Formal Verification of a Power Controller Using the Real-Time Model Checker UPPAAL

Klaus Havelund[1], Kim Guldstrand Larsen[2], and Arne Skou[2]

[1] NASA Ames Research Center, Recom Technologies, CA, USA
havelund@ptolemy.arc.nasa.gov
[2] BRICS, Aalborg University, Denmark
{kgl,ask}@cs.auc.dk

Abstract. A real-time system for power-down control in audio/video components is modeled and verified using the real-time model checker UPPAAL. The system is supposed to reside in an audio/video component and control (read from and write to) links to neighbor audio/video components such as TV, VCR and remote–control. In particular, the system is responsible for the powering up and down of the component in between the arrival of data, and in order to do so in a safe way without loss of data, it is essential that no link interrupts are lost. Hence, a component system is a multitasking system with hard real-time requirements, and we present techniques for modeling time consumption in such a multitasked, prioritized system. The work has been carried out in a collaboration between Aalborg University and the audio/video company B&O. By modeling the system, 3 design errors were identified and corrected, and the following verification confirmed the validity of the design but also revealed the necessity for an upper limit of the interrupt frequency. The resulting design has been implemented and it is going to be incorporated as part of a new product line.

1 Introduction

Since the basic results by Alur, Courcoubetis and Dill [3, 4] on decidability of model checking for real–time systems with dense time, a number of tools for automatic verification of hybrid and real–time systems have emerged [7, 14, 10]. These tools have by now reached a state, where they are mature enough for application on industrial development of real-time systems as we hope to demonstrate in this paper.

One such tool is the real–time verification tool UPPAAL[1] [7] developed jointly by BRICS[2] at Aalborg University and Department of Computing Systems at Uppsala University. The tool provides support for automatic verification of safety and bounded liveness properties of real–time systems and contains a number of additional features including graphical interfaces for designing and simulating system models. The tool has been applied successfully to a number of case–studies [13, 18, 5, 6, 16, 9] which can roughly be divided in two classes: real–time controllers and real–time communication protocols.

[1] See URL: http://www.docs.uu.se/docs/rtmv/uppaal for information about UPPAAL.

[2] BRICS – Basic Research in Computer Science – is a basic research centre funded by the Danish government at Aarhus and Aalborg University.

Industrial developers of embedded systems have been following the above work with great interest, because the real–time aspects of concurrent systems can be extremely difficult to analyze during the design and implementation phase. One such company is Bang & Olufsen (B&O) – having development and production of fully integrated home audio/video systems as a main activity.

The work presented in this paper documents a collaboration between AAU (Aalborg University) – under the BRICS project – and B&O on the development of one of the company's new designs: a system for audio/video power control. The system is supposed to reside in an audio/video component and control (read from and write to) links to neighbor audio/video components such as TV, VCR and remote–control. In particular, the system is responsible for the powering up and down of the component in between the arrival of data, and in order to do so, it is essential that no link interrupts are lost. The work is a continuation of an earlier successful collaboration [13] between the same two organizations, where an existing audio/video protocol for detecting collisions on a link between audio/video components was analyzed and found to contain a timing error causing occasional data loss. The interesting point was, that the error was a decade old, like the protocol, and that it was known to exist – but normal testing had never been sufficient in tracking down the reason for the error.

The collaboration between B&O and AAU spanned 3 weeks (4 including report writing), and was very intense the first week, where a representative from B&O visited AAU, and a first sketch of the model was produced. During the next two weeks, the model was refined, and 15 properties formulated by B&O in natural language were formalized and then verified using the UPPAAL model checker. During a meeting, revisions to the model and properties were suggested, and a final effort was spent on model revision, re-verification and report writing. The present paper is an intensive elaboration of the preliminary report [12][3].

The paper is structured as follows. Section 2 contains an informal description of the B&O protocol, and in section 3 we present the UPPAAL modeling language and tool. In section 4 we present our techniques for modeling timed transitions and interrupts in the UPPAAL language. Section 5 presents the formal modeling of this protocol in the UPPAAL language, while section 6 presents the verification results. Finally section 7 provides an evaluation of the project and points out future work.

2 Informal Description of the Power Down Protocol

In this section, we provide an informal description of the designed protocol for power down control in an audio/video component. As advocated in [15], we divide the description into environment, syntax, and protocol rules.

2.1 Protocol Environment

A typical B&O configuration (see figure 1) consists of a number of components, which are interconnected by different kinds of links carrying audio/video data and (or) control

[3] A full version of the paper is available at http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund.

information. Each component is equipped with two processors controlling audio/video devices and links, and among other tasks, the processors must minimize the energy consumption when the component goes stand by. Each processor may be in one of two modes: (1) active, where it is operational and can handle its devices and links, (2) stand by, where it is unable to do anything except wake up and enter active mode. One of the processors acts as a master in the sense that it may order the other processor (the slave) to enter stand by mode (and thereby reduce energy consumption). Due to physical laws[4] a processor cannot leave stand by mode via one atomic action, and the purpose of the protocol is to ensure that stand by operation is handled in a consistent way, i.e. when one of the processors enters or leaves stand by mode, this is also recognized by the other processor. Furthermore, whenever a processor senses valid data on an external link, it must leave stand by operation. Also, the real-time duration for switching between the modes may not exceed a given upper limit in order not to lose messages.

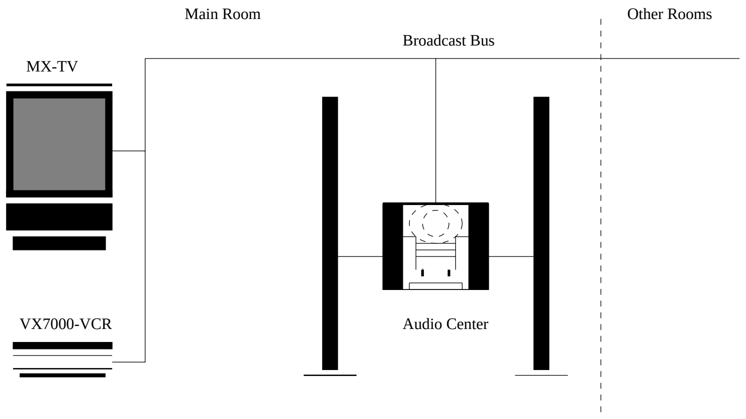

Fig. 1. Example B&O configuration.

Figure 2 illustrates the processor interconnection and our model of the software architecture for one of the processors. Each processor communicates with devices and other components via external links[5], and the two processors are interconnected via an internal link. The software architecture will be almost identical for the two processors, and in this report we concentrate on the IOP3212 processor – the slave processor. The main software module is the IOP process which communicates with the AP processor, the external link drivers, and the interrupt handlers according to the protocol rules described below. The protocol forms the crucial part of the software design, because it must assure that no data and interrupts are lost (in order to leave stand by operation at due time).

[4] It takes e.g. approx. 1 ms to make the processor operational when it has been in stand by operation.

[5] The figure illustrates a configuration with one external link, the LSL link.

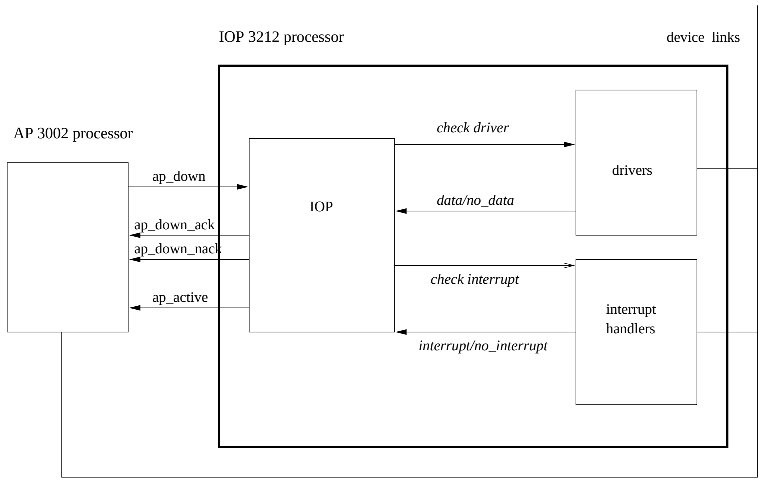

Fig. 2. Software architecture of the power down protocol. The protocol entity process (IOP) receives protocol commands (left arrows) from the drivers and interrupt handlers by issuing check commands (right arrows).

2.2 Protocol Syntax

The power down protocol entity (the IOP process) communicates with its environment (AP processor, link drivers and interrupt handlers) via the protocol commands in the set: {ap_down, ap_active, ap_down_ack, ap_down_nack, data, no_data, interrupt, no_interrupt}. The *ap_down* command is sent from the AP processor and commands the IOP processor to enter stand by operation. The *data* command is sent from a link driver and indicates that meaningful input has been detected on the link, whereas the *no_data* command indicates that there is no input from the link. Likewise, the *interrupt (no_interrupt)* command is sent from from the link interrupt handler and indicates that an interrupt (or no interrupt) has been received at the link interrupt interface. The commands *ap_active, ap_down_ack, ap_down_nack* informs the AP3002 processor about state changes of the protocol, that is, *ap_active* is sent when the IOP3212 processor becomes active, *ap_down_ack* is sent when it accepts to enter stand by mode, and *ap_down_nack* is sent when stand by cannot be entered.

2.3 Protocol Rules

In order to give an intuitive explanation of the protocol, we describe below in an informal way the major protocol rules, which must be obeyed by the IOP protocol entity. We leave out the details on communication with interrupt handlers and drivers, which will be described in the formalization section. In order to structure the description, we define the following major phases (see Figure 3 below) for the entity: the *active phase*, where the IOP is in normal (active) operation, the *check driver phase*, where the IOP process is waiting for a driver status (no data/data) in order to decide whether or not to leave the active phase, the *stand_by phase*, where the IOP processor is out of operation, and the *check interrupts phase*, where the IOP processor is waiting for an interrupt handler status (no interrupt/interrupt) in order to decide whether or not to enter the stand by phase. We use ?/! to indicate protocol input/output in the usual way.

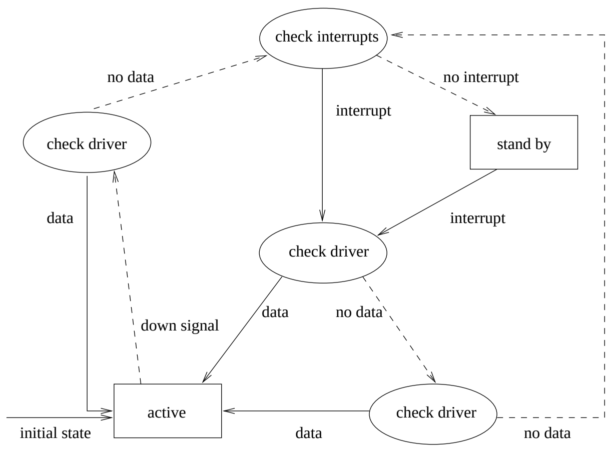

Fig. 3. Major protocol phases. The dotted lines indicate transitions leading towards power down. The full lines are leading towards power up. The two neighboring 'check driver' phases are necessary in order to be able to ignore noise from the communication lines.

Active rule In the active phase, the IOP protocol entity must enter the check driver phase, whenever a *ap_down* command is received from the AP processor.

Check driver rule In the check driver phase, the IOP protocol entity commands the drivers to check whether or not meaningful data are received from the links. The outcome of the check defines the succeeding phase according to Figure 3.

Stand_by rule Whenever an interrupt is received in the stand by phase, the IOP protocol entity must enter the check driver phase.

Check interrupts rule In the check interrupts phase, the protocol entity commands the interrupt handlers to check for pending interrupts. If no interrupts are pending, the stand by phase can safely be entered. Otherwise, the check driver phase is entered.

The above rules have to be implemented in such a way, that (1) Whenever an interrupt is received and meaningful data is present on the given link, the active phase must be entered, and (2) Whenever a down signal is received from the AP processor and no interrupts and valid data are present, the stand by phase must be entered. The delay caused by software of these transitions may not exceed $1500\mu s$ since otherwise data may be lost.

The informal rules form the basis for the model design, and in the analysis section, we present a complete list of protocol requirements in terms of properties of the formal protocol model.

3 The UPPAAL Model and Tool

UPPAAL is a tool box for symbolic simulation and automatic verification of real–timed systems modeled as networks of timed automata [4] extended with global shared integer variables. More precisely, a model consists of a collection of non–deterministic processes with finite control structure and real–valued clocks communicating through channels and shared integer variables. The tool box is developed in collaboration between BRICS at Aalborg University and Department of Computing Systems at Uppsala University, and has been applied to several case–studies [13, 18, 5, 6, 16, 9].

The current version of UPPAAL is implemented in C++, XFORMS and MOTIF and includes the following main features:

– A graphical interface based on Autograph [8] allowing graphical descriptions of systems.
– A compiler transforming graphical descriptions into a textual programming format.
– A simulator, which provides a graphical visualization and recording of the possible dynamic behaviors of a system description. This allows for inexpensive fault detection in the early modeling stages.
– A model checker for automatic verification of safety and bounded–liveness properties by on–the–fly reachability analysis.
– Generation of (shortest) diagnostic traces in case verification of a particular real–time system fails. The diagnostic traces may be graphically visualized using the simulator.

A system description (or model) in UPPAAL consists of a collection of automata modeling the finite control structures of the system. In addition the model uses a finite set of (global) real–valued clocks and integer variables.

Consider the model of Figure 4. The model consists of two components A and B with control nodes {`A0`, `A1`, `A2`, `A3`} and {`B0`, `B1`, `B2`, `B3`} respectively. In addition to these discrete control structures, the model uses two clocks `x` and `y`, one integer variable `n` and a channel `a` for communication.

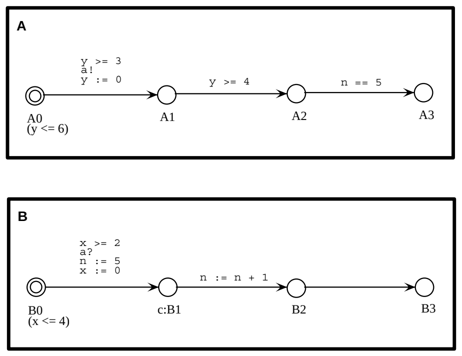

Fig. 4. An example UPPAAL model

The edges of the automata are decorated with three types of labels: a *guard*, expressing a condition on the values of clocks and integer variables that must be satisfied in order for the edge to be taken; a synchronization *action* which is performed when the edge is taken forcing as in CCS [19] synchronization with another component on a complementary action[6], and finally a number of *clock resets* and *assignments* to integer variables. All three types of labels are optional: absence of a guard is interpreted as the condition $true$, and absence of a synchronization action indicates an internal (non–synchronizing) edge similar to τ–transitions in CCS. Reconsider Figure 4. Here the edge between `A0` and `A1` can only be taken, when the value of the clock `y` is greater than or equal to 3. When the edge is taken the action `a!` is performed thus insisting on synchronization with B on the complementary action `a?`; that is for A to take the edge in question, B must simultaneously be able to take the edge from `B0` to `B1`. Finally, when taking the edge, the clock `y` is reset to 0.

In addition, control nodes may be decorated with so–called *invariants*, which express constraints on the clock values in order for control to remain in a particular node. Thus, in Figure 4, control can only remain in `A0` as long as the value of `y` is no more than 6.

Formally, states of a UPPAAL model are of the form $(\bar{l}, v)$, where $\bar{l}$ is a *control vector* indicating the current control node for each component of the network and v is an *assignment* given the current value for each clock and integer variable. The *initial state* of a UPPAAL model consists of the initial node of all components[7] and an assignment giving the value 0 for all clocks and integer variables. A UPPAAL model determines the following two types of *transitions* between states:

Delay transitions As long as none of the invariants of the control nodes in the current state are violated, time may progress without affecting the control node vector and with all clock values incremented with the elapsed duration of time. In Figure 4, from the initial state $\langle(\mathtt{A0},\mathtt{B0}), \mathtt{x}=0, \mathtt{y}=0, \mathtt{n}=0\rangle$ time may elapse 3.5 time units leading to the state $\langle(\mathtt{A0},\mathtt{B0}), \mathtt{x}=3.5, \mathtt{y}=3.5, \mathtt{n}=0\rangle$. However, time cannot elapse 5 time units as this would violate the invariant of `B0`.

Action transitions If two complementary labeled edges of two different components are enabled in a state then they can synchronize. Thus in state $\langle(\mathtt{A0},\mathtt{B0}), \mathtt{x}=3.5, \mathtt{y}=3.5, \mathtt{n}=0\rangle$ the two components can synchronize on `a` leading to the new state $\langle(\mathtt{A1},\mathtt{B1}), \mathtt{x}=0, \mathtt{y}=0, \mathtt{n}=5\rangle$ (note that `x`, `y`, and `n` have been appropriately updated). If a component has an internal edge enabled, the edge can be taken without any synchronization. Thus in state $\langle(\mathtt{A1},\mathtt{B1}), \mathtt{x}=0, \mathtt{y}=0, \mathtt{n}=5\rangle$, the B–component can perform without synchronizing with A, leading to the state $\langle(\mathtt{A1},\mathtt{B2}), \mathtt{x}=0, \mathtt{y}=0, \mathtt{n}=6\rangle$.

Finally, in order to enable modeling of atomicity of transition–sequences of a particular component (i.e. without time–delay and interleaving of other components) nodes may be marked as *committed* (indicated by a `c`–prefix). If in a state one of the components is in a control node labeled as being committed, no delay is allowed to occur and

[6] Given a channel name `a`, `a!` and `a?` denote complementary actions corresponding to *sending* respectively *receiving* on the channel `a`.

[7] indicated graphically by a double circled node.

any action transition (synchronizing or not) *must* involve the particular component (the component is so–to–speak committed to continue). In the state $((\mathtt{A1}, \mathtt{B1}), \mathtt{x} = 0, \mathtt{y} = 0, \mathtt{n} = 5)$ B1 is committed; thus without any delay the next transition must involve the B–component. Hence the two first transitions of B are guaranteed to be performed atomically. Besides ensuring atomicity, the notion of *committed* nodes also helps in significantly reducing the space–consumption during verification. Channels can in addition be defined as *urgent*: when two components can synchronize on an urgent channel no further delay is allowed before communication takes place.

In this section and indeed in the modeling of the audio/video protocol presented in the following sections, the values of all clocks are assumed to increase with identical speed (perfect clocks). However, UPPAAL also supports analysis of timed automata with varying and drifting time–speed of clocks. This feature was crucial in the modeling and analysis of the Philips Audio–Control protocol [5] using UPPAAL.

UPPAAL is able to check for reachability properties, in particular whether a certain combination of control-nodes and constraints on clock and data variables is reachable from an initial configuration. The properties that can be analyzed are of two forms: "A[] p" and "E<>p", where p is a formula over clock variables, data variables, and control-node positions. Intuitively for "A[] p" to be satisfied, all reachable states must satisfy p. Dually, for "E<>p" to be satisfied, some reachable state must satisfy p.

4 Timed Transitions and Interrupts

In this section, we shall introduce techniques for dealing with a couple of concepts that appear in the protocol, and which are not supported directly by the UPPAAL notation. These concepts are on the one hand *time slicing* in combination with *time consuming transitions*, and on the other hand prioritized *interrupts*. We refer to time slicing as the activity of delegating and scheduling execution rights to processes that all run on the same single processor. Transitions normally don't take time in UPPAAL, but this occurs in the protocol. Interrupts is a well known concept.

First, we give a small example illustrating what we need. Then we suggest the techniques that we shall apply in the modeling of the protocol.

4.1 The Problem

Assume a system with two processes A and B running on a single processor. Assume further, that these processes can be interrupted by an interrupt handler. The situation is illustrated in Figure 5, which is *not* expressed in the UPPAAL language, but rather in some informal extension of the language.

Each edge modifies a variable (A modifies x and y, B modifies v and w, and the interrupt handler modifies i and j). These assignments only serve to identify the edges and have no real importance for the example. Each edge is furthermore labeled with a time slot within parenthesis (2, 5, 7-12), indicating the amount of time units the edge takes. The slot 7-12 means anywhere between 7 and 12 time units.

Suppose the interrupt handler does not interrupt. Then the semantics should be the following: A and B execute in an interleaved manner modeling the time slicing of the processor – each transition taking the amount of time it is labeled with. No unnecessary

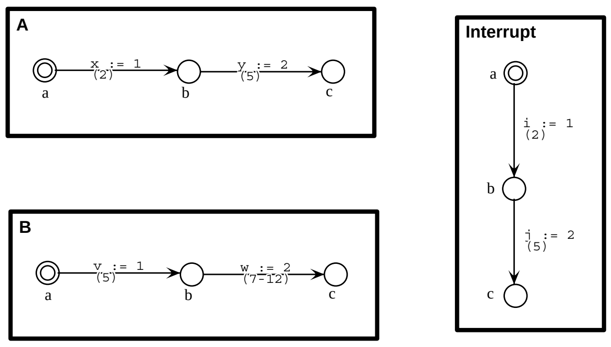

Fig. 5. What we want to express

time is spent in intermediate nodes (except waiting for the other process to execute). At the end, as soon as both A and B are in the node `c`, at least 19 $(2 + 5 + 5 + 7)$ and at most 24 $(2 + 5 + 5 + 12)$ time units will have passed.

An interrupt can occur at any moment and executes "to the end" when occurring. That is, it goes from node `a` to `c` without neither A nor `B` being allowed to execute in the meantime. If we assume that the interrupt handler can also interrupt, then it will change the above numbers to 26 $(19 + 2 + 5)$ and 31 $(24 + 2 + 5)$.

Or goal is now to formulate this in the UPPAAL language. Consider an approach where nodes are annotated with time constraints on local clocks, expressing the time consumed by the *previous* edge. This solution does not work since the two automata may consume time "together", and does not reflect the desired behavior, since they are supposed to run on a single processor. Let us first model time consuming transitions, ignoring the interrupts for a moment.

4.2 Modeling Timed Transitions

In a single processor setting it is natural to hand over time control to a single "operating system" process. Figure 6 illustrates such a process, called Timer, using a local clock `k`.

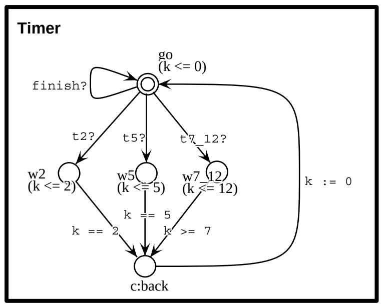

Fig. 6. The Timer

It has a start node, named `go`, in which time is constrained to not progress at all. This means that in order for time to progress, one of the edges `t2?`, `t5?` or `t7_12?` must be taken. These edges then lead to nodes where time can progress the corresponding number of time units, where after control returns immediately (back is a committed node just used to collect the edges) to the `go` node.

Now let us turn to the processes A and B, which are shown in Figure 7. These now communicate with the Timer, asking for time slots. Every time unit `T` that in the informal model, Figure 5, was in brackets `(T)` is now expressed as `tT!`. When for example A takes the edge from node `a` to node `b`, the Timer goes into the node `w2`, and stays there for 2 time units while A stays in node `b`. Hence, the time consumed by an edge is really consumed in the node it leads to. We have, however, guaranteed that B for example, cannot go to the node `b` and consume time "in parallel" since that would require a communication with Timer, and this is not ready for that before it returns to the node go.

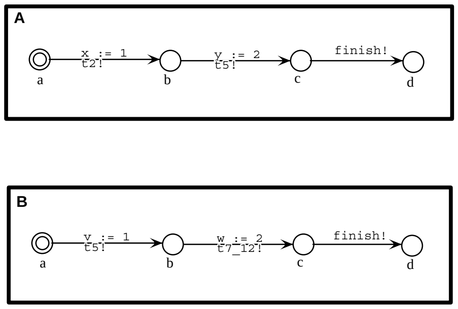

Fig. 7. A and B communicating with the Timer

When A reaches the node `c`, it has not yet consumed 7 time units $(2+5)$, it has only consumed 2. The 5 will be consumed while in node `c`. In order to reach a state where we for sure know that all the time has been consumed, we add an extra `d` node, which is reached by communicating `finish!` to the Timer. This forces the Timer to "finish" the last time consumption. Now we can express and verify the following true property, where `gc` is a global clock variable that is never reset:

```
A[] (A.d and B.d) imply ((19 <= gc) and (gc <= 24))
```

That is, if both A and B reach node `d`, then they will do so within $19-24$ time units. Note that due to the design of the Timer, time cannot progress further when that happens (the Timer will be in the `go` node where time cannot progress). Of course one can design a Timer that allows time to progress freely when asked to, and that is in fact what happens in the protocol. Basically one introduces an idle node in the Timer, that can be entered upon request, and where time can progress without constraints.

It is possible to model such single processor time scheduling in model checkers lacking real-time features, such as for example SPIN [15]. However, when trying to formulate and verify properties where time ticks are summed up, such explicit modeling easily leads to state space explosion.

4.3 Modeling Interrupts

Now we incorporate the interrupt handler. The basic idea is to give a priority to each process, and then maintain a variable, which at any moment contains the priority currently active. Processes with a priority lower than the current cannot execute. When an interrupt occurs, the current priority is set to a value higher than those of the processes interrupted.

Processes A and B can for example have priority 0 while the interrupt handler gets priority 1. When the interrupt occurs, the current priority is then set to 1, preventing priority 0 processes from running. We introduce the variable `cur` for this purpose, see Figure 8. The Timer stays unchanged.

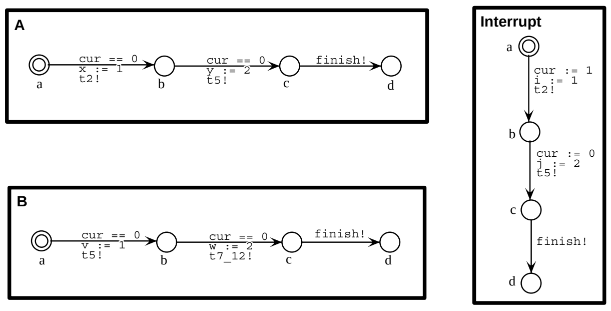

Fig. 8. Dealing with interrupts

Note how the variable `cur` occurs in guards of A and B, and how it is assigned to by the interrupt handler. In this model, we can verify the following property to be true:

```
A[] (A.d and B.d and Interrupt.d) imply
  (26 <= gc and gc <= 31)
```

5 Formalization in UPPAAL

In this section, we shall formalize the system in UPPAAL. We start with an overview of the components and their interaction via channels and shared variables. Then we describe the IOP in detail.

5.1 Component Overview

The system consists of 7 automata, as illustrated in Figure 9. The Timer controls the time slicing between the components using the technique described in section 4.2. In addition, there is an environment which generates interrupts corresponding to data arriving on the links; hence this environment is referred to as the Interrupt Generator.

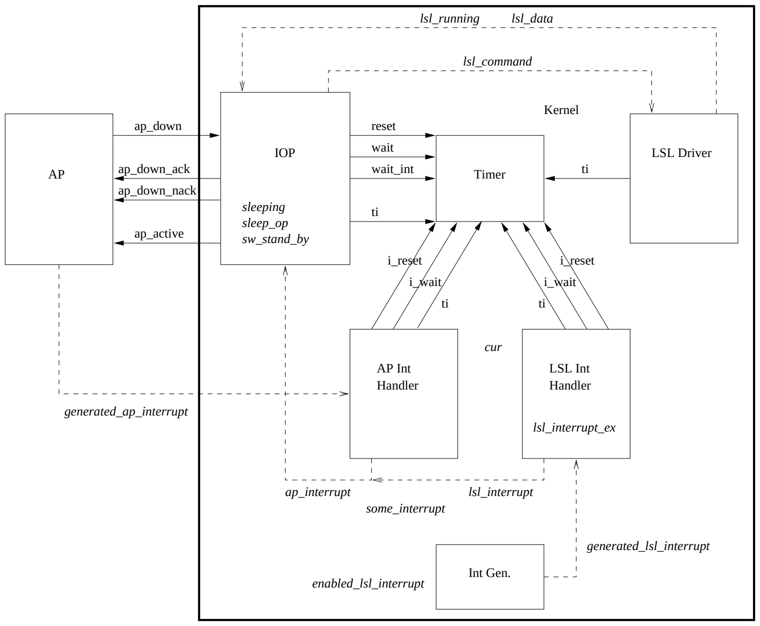

Fig. 9. The components

The components communicate via channel synchronization and via shared variables. The figure illustrates the channel connections by fully drawn arcs, each going from one component (the one that does a send "`!`") to another (the one that does a receive "`?`"). Also, all shared variables are plotted into the figure, in italics, with dotted lines indicating their role as message carriers, from the process that typically writes to the variable to the process that typically reads the variable. This notation is informal, but it should give an overview of the shared variables and the role they play in communication. Channels and variables are described below.

5.2 The Channels

The AP signals the IOP to go down by issuing an `ap_down!` (which the IOP then consumes by performing a dual `ap_down?`). The channels `ap_down_ack` and `ap_down_nack` correspond to the IOP's response to such an `ap_down` signal from the AP. They represent the acknowledgment (ack) respectively the negative acknowledgment (nack) that the closing down has succeeded respectively not succeeded. The `ap_active` channel is used by the IOP to request the AP to become active.

The channels `reset`, `wait`, `wait_int`, `i_reset`, `i_wait` are all used to operate the timer. Basically, the `reset` and `i_reset` channels are used to activate the timer, to start delivering time slots, while the `wait`, `wait_int` and `i_wait` channels are used to dis-activate the timer, to stop delivering time slots. Different channels for resetting (`reset` and `i_reset`) respectively waiting (`wait`, `wait_int` and `i_wait`) are needed due to different interpretations of these commands in different contexts.

Whenever activated, the timer then delivers time slots to the IOP, the LSL (Low Speed Link) driver, and the interrupt handlers when these issue signals on the $\mathtt{t}_i$ channels.

5.3 The Shared Variables

The interrupt generator generates interrupts corresponding to data arriving on the links. Such an interrupt is generated by setting the variable `generated_lsl_interrupt` to 1 ($true$). The LSL interrupt handler then reacts on this by interrupting the IOP or the driver, whichever is running. A result of such an interrupt is that the variable `lsl_interrupt` is set to 1. The IOP reads the value of this variable, and hence is triggered to deal with new data if it equals 1. In order for the interrupt generator to generate interrupts at all, the variable `enabled_lsl_interrupt` must be 1. Concerning the AP, there is a `generated_ap_interrupt` and an `ap_interrupt`, but there is no `enabled_ap_interrupt`. The AP itself plays the role as AP interrupt generator, and hence sets the `generated_ap_interrupt` to 1, while the AP interrupt handler reacts to this by setting the `ap_interrupt` to 1. The variable `some_interrupt` is 1 whenever either `ap_interrupt` or `lsl_interrupt` is 1.

The variable `cur` is used to secure that an interrupt handler gets higher priority than the process it interrupts. Note that in this sense, the IOP and the driver have the lowest priority (0), while the LSL interrupt handler has one higher (1), and the AP interrupt handler has the highest (2). Hence, whenever the value of `cur` is 0, the IOP and the LSL driver are allowed to execute. When the LSL interrupt handler starts executing, it sets the value to 1, whereby the IOP and driver are no longer allowed to execute. The AP interrupt handler can further interrupt all the previous processes, assigning 2 to `cur`, whereby all other processes with lower priority are denied to execute.

We said that the AP interrupt handler can interrupt the LSL interrupt handler. This is a truth with modifications. In fact, it is not allowed to interrupt during the initialization phase of the LSL interrupt handler. This is modeled by introducing a semaphore `lsl_interrupt_ex`. It is used to exclude the AP interrupt handler from interrupting the LSL interrupt handler during the latter's first activities.

The IOP sends messages to the LSL driver by assigning values to the variable `lsl_command` with the following meanings: 1 = *Initialize the driver*, 2 = *Close down the driver*, and 3 = *Activate the driver*. After initialization of the driver, the IOP can read the results of the driver's activity (whether it is still running and whether there are data or not) in the variables `lsl_running` and `lsl_data`. Since the model is a reduction from a bigger model also involving the AP driver, we had early in the design a need for maintaining a variable `some_running`, being true if either `ap_running` or `lsl_running` was true, and likewise we needed a variable `some_data`, being true if either `lsl_data` or other similar variables were true. These two variables have survived after we have reduced the model.

The three variables `sw_stand_by`, `sleeping` and `sleep_op` are central to the closing down procedure, and the interaction between the IOP and the interrupt handlers. Figure 10 illustrates the relevant pieces of code in the IOP (when approaching stand by mode), respectively the Interrupt handlers. To start with the IOP, the variable `sleep_op` is a kind of *"emergency break"* which can be "pulled" by the interrupt handler. The IOP assigns $true$ to this variable, and it has to be $true$ before going to sleep.

The interrupt handler can change the value of *sleep_op* “in last micro second”. Next, the IOP assigns $true$ to the variable `sw_stand_by` when approaching the `stand_by` node. Hence this variable is $true$ in a certain critical time zone just before closing down[8]. When the IOP finally goes down (enters the `stand_by` mode), the variable `sleeping` becomes $true$.

The value of `sw_stand_by` is used by the interrupt handlers when activated to see whether the IOP is in its critical closing down zone. If so, they assign the value $false$ to the variable `sleep_op`, and this will then prevent the IOP from going to sleep. The interrupt handlers also “wake up” (`sleeping := 0`) the IOP in case it is sleeping (`sleeping == 1`). The `sleeping` variable is used by the interrupt handler to direct the amount of time used to restart the IOP. If `sleeping == 1` it takes 900 micro seconds, otherwise it is instantaneous. We shall see the IOP algorithm formulated in UPPAAL below.

```
IOP:                              Interrupt Handler:
   sleep_op := 1;                    If sleeping == 1 Then
   sw_stand_by := 1;                   ``spend 900 ms''
   If sleep_op == 1 Then               sleeping := 0
     sleeping := 1;                  End;
     ``stand by''                    If sw_stand_by == 1 Then
   End;                                sleep_op := 0;
   ``after interrupt'':                sw_stand_by := 0
   sw_stand_by := 0                  End;
   ``go up''
```

Fig. 10. The variables `sw_stand_by`, `sleeping` and `sleep_op`

5.4 The IOP

The IOP, Figure 11, is obtained by refining (in an informal sense) the abstract model presented in Figure 3. The model is refined using *state refinement* as well as *action refinement*. By state refinement we mean that certain states (the ovals) are expanded out to sub–transition systems with new states connected with new (labeled) arcs. We have enclosed these new sub–systems in boxes on Figure 11 such that they can be easier related to Figure 3. Note, however, that this is not formal UPPAAL notation. By action refinement we mean that also arcs are expanded out to such sub–transition systems. Concerning state refinement, we have expanded each *“check driver”* state into a couple of states: `driver_call` – representing the point where a driver has been called – and `driver_return` – representing the point where the driver returns. The state *“check interrupts”* has been expanded out to a small transition system consisting of the four states: `insert_noop`, `set_stand_by`, `check_interrupts` and `check_noop`.

The IOP starts being active, in the node `active`. In this node it does not need time slots, hence the timer is supposed to be inactive. Note that although the IOP is in the node `active`, and hence intuitively is active, from a technical point of view, we don’t see it as requiring time slots, since it does not take any transitions.

[8] In the C-implementation, the variable `sw_stand_by` is a register informing the processor hardware about the approaching close down.

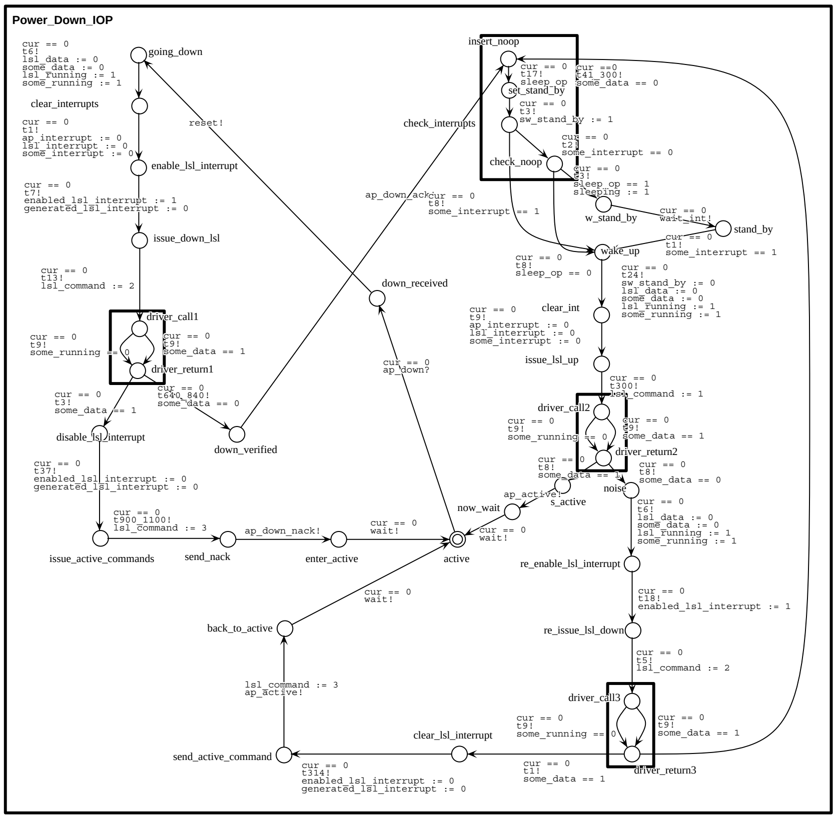

Fig. 11. The IOP

Now it can receive an `ap_down` signal from the AP, ordering it to close down. It then proceeds (up, left – referring to the approximate position on the figure) by resetting the timer – `reset!`, indicating that now it wants processor time slots necessary to close down. It then initializes the variables `lsl_running` (to 1) and `lsl_data` (to 0) preparing the activation of the LSL driver, initially assuming that there are no data. Note the "*priority 0*" guard – `cur == 0` – and the time slot demand – `t6!` – requiring 6 micro seconds to initialize these variables. The time constant, and all other time constants in the model, have been estimated by the protocol developers at B&O. When the driver later returns, it will have set the variable `lsl_running` to 0, and now the IOP can check the value of `lsl_data`. The driver is, however, first activated with the assignment of 2 (close down) to the variable `lsl_command` in the edge leading to the node `driver_call1`.

In this node the IOP waits for the driver to finish its job. If at that point, in node `driver_return1`, `lsl_data` equals 1 there is data, and the IOP must activate the driver – `lsl_command` is assigned the value 3 – and it must respond to the AP with a negative acknowledgment – `ap_down_nack!`. If on the other hand `lsl_data` equals 0, then there are no data on the link, and the IOP can proceed successfully to close down, next checking whether there are any interrupts. First, however, it acknowledges via an `ap_down_ack!` signal to the AP, and then goes to the node `insert_noop` (up, right) to check interrupts. A possible trace from here leads to the node `stand_by`, where the IOP is sleeping, and can only be wakened by an interrupt. The waiting for an interrupt is done by issuing a `wait_int!` signal to the timer just before entering the `stand_by` node. When an interrupt occurs thereafter, the timer will ensure that the IOP is re-activated immediately.

If on the other hand, before reaching the `stand_by` node, an interrupt has already occurred, then the IOP will avoid going into that node and instead go directly to the `wake_up` node. Hence, in this node we assume that an interrupt has occurred, and now the LSL driver has to be re-started, since apparently there must be data. This means re-initializing the variables `lsl_running` and `lsl_data`, and then assigning the value 1 (initialize) to `lsl_command`. In the node `driver_call2`, the IOP then waits for the LSL driver to return. If there is data – `lsl_data` equals 1 – the AP is asked to become active – `ap_active!` – and the IOP goes into the node `active`. Note that when entering this node, a `wait!` signal is issued to the timer to dis-activate it. If on the other hand there are no data – `lsl_data` equals 0 – then what has been encountered is noise, and the node `noise` is entered. In this node the IOP wants to close down, but before doing this, the driver is asked to close down – `lsl_command` is assigned the value 2. The IOP then waits in the node `driver_return3` for the drivers response.

Now, if there is data – `lsl_data` equals 1 the AP is activated – `ap_active!` – and the node `active` is entered. If on the other hand there are no data – `lsl_data` equals 0 – then the IOP returns to the node `insert_noop` (up, right), ready to check the interrupts again, and close down (if an interrupt does not occur, etc.).

Note that some transitions labeled with channel communications are not labeled with the priority guard `cur == 0`. These channels are elsewhere defined as urgent, meaning that communication must take place immediately whenever enabled.

6 Verification of Selected Properties

In this section a collection of properties will be formulated and verified using the UPPAAL logic and verification tool. In order to verify these properties, a set of techniques for annotating the model and for defining observer automata have been applied. These techniques are presented first. Then follows the formulation and verification of the individual properties of which there are 15.

6.1 Model Annotation and Test Automata

Amongst the properties formulated by B&O, in particular three kinds were typical and needed special techniques. The general principle behind the three techniques, to be described below, is to *annotate* the model by adding new variables or communication actions, and then observe these, either by mentioning the variables in the formulae to be verified (the first two techniques) or by letting the new communication actions synchronize with a furthermore added observer automaton (the third technique). The need for these techniques is caused by the existing logic in which it only is possible to state properties like: "`A[]p`" and "`E<>p`", where `p` is an atomic predicate over program variables and nodes (hence no nesting of modal operators). Theoretical as well as practical work is currently undertaken to extend the UPPAAL logic, defining translations into model annotations and observers as outlined below.

The FLAG **Technique** The first technique, called the FLAG technique for later reference, is illustrated in Figure 12. Suppose we have an automaton A containing two states (amongst others): `a` and `b`, and suppose we want to verify, that *"there is a path from* `a` *to* `b`*"*.

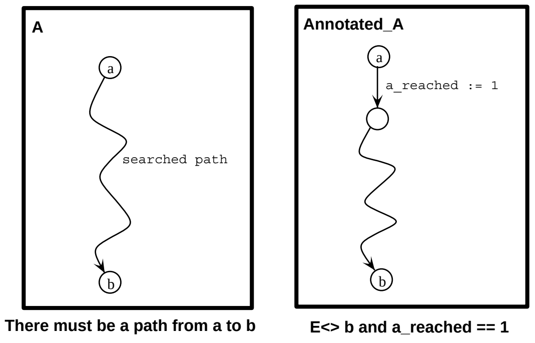

Fig. 12. Automaton A and its annotation

Note, that the current logic does not allow nested modal operators, hence it is for example not possible to state this as: "`E<> (a and E<>b)`" saying that there exists a path such that eventually node `a` is reached and from there node `b` can be reached. The technique consists of annotating automaton A, obtaining automaton Annotated_A, by adding a boolean *flag* variable `a_reached`, which initially has the value 0, and which is assigned the value 1 when passing through `a`. The property can now be formally stated as follows: "`E<>(b and a_reached == 1)`". That is, eventually node `b` is reached, after having passed through node `a`.

The DEBT **technique** The second technique, called the DEBT technique, is illustrated in Figure 13. Suppose we have an automaton B containing three states (amongst others): `a`, `b` and `x`, and suppose we want to verify, that *"every path from* `a` *to* `b` *must pass through* `x`*"*.

In an imagined extended logic this could be formulated as follows:
"`A[] (a imply ((not b) Until x))`" saying that if at any time `a` is reached, then "`not b`" will hold until `x` has been reached[9]. The technique consists of annotating automaton B, obtaining automaton Annotated_B, by adding a boolean variable `debt`, which initially has the value 0, and which is assigned the value 1 when passing through `a`. Furthermore, when passing through `x` it is reset to 0 – the debt has been "cashed". The property can now be formally stated as follows: "`A[] b imply debt == 0`". That is, if at any point node `b` is reached, then `debt` must not be 1, since that would indicate that node `a` had been reached before, but not `x` in between.

[9] Note that the `Until` operator here must be *weak* in the sense that node `x` need not be reached at all, and hence node `b` need not be reached neither, which is what we want.

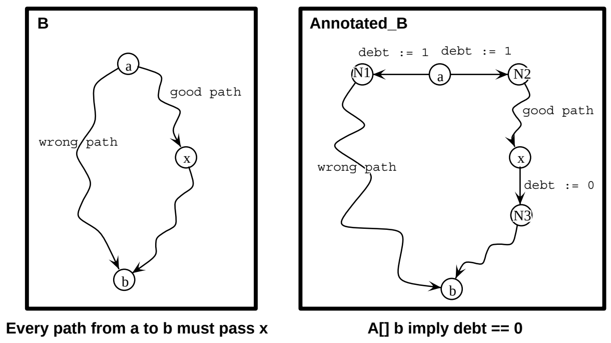

Fig. 13. Automaton B and its annotation

The OBSERVER **Technique** The last technique, called the OBSERVER technique, is illustrated in Figure 14. Suppose we have an automaton C containing two nodes (amongst others): `a` and `b`, and suppose we want to verify, that *"from node* `a`*, node* `b` *must be reached within T time units"*.

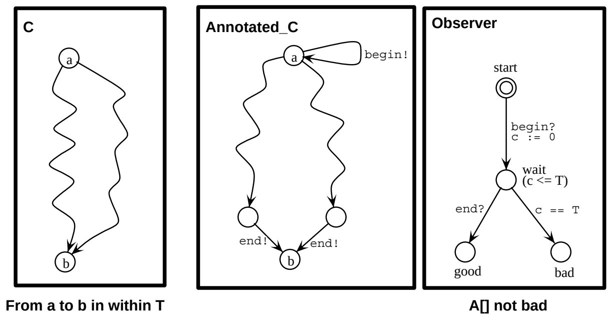

Fig. 14. Automaton C, its annotation and observer

In an extended logic this could be formulated as follows: "`A[] (a imply A<T> b)`" saying that if at any time `a` is reached, then eventually – within T time units – node `b` will be reached. The technique consists of annotating automaton C, obtaining automaton Annotated_C, by adding two kinds of communication actions, each of which communicates with an added observer that measures time. Let's first look at Annotated_C. When in node `a`, a `begin!` signal can be issued, telling the observer to start measure time. When reaching node `b`, no matter along which path, an `end!` signal is issued, telling the observer to stop measure time. The channel `end` is declared as *urgent*, hence it will be taken as soon as node `b` is reached.

The Observer automaton rests in the `start` node until it receives a `begin?` signal (node `a` reached), where after it initializes its local clock `c` and enters the node `wait` where time can progress. Time can, however, only progress T time units due to the node invariant, where after the node `bad` is entered. If on the other hand an `end?`

signal is received before that, then the node `good` is entered. The property can now be formally stated as a property of the observer: "`A[] not bad`". That is, the Observer will never reach node `bad`: an `end?` signal will always be received (`b` reached) before T time units.

6.2 Property Verification

In this section we shall present the results of analyzing in UPPAAL various desired properties. The properties as directly formulated by B&O are listed below, with explanatory comments in brackets. The listing is just supposed to give the reader a general feeling of the kinds of properties formulated.

1. `sleeping` must not change from 0 to 1 while `sleep_op` has the value 0. *(The IOP must not go to sleep if there has been an interrupt – see Figure 10 for an explanation of these variables.)*
2. There must be a path from `active` to `stand_by` and vice versa. *(It must be possible for the IOP to switch between its two final states.)*
3. Every path from `active` to `noise` must pass through `stand_by` *(The IOP must have been asleep before reaching the* `noise` *state where it on its way up due to an interrupt discovers that the interrupt is "false", and hence caused by noise only.)*
4. The variable `sleeping` must not change from 0 to 1 while `lsl_interrupt` is 1 or `ap_interrupt` is 1 *(The IOP must not go to sleep as long as there is an untreated interrupt.)*
5. The shortest way from `driver_return1` to `driver_call2` does not take more than 1500 μs *(If the IOP on its way down verifies that the link is empty by calling the driver, and then immediately thereafter data arrive (an interrupt occurs) no more than 1500 μs must pass before the driver is called again.)*
6. The shortest way from `driver_return1` to `active` does not take more than 1500 μs *(If the IOP on its way down discovers data on the link by calling the driver, then no more than 1500 μs must pass before the IOP is active again.)*
7. The shortest way from `driver_return3` to `driver_call2` does not take more than 1500 μs *(Like 5, but in a different place in the protocol's execution.)*
8. The shortest way from `driver_return3` to `active` does not take more than 1500 μs *(Like 6, but in a different place in the protocol's execution.)*
9. If the last value of the variable `lsl_command` has been 1 or 3 (driver starting commands), then the value of `sleeping` must not change from 0 to 1 *(If the last command issued to the driver was a "start command", then the IOP must not go to sleep.)*
10. If the last value of `lsl_command` has been 3 (activate driver), then the next value must not be 1 (initialize driver), and vice versa *(In between two driver starting commands must come a driver closing command.)*
11. No more than 1500 μs must pass from an interrupt occurs until all drivers are active
12. It must be possible for both interrupt handlers to want to assign 0 to `sleep_op` at the same time, while in addition this variable's value is already 0 *(Intuition missing – "technical" property.)*
13. If both interrupt handlers want to assign 0 to `sleep_op` at the same time, then the IOP will be in one of the nodes: `set_stand_by`, `check_interrupts`, `check_noop`, `w_stand_by`, `stand_by`, or `wake_up` *(If both an LSL and an AP interrupt occur, and both interrupt handlers believe that the IOP is approaching stand by mode, then this is the case.)*
14. It must be possible to come from the node `noise` to the node `stand_by` *(In case IOP has discovered noise on the link, it will reach stand by mode and go to sleep, unless data arrive.)*

15. I should not be possible to come from the node `stand_by` to the node `active` without synchronizing on the channel `ap_active` *(The IOP cannot get from stand by mode to active mode without activating the AP.)*

Figure 15 shows the verification results, indicating the outcome (satisfied or not) and the verification technique used. Those properties not verified using any of the three techniques outlined in section 6.1 have been verified using other and simpler techniques: "*trivial*" means the property was seen correct without verification. "*formula*" means that the property could be directly stated in the UPPAAL temporal logic. Finally, "*formula + aux. variable*" means that by adding an additional variable being updated in appropriate places, the property could be directly stated in the UPPAAL temporal logic. The properties were verified using UPPAAL version 2.17 from March 1998, on a Sun Ultra Sparc 60 with 512 MB main memory.

No.	Satisfied?	Technique	Comment	Memory (MB)	Time (min:sec)
1	YES	trivial			
2	YES	FLAG		5.3	0:5
3	NO	DEBT	should not be satisfied	4.1	0:2
4	YES	formula		8.2	0:9
5	NO	OBSERVER	18 AP interrupts causes error	36.0	1:42
6	NO	OBSERVER	24 AP interrupts causes error	22.0	0:56
7	?	OBSERVER	state explosion		
8	NO	OBSERVER	79 AP interrupts causes error	157.0	33:39
9	YES	formula + aux. variable		8.3	0:9
10	YES	formula + aux. variable		8.7	0:25
11	YES	OBSERVER		16.0	0:41
12	NO	formula	should not be satisfied	7.9	0:8
13	YES	formula		8.2	0:9
14	YES	FLAG		8.0	0:8
15	YES	trivial			

Fig. 15. Verification results

Properties 3 and 12 turned out not to be satisfied, and after having examined the error traces B&O recognized that these properties were wrongly formulated and hence the "error" traces showed valid behaviors.

Properties 5–8, on the other hand, are interesting in the sense that their verifications failed and caused B&O to reconsider their design. In particular property 5 gave an error trace, where a single LSL interrupt and 18 AP interrupts, all consuming time, are generated before the next driver call. As a result, B&O decided to only allow one AP interrupt to occur in their implementation.

7 Conclusion

During a period of 3 weeks, a model of B&O's Power Down protocol was developed and verified using the UPPAAL language and model checker. The first week consisted of an intense collaboration between AAU and B&O, where the B&O representative visited AAU. During this week, a first sketch of the model was written down in UPPAAL's language. The model was based on an initial design sketch made by the company representative. The work carried out during the following two weeks was mainly carried out by AAU. Hence, during the second week, a technique was introduced for dealing with timed transitions and interrupts. During this same week, the model was reduced by omitting certain components in order to obtain a model being verifiable within reasonable time and memory space. In other words, at the end of the second week, a model was produced that was ready for verification. At the beginning of the third (and last) week, various properties to be verified were formulated by B&O in natural language. These were then translated into the UPPAAL temporal logic, together with various modifications to the model, and all verifications were then carried out.

After the collaboration, the company made a C-code implementation, and after a testing phase (which did not reveal any design errors), the implementation is by now ready to be put into operation in the new company product.

During the development of models, we found that the notion of timed automata and their graphical representation served extremely well as a communication medium between the industrial protocol designer and the tool expert doing the simulation and verification. In addition, the graphical simulation features of UPPAAL lead to fast detection of (obvious) errors in the early models.

The protocol was verified correct wrt. the 15 properties formulated by B&O, and although no bugs were identified, various critical time constants were identified, which should be obeyed in order to keep the protocol correct. Various unexpected, but correct, behaviors were furthermore demonstrated, challenging the understanding of the protocol. Overall, the experience appeared to increase B&O's confidence in their design. The fact that 3 errors were caught during the modeling phase suggests that just specifying a system can be very informative. In fact, B&O claimed they had got a better understanding of their system this way.

The collaboration has been beneficial for both partners: B&O now considers tools like UPPAAL as viable means to improve the design process for time-critical software. Also, in order to model the system, we have developed techniques for modeling timed transitions and prioritized interrupts. A timed transition is a transition which consumes time, like code in a program which takes time to execute. It is a special circumstance, that several processes run on a single processor. To the best of our knowledge, such techniques have not been presented elsewhere.

What concerns the UPPAAL tool set, we anticipate investigating techniques for version control, (keeping track of several related models), and we consider tool support for defining abstractions. Both themes appear non-trivial in fact. Concerning the UPPAAL language, a technical contribution of the work is a way of modeling timed transitions and interrupts in a setting where several processes share one processor. In the forthcoming new version of UPPAAL, the introduction of *parameterized* timed automatons will support a more structural way to define time consuming transitions than we have presented in this paper. In [11], the problem of supporting task scheduling is treated. It is likely that this work will be included in later versions of UPPAAL.

In this work, we have sketched a number of patterns which may be used to define properties of real-time systems. In [1, 2] the limits of UPPAAL's model checking language are characterized. In future versions of UPPAAL, its timed logic will be modified according to these results - thereby supporting the definition of the patterns in a more direct way.

Acknowledgments The B&O representative was Johnny Kudahl, who we thank for being extremely collaborative and productive, as well during the model building as in formulating the properties to be verified. Also thanks to the reviewers.

References

1. L. Aceto, A. Bergueno, and K. G. Larsen. Model Checking via Reachability Testing for Timed Automata. In B. Steffen, editor, *Proceedings of TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280, 1998.
2. L. Aceto, P. Bouyer, A. Burgueno, and K. G. Larsen. The Limit of Testing for Timed Automata. In *Proceedings of FST TCS'98*, Lecture Notes in Computer Science, 1998.
3. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for Real-Time Systems. In *Proc. of Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.
4. R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, 1990.
5. J. Bengtsson, D. Griffioen, K. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In *Proc. of CAV'96*, volume 1102 of *Lecture Notes in Computer Science*. Springer–Verlag, 1996.
6. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — A Tool Suite for Symbolic and Compositional Verification of Real-Time Systems. In *Proc. of the 1*st *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*. Springer–Verlag, May 1995.
7. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Proc. of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, March 1996.
8. A. Bouali, A. Ressouche, and V. Roy R. de Simone. The FC2Toolset. *Lecture Notes in Computer Science*, 1102, 1996.
9. P.R. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans. Modelling and Verifying a Bounded Retransmission Protocol. *In Proc. of COST 247, International Workshop on Applied Formal Methods in System Design*, 1996.
10. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.
11. C. Ericsson, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of Nordic Workshop on Programming Theory*, 1998. To appear in a special issue of Nordic Journal of Computing.
12. K. Havelund, K. G. Larsen, and A. Skou. Documentation of the Modeling and Verification of Bang & Olufsens's IOP Power Down Module in UPPAAL. Internal AUC document delivered to B&O. Early version of this report., September 1997.
13. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 2–13, Dec 1997. San Francisco, California, USA.
14. P.-H. Ho and H. Wong-Toi. Automated Analysis of an Audio Control Protocol. In *Proc. of CAV'95*, volume 939 of *Lecture Notes in Computer Science*. Springer–Verlag, 1995.
15. G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.

16. H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *The Second Workshop on the SPIN Verification System*, volume 32 of *DIMACS, Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.
17. K. G. Larsen, P. Pettersson, and W. Yi. Diagnostic Model Checking for Real-Time Systems. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.
18. M. Lindahl, P. Pettersson, and W. Yi. Formal Design and Analysis of a Gear-Box Controller. In Bernhard Steffen, editor, *Proc. of the* 4th *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems – LNCS 1384*, pages 281–297. Gulbelkian Foundation, March 1998. Lisbon, Portugal.
19. R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, 1989.
20. S. Tripakis. Timed Diagnostics for Reachability Properties. In *Proceedings of TACAS'99*, Lecture Notes in Computer Science, 1999.