Coupler that runs on a different board and operating system than VFI. This communication is the object of this paper.

2.3. The Data Link Layer: Bus Coupler

The Bus Coupler corresponds to the data link layer in the ISO standard. It is located on a separated board running its own operating system. The Bus Coupler communicates with VFI via an interface (a buffer accessible to both parts). The communication with the bus is achieved via *ports*. There are four dedicated ports per station. For each port, there is an associated Bus Coupler task, listening to the bus or the VFI. These dedicated ports are to be used in the following ways:

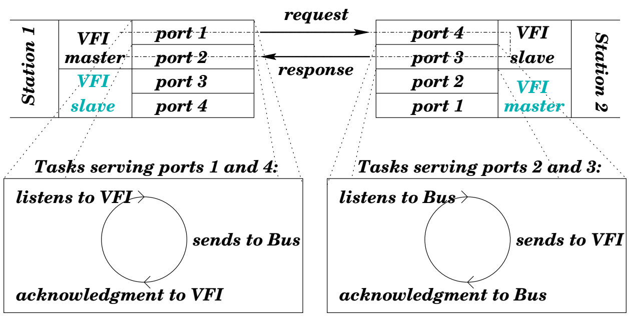

Figure 2. Bus Coupler communication scheme.

- a **request** from the master is sent to **port 1**
- a **response** is received by the master from **port 2**
- a **request** is received by the slave from **port 3**
- a **response** from the slave is sent to **port 4**.

Figure 2 illustrates this communication scheme on which we will focus. Tasks for the master and the slave are depicted. The request and the response are acknowledged.

This communication is symmetric for a request sent from the master or a response sent from the slave. From the Bus Coupler point of view it is just a point to point communication and this will give us an abstraction in the modelling.

After sending to the bus, the Bus Coupler will wait for an acknowledgment if the packet is not transparent, otherwise it returns and "lies" to VFI with a positive acknowledgment. The same holds for VFI receiving from the Bus Coupler.

Communication between VFI and the Bus Coupler is achieved via a buffer which is separated into fields writable by only one side, but readable by the other side. The synchronization bits are:

- **mail box reserved** to reserve access to the buffer
- **data read** to notify that data was read
- **data written** to notify that data was written
- **data lost** to return positive or negative acknowledgment.

Furthermore a data field to write the packet itself is reserved. Both sides have these fields.

The Bus Coupler protocol has two main components: the communication with VFI through this interface and the communication with the other Bus Coupler. The communication between bus couplers involves a minimum control of packets with management of re-sending packets and associated acknowledgments. The layer below is used via a simple API of the form send/receive. This part of the protocol is known to be robust so we will not treat it in this paper, which will be abstracted away in the model.

The implementation of the protocol uses signals to notify the reading side when a bit has been written. The interruption/signal mechanism is specific to this implementation and uses semaphores on both sides (different boards and operating systems) and the modelling stresses this feature.

3. Modelling and Abstraction

3.1. Some Experiences

We started to study the VFI layer. The documentation received from ABB included a functional description, a design description and a programmer's guide, that is 140 pages and 62 pages of Modula-2 source code covering the relevant parts of VFI, notably parts of the interface with the Bus Coupler.

The source code was used to construct the models and the documentation to understand the protocol and to make abstractions. The role of ABB was crucial in this phase. It turned out that to understand VFI, we had to consider the Bus Coupler because we needed information from the low level mechanisms. This ended up with the study of the Bus Coupler with 40 pages of dedicated source code. It resulted in a detailed model where the control structure of the source code can be tracked down though the automata. Then we simplified this model using abstraction techniques to localize errors.

3.2. The Modelling Process

We have adopted a top-down approach first to find and understand the relevant components of the systems and then a bottom-up approach with progressive abstractions which allows us to build up several abstract models for verification. As the goal of the project is debugging, finding the right properties to check has been an interesting experience itself. The following steps have been taken in the modelling process as illustrated in Figure 3:

1. model the Bus Coupler based on the source code
2. simplify the model for the Bus Coupler with abstractions
3. model VFI master and slave separately based on source code
4. simplify the model for VFI master and the slave part with abstractions
5. compose the complementary parts of the relevant components to derive abstract models for the whole system

Phase 1 is to construct a detailed model respecting the source code of the Bus Coupler which is presented in subsection 3.3. Phase 2 is to derive an abstract model of the implementation model of the Bus Coupler, presented in subsection 3.4 using the abstraction mechanism implemented in UPPAAL. Phase 3 was the first attempt in the project. As it turned out that we had to deal with the Bus Coupler first,

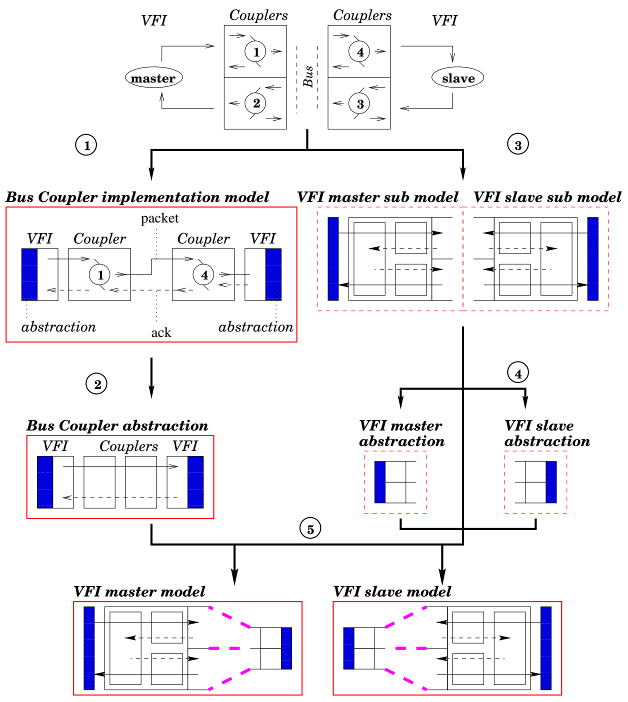

Figure 3. The modelling steps.

in phase 3, we aimed at deriving a test automaton based on the abstract behaviour of the VFI master and slaves. The simplified model for the Bus Coupler has been constructed and it contains the essential timing properties of the components, allowing us to carry out the last phases.

The work presented in this paper concerns mainly the first two phases. To avoid the state explosion problem, we have adopted the standard compositional verification techniques namely checking properties on the abstract models derived from low level models that are basically the direct transformation of the source code to the UPPAAL description language. In parallel, we have developed *abstract source codes* in a C-like language, corresponding to the timed automata written in the UPPAAL modelling language. Assumptions, abstractions, approximations and the event models are annotated in the abstract sources. It turns out that the abstract codes are very helpful for debugging and justifying our modelling decisions for our industrial partner.

3.3. Faithful Models of the Bus Coupler

A particular requirement from our industrial partner is that we should construct UPPAAL models that respect the protocol implementation so that all errors found and improvements suggested should be related to the source code. In developing the first model for the Bus Coupler, we had to keep the same structure as the source code and model all the implementation details except irrelevant data e.g. contents of data packets. This has been important for us to understand the protocol and to abstract away irrelevant details in the later phases of the project.

The first UPPAAL model derived directly from the source code consists of 16 automata, 4 clocks and 32 integer variables, modelling 4 sending and receiving processes, 4 semaphores and 8 functions. Note that this is only a part of the whole AF100 protocol. The structure of the model follows figure 2. Considering the number and the size of the automata, there is no point in giving them in this paper.

As usual, the contents of the packets are irrelevant to the correct behaviour of the protocol, except the transparent bit. The transparent bit is global in the sense that it is set by the VFI and read by all the layers of the protocol. It may have the special value -1 to mark corrupted data that should not be read. As there may be several applications using the bus coupler via VFI, packets $0, 1$, or -1 may arrive randomly at the Bus Coupler to be sent or delivered. We assume that packets may be generated randomly. The same idea is used in the receiver part where the upper layers may accept or refuse a packet; so positive or negative acknowledgment are randomly generated as well. Note that the value -1 is not part of the protocol but modelling bad data for the purpose of verification.

There are two bus couplers involved in the master side and the slave side connected by the physical bus for data transmission. There is a protocol for this layer. The bus is modelled as a lossy channel preserving the ordering of data packets. On each side of the bus couplers, there is a queue. The queue only introduces delay from the Bus Coupler when it wants to send a message. The same applies if the queue is full and the sender has to wait. The model concerning this part is a non-deterministic process sending or ignoring a packet within a time window. Timeout may occur as well. Concerning transmission, the delays are neglected with respect to the timeout periods controlling retransmissions.

We have derived 3 different models. The variations express different levels of assumptions on the program. The idea of the verification is to use an *error pruning* technique which is to detect an error and to go to that state where we deliberately cause a deadlock so that the state space is not explored further. We call this set of states the *error border*. When verifying properties, the interpretation of the results is as follows: if such a state is reached, then the property is *partially* verified for a system which does not contain the "error" states. However we know that they occur; so we make another model with less pruning, and in this way we have different refinement levels of the model with different levels of assumptions with corresponding partial properties. This is useful to track bugs. The variations of the model are as follows:

1. semaphore counter limited to 1, pruning error space

2. semaphore counter limited to 2, pruning error space
3. semaphore counter limited to 3, full space

The limitation on the counter is still kept because it was proved that the semaphores could be badly used and the counter could grow. It was limited to 3 because there is one class of semaphores which can have their counter reach 2 but not 3. The aim is to include the case where we have a greater counter of one class over the other one. Finally the 4^{th} model with a proposed correction was derived. The models are constructed so that the following inclusions between their state spaces hold:

$$space_1 \subseteq space_2 \subseteq space_3$$
$$space_{1\backslash EB} \subseteq space_{2\backslash EB} \subseteq space_4 \subseteq space_3$$

Where $space_{i\backslash EB}$ denotes $space_i$ excluding EB.

In the modelling process, the models are refined by various modelling mechanisms implemented in UPPAAL including:

- *committed* states: the state must be left immediately with no delay. Interleaving is allowed only between other committed states. Atomicity in a sequence of states may be achieved, thus reducing the state space.
- *urgent* states: time is not allowed to progress in such a state, but all interleavings are allowed. It is useful to model race condition and non-determinism.
- *urgent* transitions: they should be taken whenever the guards become true. It is useful to model progress.
- states decorated with *invariants* which are constraints on clocks. It can be used to model timeouts.

3.4. Abstract Models of the Bus Coupler

To debug the protocol logic, we had to simplify the detailed model (which is based on the source code) using abstraction techniques and the modelling mechanisms listed above, in particular, the notions of *committed* and *urgent* states. The derivation takes away specific parts related to implementation which are the signal implementation and the way to wait on the bits.

The evolution of the models is in two dimensions: breaking atomicity of transitions and allowing delay in reading the bits. This process yields five models:

Model 1 is the simplest model where some transitions are considered to be atomic to study their consequences.

Model 2 relaxes model 1, by removing the atomicity of the transitions performing data-reading.

Model 3 relaxes model 2 by allowing delays when a bit is set to the expected value.

Model 4 also relaxes model 2 but by converting committed states related to data reading and writing to urgent states.

Model 5 relaxes model 4 by allowing delays as in model 3.

The case with no delay is modelled by an urgent synchronization which is always enabled, but in order to take the transition, a guard on a condition (the bit the component is waiting for) must be satisfied. When enabling delay, this synchronization is removed, allowing time to progress even if the guard is true. By the semantics of *committed* and *urgent* states, the state spaces of the derived models are related as follows :

$$\begin{array}{ccccc} space_1 & \subseteq & space_2 & \subseteq & space_4 \\ & & \cap\!\!\!\shortmid & & \cap\!\!\!\shortmid \\ & & space_3 & \subseteq & space_5 \end{array}$$

These variants are used to analyse different aspects of the behaviour. The idea to derive models 3 and 5 is to stress delay and models 4 and 5 is to stress race condition. Note also that the reason for having these different models is to understand the influence of slight variations of the interpretation of the protocol and how they are related to the properties we want to check. The verification results are consistent with the inclusions.

4. Verification and Debugging

In this section we present the correctness properties checked. They are either reachability properties of the form $\exists\Diamond\ \phi$ or invariants of the form $\forall\Box\ \phi$. The ϕ predicate is on states, variables and time. In addition to these explicit properties we also found deadlocks reported by UPPAAL.

4.1. Properties

Finding the properties to check was a problem in itself because the documentation was not adequate for verification. We succeeded in formalizing 4 classes of properties for the full model and the reduced models :

- 6 correctness properties (resp. 6 for the reduced model), related to the logics of the protocol. Violating these properties result in inconsistent data read. 4 of these properties are equivalent in both abstraction levels.
- 25 functional properties (resp. 5), related to the synchronization of the components. Violating these properties could induce bad/wrong behaviour. The properties of the implementation models are classified as follows: 8 related to the implemented semaphores, 10 to detection of possibly bad states belonging to the *error border* and 7 related to precedence between states. The abstract models properties were based only on precedence.
- 19 behaviour properties (resp. 5), which are intuitively believed to hold with respect to the protocol. This is

Model	Size	Construction	Verification
1	213 MB	4:43 min	10:02 min
2	320 MB	8:34 min	19:10 min
3	892 MB	37:19 min	55:28 min
4	600 MB	21:51 min	44:13 min

Table 1. Resources used for verification.

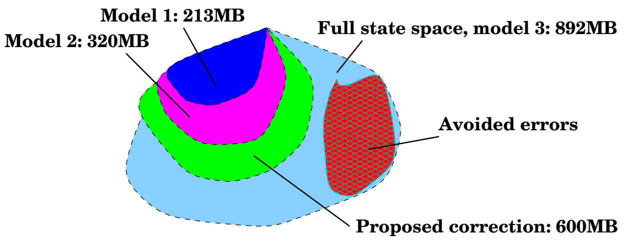

Figure 4. View of the state space.

expected behaviour which has only performance impact.

- 32 validation properties (resp. 19), related to the model itself to validate it. The protocol works in practice and the model must work the same. A more complex model requires more validation hence the difference.

We do not intend to present the 82 properties (resp. 35) but rather the important ones in the following sub-sections. Verification was conducted on Sun Ultra-Sparc-II Ultra-Enterprise server 248MHz equipped with 1.2GB of memory running SunOS 5.6. The version of UPPAAL was 2.28.8 and the verification options were re-use state space, breadth-first search and no trace generation to save memory.

4.2. Checking the Implementation Models

The resources needed to verify the properties are given in table 1 and they are consistent with the inclusions $space_{1\setminus EB} \subseteq space_{2\setminus EB} \subseteq space_4 \subseteq space_3$ that were given in sub-section 3.3. The construction time corresponds to verifying the first property that requires to construct the whole state space. The verification time is the cumulative time of the verification of 82 properties. The interest of re-using state space is clear. Figure 4 illustrates the space inclusions. A typical found trace is 100 steps long.

The correctness properties are:

```
1:A[]VFIToCoupler_1P1.written imply
vfiTrans1!=-1
2:A[](CouplerFromVFI_1P1.done and
resultC11==0) imply bcTrans11!=-1
3:A[]CouplerToBus_1P1.sent1 imply
bcTrans11!=-1
4:A[]CouplerFromBus_2P4.received imply
bcTrans24!=-1
5:A[]CouplerToVFI_2P4.step2w0 imply
bcTrans24!=-1
6:A[]VFIFromCoupler_2P4.dataTaken imply
vfiTrans2!=-1
```

where `A[]` stands for $\forall\Box$. They concern the transparent bit (data modelled) which should not be written/read when not valid (-1) by VFI (vfiTrans) and the Bus Coupler (bcTrans). The full state model 3 does not satisfy properties 2,3,4 and 6. The models 1 and 2 partially verify these properties and the 4^{th} one appears to be able to avoid the error. The other properties are satisfied. By enabling the error trace one sees that the problem may come from a de-synchronization.

4 Properties concerning semaphores are:

```
44:A[]not SemVFItoCoupler24.signalNotTaken
45:A[]not SemCouplertoVFI24.signalNotTaken
46:A[]not SemVFItoCoupler11.signalNotTaken
47:A[]not SemCouplertoVFI11.signalNotTaken
```

They mean that whenever a signal is sent, the previous one should have been accepted otherwise "it has not been taken" and if there is a wait on that signal, it will not make much sense since the semaphore stores previous signals. Other properties are checked on the explicit value of the counter as well. None of these properties are verified for model 1, only property 45 fails for the other models. With respect to the models, we proved that the counter may reach 2, but not 3 except for one semaphore where the bound is not known: a separate test with a temporary modified automaton was performed with `E<> SemCoupler-toVFI24.signalNotTaken` with a limit on the stored signals of 10 and this was satisfied. `E<>` stands for $\exists\Diamond$. There seems to be a live-loop.

2 Precedence properties are:

```
75:A[]not (CouplerToVFI_2P4.endWait2 and
(VFIFromCoupler_2P4.waited or VFI
FromCoupler_2P4.wait0))
76:A[]not (VFIToCoupler_1P1.testOK and
(CouplerFromVFI_1P1.step1w0 or
CouplerFromVFI_1P1.step2))
```

Only the full error model does not satisfy these ones. However it is not true for all this kind of properties. They mean that a side should not be sending an acknowledgment while the other side is going to begin to send a packet, or one side is at the end of sending a packet with success while the other side still waits for acknowledgment. These precedences are between VFI and the couplers, where the communication is not lossy and closely synchronized.

Examples of behaviour properties are:

```
10:A[]not (VFIToCoupler_1P1.done and
resultV1!=0 and bcTrans11==1)
16:A[]not (Coupler_1P1.sentTO and
bcTrans11==1)
32:A[]not (Coupler_2P4.acking and
saveTrans24==1)
```

Model	Size	Construction	Verification
1	3.8 MB	5 sec	8 sec
2	4.1 MB	5 sec	9 sec
4	5.0 MB	7 sec	10 sec
3	11 MB	28 sec	32 sec
5	14 MB	37 sec	41 sec

Table 2. Resources used for the abstract models.

```
78:A[](VFIToCoupler_1P1.testOK and
vfiTrans1==1) imply devdatalost11==0
```

Property 10 states that sending a transparent packet should never fail and this is false. Property 16 states that timeout should not occur on transparent packet which is true. Property 32 states that acknowledgement is not sent after transparent packets which is true. Property 78 states that the coupler "lies" properly to VFI when a transparent packet is sent, which is true.

As we will see in the next sub-section, the origin of the de-synchronization may come from race conditions. We use "may" because the model-checker finds just one counter-example which happens to be like that. Other cases may be possible. This explains that the protocol still works in practice. However problems are possible and they do occur. The partial correctness of properties allow us to pinpoint the source of a possible problem, which is in essence de-synchronization.

4.3. Debugging the Abstract Models

The resources needed to verify the properties are given in table 2 and they are consistent with the inclusions $space_1 \subseteq space_2 \subseteq space_4 \subseteq space_5$ and $space_2 \subseteq space_3 \subseteq space_5$ that were given in sub-section 3.3. 35 properties were verified in this case. Due to the way we constructed the models, we believe that $space_2 = space_3 \cap space_4$ though we can not prove it. However even the weaker relation $space_2 \subseteq space_3 \cap space_4$ is interesting especially when some properties are verified in $space_1$ but not in $space_2$ and therefore neither in $space_3$ nor $space_4$. This allows us to pinpoint behaviour differences.

The correctness properties are:

```
1:A[]master.waitDataR imply vfitrans1!=-1
2:A[]coupler1P1.sending imply bctrans11!=-1
3:A[]coupler2P4.gotMsg imply store24!=-1
4:A[]slave.read imply vfitrans2!=-1
5:A[]master.OK imply devdatalost11!=-1
6:A[]coupler2P4.readnottrans imply
cpudatalost24!=-1
```

These are of the same type of the implementation properties. They state that wrong data should not be read. Wrong data are too early or too late. We added here the explicit test on the acknowledgment answer from the coupler or the VFI-slave with dev/cpudatalost. This is verified in the implementation as well, but indirectly.

Properties 1 and 5 are satisfied by all the models. Properties 2 and 3 are satisfied only when no delay is allowed, which is the case for the models 1, 2 and 4. When delay is allowed a timeout may occur concurrently leading to an unwanted change which leads to a race condition. To interpret this as realistic or not, the hardware and runtime environment has to be taken into consideration. In the context of multitasking that we have with non-preemption on the Bus Coupler side, this situation could be possible if the coupler blocks while sending.

Property 4 is satisfied only for the first model. This property is sensitive to race condition. Property 6 is satisfied only for the 3 first models. Models 4 and 5 introduce new interleavings and a race condition is enabled by changing *commit* states to *urgent* states.

The functional properties are:

```
31:A[]not (coupler2P4.readnottrans and
slave.read)
32:A[]not (coupler2P4.readtrans and
slave.read)
33:A[]not (master.OK and
coupler1P1.sending)
34:A[]not (coupler2P4.sending and
cpumbr24==1 and slave.read)
35:A[]not (master.waitMBR and devmbr11==1
and coupler1P1.sending and ck11==0)
```

They concern de-synchronization, when a component is one cycle late on the other. Properties 31 and 32 state that the coupler should not be in a state ready to read the acknowledgment from the slave while this one has not written it and is about to do it: models 4 and 5 violate these properties. This result is similar to property 6.

Property 33 states that the master should not have read the acknowledgment from the coupler whereas this one has not written it yet. Model 5 does not satisfy this one, which means that this property is related to delay *and* race condition.

Property 34 states that the coupler should not be in a state waiting for the mailbox being available in order to write data while the slave has read data and not reserved yet the mailbox. This is satisfied by all the models.

Property 35 states that the coupler should not be in a state when it has just reserved the mailbox and read data from the master though this one is waiting for the mailbox to be freed in order to write data. Models 3 and 5 do not satisfy this one. This property is sensitive to delays.

The conclusion on the abstract model is that the protocol is implementable since the first model is valid. However the implementation has to avoid some possible race conditions as well as some delays in order to work. This is a feasibility proof with warnings on the robustness of the protocol.

5. Conclusion

In this paper, we have presented an industrial case study where the UPPAAL tool is applied to model and verify a commercial real-time communication protocol. The main output of the case-study is a sequence of abstract models of the protocol logic and its implementation at different abstraction levels, and a number of properties verified on the models to check functionality of the protocol and to localize errors in the implementation. During the case study, a number of imperfections in the protocol logic and its implementation have been found and debugged based on abstract models of the protocol; respective improvements have been suggested. This may be considered as one piece of evidence that the validation and verification tools of today are mature enough to be applied in debugging industrial systems. Finally we are pleased to mention that the work has been very much appreciated by ABB and the company intends to extend further the use of formal methods.

6. Acknowledgment

We would like to thank Ulf Hammar, Tomas Lindström and Patrik Gunnarsson from ABB Automation Products who devoted much of their time to explain the protocol. We are very grateful to Pedro D'Argentinio who participated in the initial phase of the project, as well as the whole UPPAAL team for their support.

References

[AD94] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[BGK+96] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. CAV'96, LNCS 1102 in, pages 244–256. Springer–Verlag, July 1996.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[BLL+96] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the* 2nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, in LNCS 1055, pages 431–434, March 1996.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *Proc. of Workshop on Verification and Control of Hybrid Systems III*, LNCS 1066, pages 208–219, October 1995.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: The Next Generation. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 56–65. IEEE Computer Society Press, December 1995.

[JLS96] Henrik E. Jensen, Kim G. Larsen, and Arne Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *Proc. of* 2nd *Int. Workshop on the SPIN Verification System*, pages 1–20, August 1996.

[SMF97] Thomas Stauner, Olaf Mller, and Max Fuchs. Using HyTech to Verify an Automotive Control System. In *Proc. Hybrid and Real-Time Systems, Grenoble, March 26-28, 1997*. Technische Universität München, Lecture Notes in Computer Science, Springer, 1997.

[Tan81] A.S. Tanenbaum. *Computer networks*. Prentice–Hall, 1981.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.