Formal Verification of an Audio/Video Power Controller using the Real-Time Model Checker UPPAAL[1]

Klaus Havelund, Kim Guldstrand Larsen, Arne Skou

BRICS, Aalborg University, Denmark
{havelund,kgl,ask}@cs.auc.dk

October 6, 1997

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

Abstract

An input/output–link control protocol is modeled and analyzed in the real-time model checker Uppaal. The protocol is supposed to sit in an audio/video component and control (read from and write to) a link to neighbour audio/video components. The component may for example be a TV, and a neighbour may be a VCR. The protocol also communicates with the remote–control. The protocol is in addition responsible for the powering up and down of the component in between the arrival of data. It is this power control that is the focus of the modeling and verification demonstrated in this report. The work has been carried out in a collaboration between Aalborg University and the audio/video company B&O, which plans to incorporate the protocol as part of a new product line. The work was carried out in a limited period of 3 weeks, with an attempt to examine how well such a collaboration would proceed. The paper elaborates on the lessons learned. Amongst technical results are techniques for modeling transitions that take time, and interrupts.

Contents

Chapter 1

Introduction

Since the basic results by Alur, Courcoubetis and Dill [1, 2] 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 [5, 11, 8]. These tools have by now reached a state, where they are mature enough for application on industrial case–studies as we hope to demonstrate in this report.

One such tool is the real–time verification tool UPPAAL [5] developed jointly by BRICS[1] 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 [10, 14, 3, 4, 13, 7] which can roughly be divided in two classes: real–time controllers and real–time communication protocols.

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 analyse 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 report documents a collaboration between AUC (Aalborg University center) – under the BRICS project – and B&O on a case study based on one of the company's new designs: a protocol for audio/video power control, that is currently being designed. The protocol is supposed to sit in an audio/video component and control (read from and write to) the link to neighbour components. The component may for example be a TV, and a neighbour may be a VCR. The protocol also communicates with the remote control. The protocol is furthermore responsible for the powering up and down of the component in between the arrival of data. It is this power control that is the focus of the modeling and verification.

The collaboration between B&O and AUC spanned 3 weeks (4 including report writing), and was very intense the first week, where a representative from B&O visited AUC, 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 report is an intensive elaboration of the preliminary report [9]. The B&O representative was Johnny Kudahl, and we thank him for being extremely collaborative and productive, as well during the model building as in formulating the properties to be verified.

The work is a continuation of an earlier successful collaboration between the same two organisations, 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

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

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. This work is described in [10].

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. All analyzed properties were satisfied, although some of the verification results indicated critical timing constants that should be obeyed in order to keep the protocol correct. As a technical contribution, techniques for modeling timed transitions and interrupts are presented. 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 processes run on a single processor.

The report is structured as follows. In chapter 2 we present the UPPAAL modeling language and tool. In chapter 3 we present some techniques for modeling timed transitions and interrupts in the UPPAAL language. Chapter 4 contains an informal description of the B&O protocol. Chapter 5 presents the formal modeling of this protocol in the UPPAAL language, while chapter 6 presents the verification results. Chapter 7 provides an evaluation of the project, an evaluation by each of the participants: B&O and AUC. Finally chapter 8 contains a conclusion.

Chapter 2

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 [2] extended with 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 [10, 14, 3, 4, 13, 7].

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

- A graphical interface based on Autograph [6] 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 2.1. 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.

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 [15] synchronization with another component on a complementary action[1], 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 2.1. Here the edge between `A0` and `A1` can only be taken, when the value of the clock `y` is greater than or equal

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

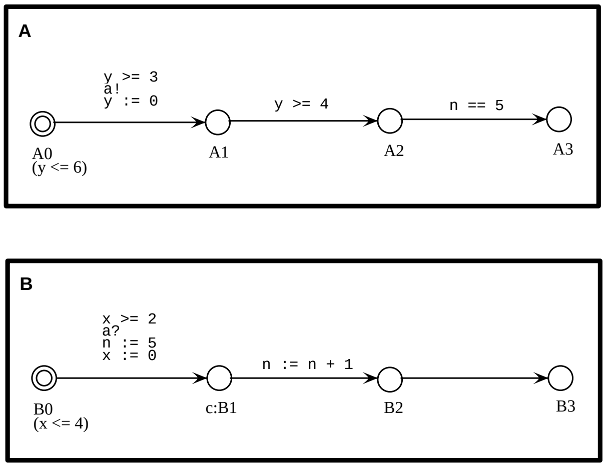

Figure 2.1: An example UPPAAL model

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 2.1, 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[2] 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 2.1, from the initial state $\langle(\texttt{A0},\texttt{B0}), \texttt{x} = 0, \texttt{y} = 0, \texttt{n} = 0\rangle$ time may elapse 3.5 time units leading to the state $\langle(\texttt{A0},\texttt{B0}), \texttt{x} = 3.5, \texttt{y} = 3.5, \texttt{n} = 0\rangle$. 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 $\langle(\texttt{A0},\texttt{B0}), \texttt{x} = 3.5, \texttt{y} = 3.5, \texttt{n} = 0\rangle$ the two components can synchronize on `a` leading to the new state $\langle(\texttt{A1},\texttt{B1}), \texttt{x} = 0, \texttt{y} = 0, \texttt{n} = 5\rangle$ (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 $\langle(\texttt{A1},\texttt{B1}), \texttt{x} = 0, \texttt{y} = 0, \texttt{n} = 5\rangle$, the B–component can perform without synchronizing with A, leading to the state $\langle(\texttt{A1},\texttt{B2}), \texttt{x} = 0, \texttt{y} = 0, \texttt{n} = 6\rangle$.

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 any action transition (synchronizing or not) *must* involve the particular component (the component is so–to–speak committed to continue). In the state $((\texttt{A1},\texttt{B1}), \texttt{x} = 0, \texttt{y} = 0, \texttt{n} = 5)$ `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

[2] indicated graphically by a double circled node

be performed atomically. Besides ensuring atomicity, the notion of *committed* nodes also helps in significantly reducing the space–consumption during verification.

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 [3] using UPPAAL.

Chapter 3

Timed Transitions and Interrupts

In this chapter, 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 *interrupts*. We refer to time slicing as the activity of delivering 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.

3.1 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 3.1, which is *not* expressed in the UPPAAL language, but rather some informal extension of the language.

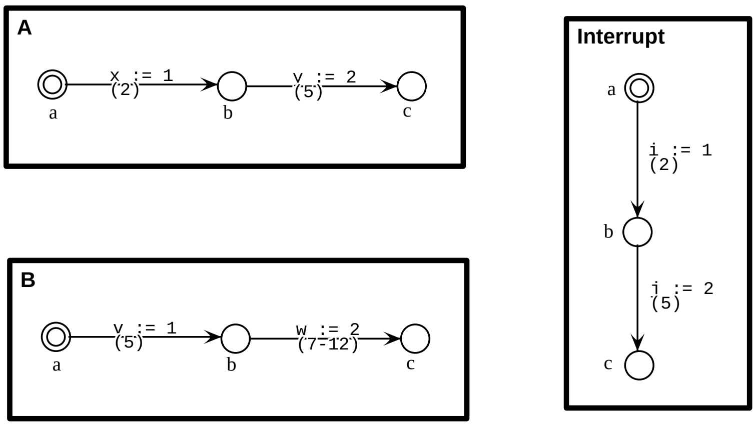

Figure 3.1: What we want to express

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 labelled 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 cannot interrupt. Then the semantics should be the following: A and B execute in an interleaved manner – each transition taking the amount of time it is labelled with. No unnecessary 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.

3.2 A Wrong Solution to Timed Transitions

First, we shall try to model time consuming transitions, ignoring the interrupts for a moment. For illustrative purposes we shall first provide a naive, and wrong, solution adding time consumption to a UPPAAL graph in the standard way, by annotating nodes with time constraints. This is illustrated in figure 3.2, which is now expressed in the UPPAAL language, like every other automata that follows from now on.

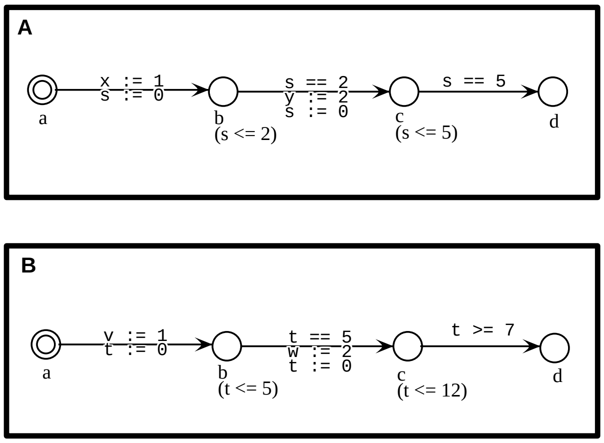

Figure 3.2: A wrong solution

The solution is based on introducing a clock for each automata A (`s`) and B (`t`). Each edge can in addition to the variable modification be labelled with a guard on the clock and a initialization to 0 of the clock. The guard expresses the time consumed by the *previous* edge. Nodes are furthermore labelled with time constraints: the time consumed by the edge leading to the node. For example, consider automata A. When it leaves node `a` it initializes the clock `s`, and node `b` now represents the time consumption of that edge. When the time has been consumed, (`s` equals 2) the next edge is taken, and so on.

This solution does, however, not work since the two automata may consume time "together". For example, automata A may enter the node `c` and automata B may enter node `b`, and they may then consume 5 time units "in parallel". This corresponds to the fact that the A edge leading to `c` (`y := 2`) executes in true parallel with the B edge leading to `b` (`v := 1`). This does not reflect the desired behaviour, since A and B are supposed to run on a single processor, and their edges must therefore execute interleaved.
We can demonstrate this by verifying the following desired – but in this model false – property:

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

That is, when A and B both reach node d, at least 19 ($2+5+5+7$) time units must have passed. The UPPAAL model checker rejects this property returning an error trace describing the above situation.

3.3 Modeling Timed Transitions

The problem with the above solution is, that each process has control over time. In a single processor setting it is natural to handle over time control to a single "operating system" process. Figure 3.3 illustrates such a process, called Timer, using a local clock k.

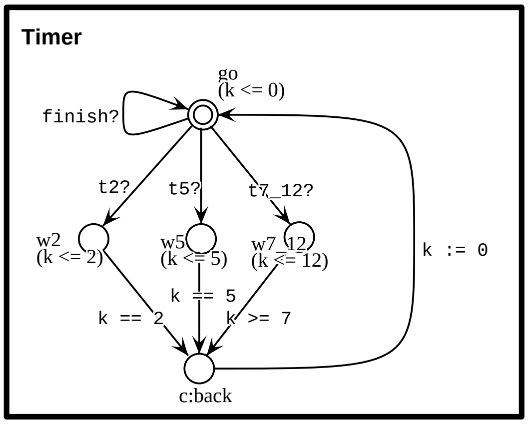

Figure 3.3: The Timer

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 3.4. These now communicate with the Timer, asking for time slots. Every time unit T that in the informal model, figure 3.1, 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.

When A reaches the node c the first time, 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 the property that did not hold in the naive model, namely:

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

This time, it holds. What also holds is:

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

That is, if both A and B reach node d, then they will do so within 24 time units. Note that due to the design of the Timer, time cannot progress further when that happens (the Timer will be in

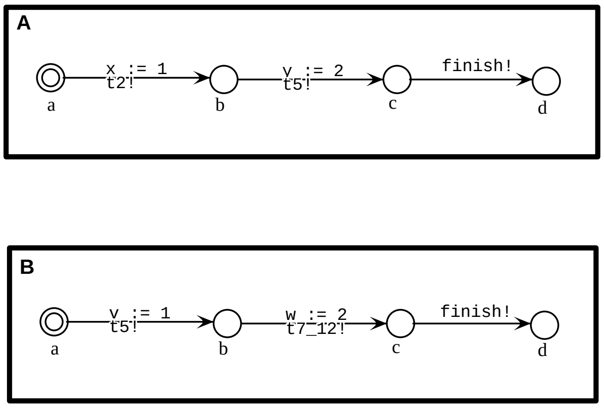

Figure 3.4: A and B communicating with the Timer

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, and which will be explained. Basically one introduces an idle node in the Timer, that can be entered upon request, and where time can progress without constraints.

3.4 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 3.5. The Timer stays unchanged.

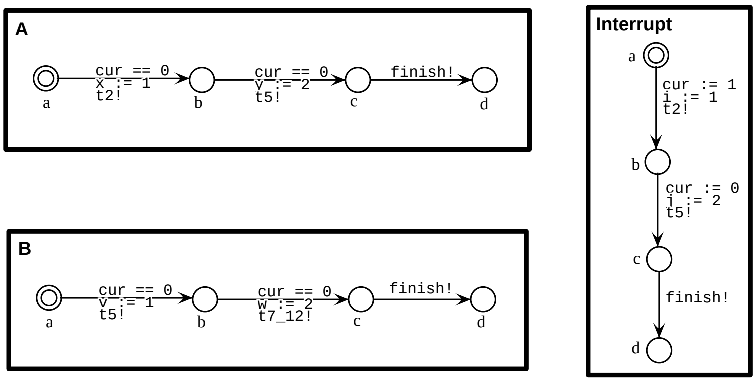

Figure 3.5: Dealing with interrupts

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
      (gc >= 26 and gc <= 31)
```

3.5 Test Automata

In this section we shall shortly describe how we can formulate and verify bounded liveness properties about a model like the one just presented. Recall the previous properties that we showed, they all had the form of a safety property:

At any moment, if in state S then P holds

However, nothing guarantees that we reach state S. For this we need a bounded liveness property like:

Within T time units state S will be reached

We shall illustrate how this is expressed in UPPAAL. Suppose we want to prove that process B will be in node `d` within at most 31 time units. For this purpose, we add a new node (`e`) to B, and an edge from `d` to `e` labelled with the signal `obs_end!` (end of observation), see figure 3.6.

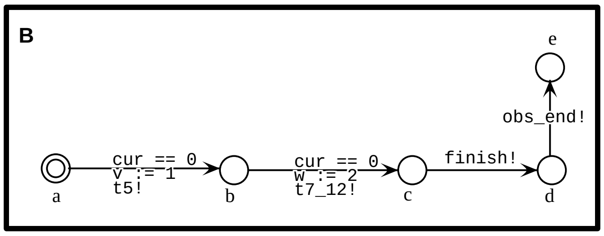

Figure 3.6: B modified to communicate with an observer

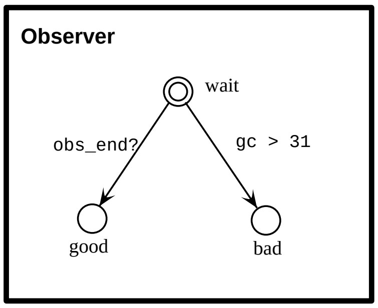

Figure 3.7: The Observer

This signal will then be received by an observer process, that we add to the system, see figure 3.7. The channel `obs_end` is declared as urgent, hence it will be taken as soon as B is in node `d`. The observer waits in the node `wait` until it either receives this signal or until more than 31 time units have passed. If more than 31 time units pass before `B` reaches node `d` (and signals `obs_end!`), the observer enters node `bad`. The property to be verified is then that the observer is never in node `bad`:

```
A[] not Observer.bad
```

The UPPAAL model checker verifies this property to be true. If we, however, modify the property, replacing 31 with for example 30, then the property no longer holds, and UPPAAL will generate an error trace showing the sequence of events that prevents B from reaching node `d` within 30 time units.

Chapter 4

Informal Description of the Power Down Protocol

In this chapter, we provide an informal description of the power down protocol. As advocated in [12], we divide the description into environment, syntax, and protocol rules.

4.1 Protocol Environment

A typical B&O configuration (see figure 4.1) consists of a number of components, which are interconnected by different kinds of links carrying audio/video data and (or) control information. Some of the components are intelligent (masters) in the sense that they control a number of (slave) components and also communicate with other masters. Each master is equipped with a dual processor system controlling audio/video devices and links, and among other tasks, the dual system must minimize the energy consumption when it goes stand by. Due to physical laws[1] the dual system cannot enter stand by mode via one atomic action, and the purpose of the present 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, this is also recognized by the other processor. Furthermore, whenever a master processor senses valid data on an external link, it must leave stand by operation.

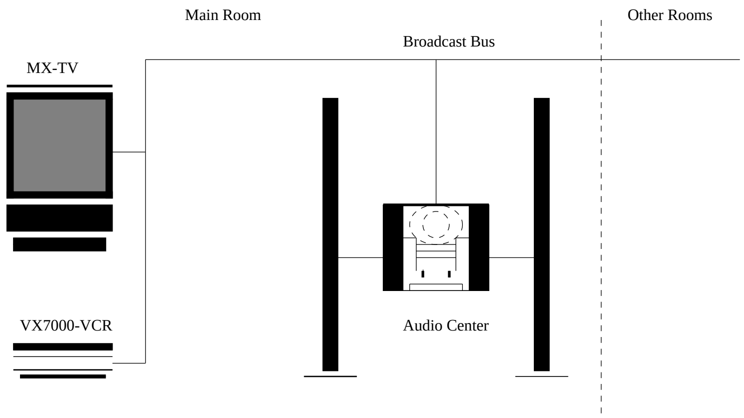

Figure 4.1: Example B&O Configuration

Figure 4.2 illustrates the dual processor system. Each processor communicates with devices and masters via external links, and the two processors are interconnected via an internal DMA link.

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

The AP processor acts as a master in the sense that it can command the IOP processor to enter stand by. In this report, we describe a model for the power down protocol for the IOP processor. However, it is foreseen that the model for the AP protocol will be almost identical.

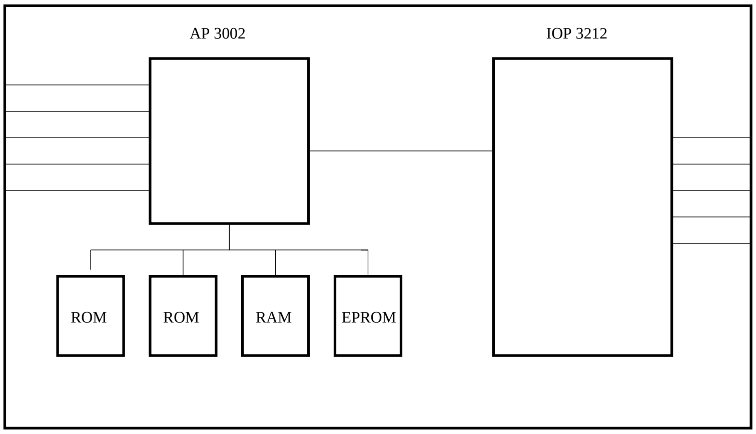

Figure 4.2: The B&O Dual Processor System

4.2 Protocol Syntax

The power down protocol entity in a IOP processor communicates with its environment (AP processor and external links) via the following protocol commands:

- {AP_down, AP_down_ack, AP_down_nack, AP_active, L_start, L_activate, L_stop, L_running, L_data, L_interrupt}

The L prefix above is generic in the sense that there is a set of commands for each external link. The AP_down command orders the IOP processor to enter standby operation, whereas the AP_active command indicates (to the AP) that the IOP has left standby. The L_interrupt command indicates that an interrupt has been received from the link, whereas the remaining L_commands are applied for controlling the link driver (start,stop,status etc).

4.3 Protocol Rules

In order to give an intuition on 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 driver communication, which will be described in the formalization chapter. In order to structure the description, we define the following meta phases (see figure 4.3 below) for the entity: The *active phase*, where the IOP is in normal (active) operation, the *going_down phase*, where the AP processor has ordered the IOP to enter the *stand_by phase*, and the *going_up phase*, where the IOP processor has decided to enter the *active phase* because it has sensed activity on one of its links.

Active rule In the active phase, the IOP protocol entity must enter the going_down phase, whenever a AP_down command is received from the AP processor

Going_down rule In the going_down phase, the IOP protocol entity must ensure that all link drivers are inactive, and that there are no pending link interrupts. If this is fulfilled, the entity must send an acknowledge to the AP processor and enter the stand_by phase (and set the processor in stand by mode). Otherwise, the entity must send a negative acknowledge to the AP processor, and reenter the active phase.

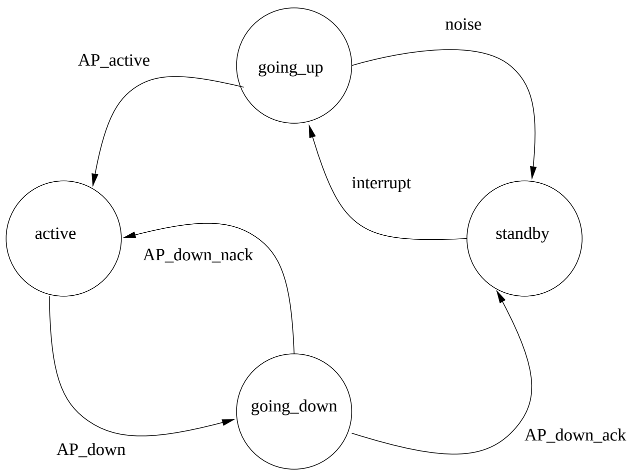

Figure 4.3: Major Protocol Phases

Stand_by rule Whenever an interrupt is received in the stand_by phase, the IOP protocol entity must start all link drivers and enter the going_up phase.

Going_up rule In the going_up phase, the protocol entity must check if the invoking interrupt (in the stand_by phase) was caused by signal noise on the link. If this is the case (and there are no other interrupts in the system), the stand_by phase is reentered.[2] Otherwise, an active command must be sent to the AP processor, and the the active_phase must be entered.

In the analysis chapter, we present a complete list of protocol requirements in terms of properties of the formal protocol model. These detailed requirements subsumes the informal rules described above.

[2]Noise is recognized via protocol command exchange with the link drivers. For simplicity reasons, this is described as one single transition in figure 4.3.

Chapter 5

Formalization in UPPAAL

In this chapter, 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 each component in detail.

5.1 Overview

5.1.1 The Components

The system consists of 8 automata, as illustrated in figure 5.1. The main components are the AP, the IOP, the LSL driver and the two interrupt handlers. As can be seen, several abstractions have been performed to obtain this model from the full system. First of all, focus has been put on the IOP, hence, the AP has been simplified considerably. For example, there is no AP driver, only an LSL driver. Also, the IIC component (driver as well as interrupt handler) has been left out.

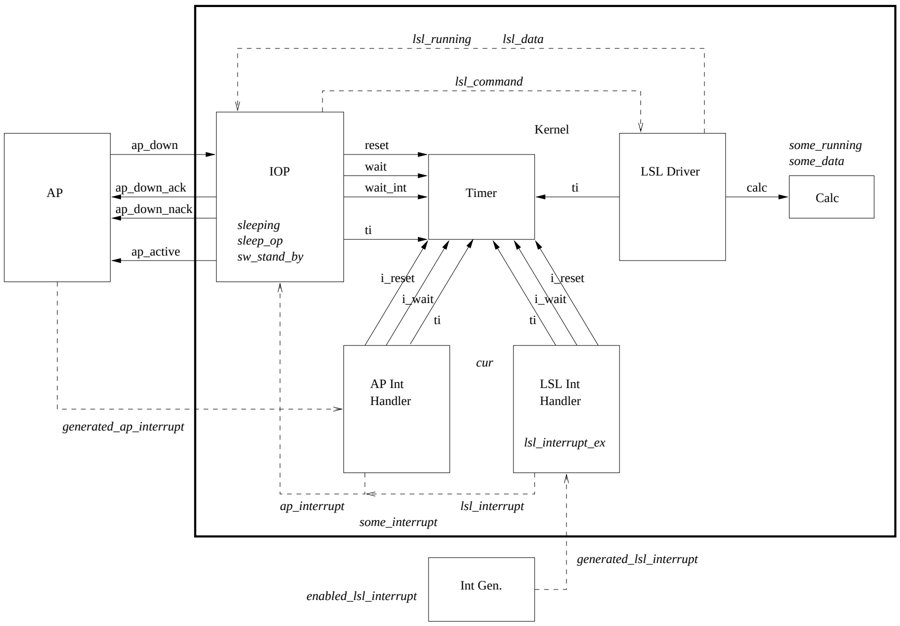

Figure 5.1: The Components

The IOP, the LSL driver and the interrupt handlers can be seen as running on one processor, hence they are grouped together on the figure, together with some additional auxiliary automata: Timer and Calc (calculator). The timer controls the time slicing between the components all using the same processor as described in section 5.9, and the calculator component represents a procedure, that is "called" by the LSL driver to perform some internal computation. We shall refer to this collection of automata as the *kernel*, all running on one processor. The kernel runs in parallel with the application process: AP. 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.

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.
We shall go though the channels and the shared variables below.

5.1.2 The Channels

AP/IOP Communication

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 acknowledgement (ack) respectively the negative acknowledgement (nack) that the closing down has succeeded. The `ap_active` channel is used by the IOP to request the AP to become active.

Timer Communication

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

Calc Communication

Finally, the `calc` channel is used by the LSL driver to "call" the calculator to perform its computation, as a procedure call.

5.1.3 The Shared Variables

Interrupts

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.

Driver Control

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
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`. These two variables have survived after we have reduced the model.

IOP Mode

The IOP can be sleeping or awake. This is modeled via the variable `sleeping`, which is 1 whenever the IOP is sleeping. The variable `sleep_op` is 1 whenever the IOP is supposed to go into sleep. Finally, the variable `sw_stand_by` is 1 whenever the IOP is supposed to go into stand by mode; that is: sleeping and waiting for an interrupt.

5.2 The IOP

The IOP, figure 5.2, 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.

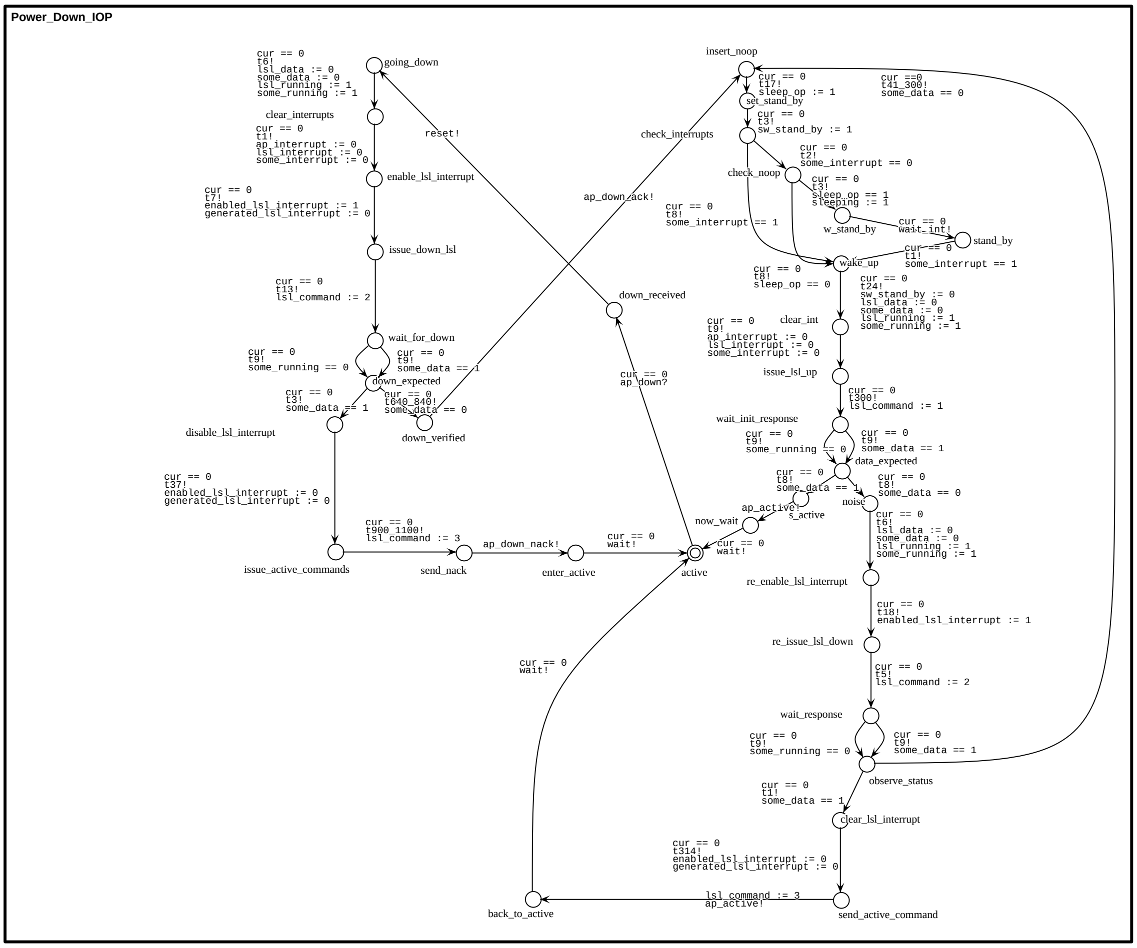

Figure 5.2: The IOP

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` equals 0 – and the time slot demand – `t6!` – requiring 6 micro seconds to initialize these variables. 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 `wait_for_down`. In this node the IOP waits for the driver to finish its job. If at that point `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 acknowledgement – `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 the closing down. It acknowledges via an `ap_down_ack!` signal to the AP and goes to the node `insert_noop` (up, right). 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 `wait_init_response`, 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 `observe_status` 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 close down (if an interrupt does not occur, etc.).

5.3 The AP

The AP, figure 5.3 has been simplified by only focusing on a few of its functionalities. First of all, it is capable of producing interrupts, by assigning the value 1 to the variable `generated_ap_interrupt`. Note that the local variable `no_ap_ints` (number of AP interrupts) keeps track on the number of interrupts generated, and in the presented model, the AP is only allowed to issue one interrupt. During the verification, we experiment with this bound, occasionally removing the bound.

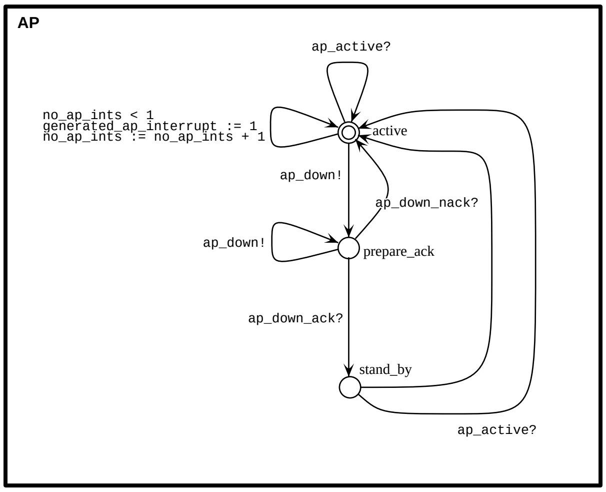

Figure 5.3: The AP

The AP can also issue an `ap_down!` signal to the IOP. If it receives a positive acknowledgement – `ap_down_ack?` – it then enters the `stand_by` node, from where it can leave in two ways. Either because it receives an `ap_active?` (become active) signal from the IOP, or because it simply non-deterministically decides to become active. This non-deterministic choice represents an abstraction from details: that certain events may occur that triggers the AP to become active.

Note that the AP runs in parallel with the kernel, and hence does not have `cur` guards and does not communicate with the timer.

5.4 The LSL Interrupt Handler

The LSL interrupt handler, figure 5.4, is triggered by the variable `generated_lsl_interrupt` becoming 1. Recall, that this variable is set by the interrupt generator, corresponding to the arrival of data on the link. The requirement that this variable equals 1 is expressed as a guard on the edge leading out from the initial node `lsl_int_service`. Another guard is that `enabled_lsl_interrupt` equals 1. Furthermore, the variable `cur` is required to be 0, corresponding to the fact, that the LSL interrupt handler can only interrupt the IOP and the LSL driver: it cannot interrupt the IIC interrupt handler (in case we had included this), and it cannot interrupt the AP interrupt handler. The value of `cur` becomes 1 to model the fact that neither the IOP nor the LSL driver can now execute. The variable `lsl_interrupt_ex` is assigned to 1 to mutually exclude the AP interrupt handler to interrupt while the LSL interrupt handler starts executing. It is assigned to 0 again a few edges later. Finally, the timer is reset – `i_reset!` – to measure time and deliver time slots. Another reset channel is used than the one the IOP uses (which is `reset`), this will be explained in section 5.9.

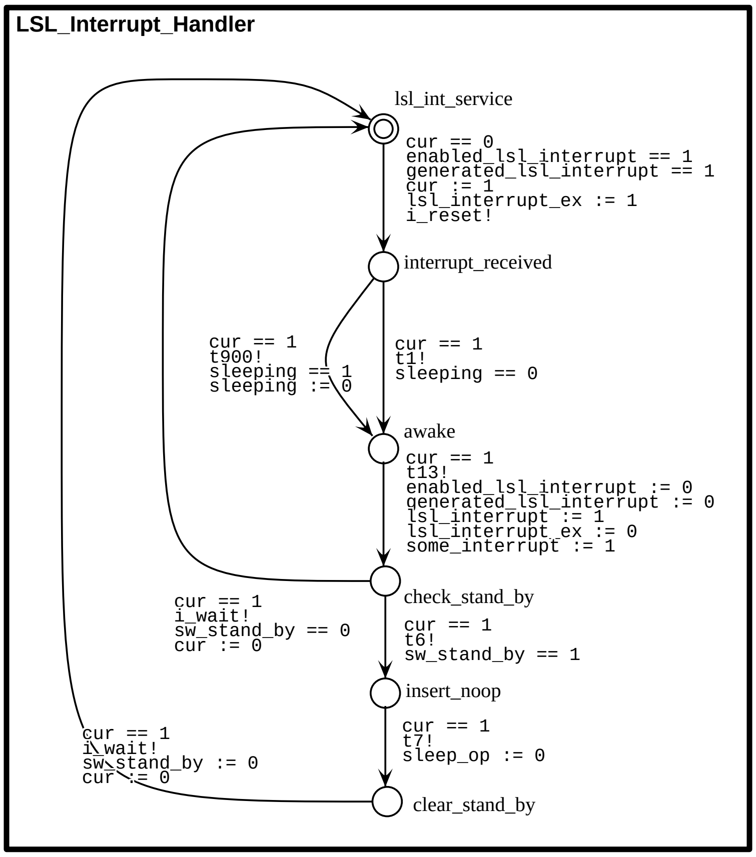

Figure 5.4: The LSL Interrupt Handler

From the node `interrupt_received` there are two edges, depending on the value of `sleeping`. If this variable equals 1 (the IOP is sleeping) then 900 μs is used to restart the IOP.

In the next edge, from the node `awake`, the `lsl_interrupt` variable is assigned the value 1 (together with `some_interrupt`) such that the IOP can detect the interrupt. Also, the AP interrupt handler is now allowed to interrupt, by setting the variable `lsl_interrupt_ex` to 0. What happens next is that in case the IOP is in stand by mode – `sw_stand_by` equals 1 – and hence is waiting for an interrupt, this situation is changed now. The variable `sleep_op` is assigned 0 to indicate that the IOP should not sleep now.

5.5 The AP Interrupt Handler

The AP interrupt handler, figure 5.5, is similar to the LSL interrupt handler, though there are a few differences.

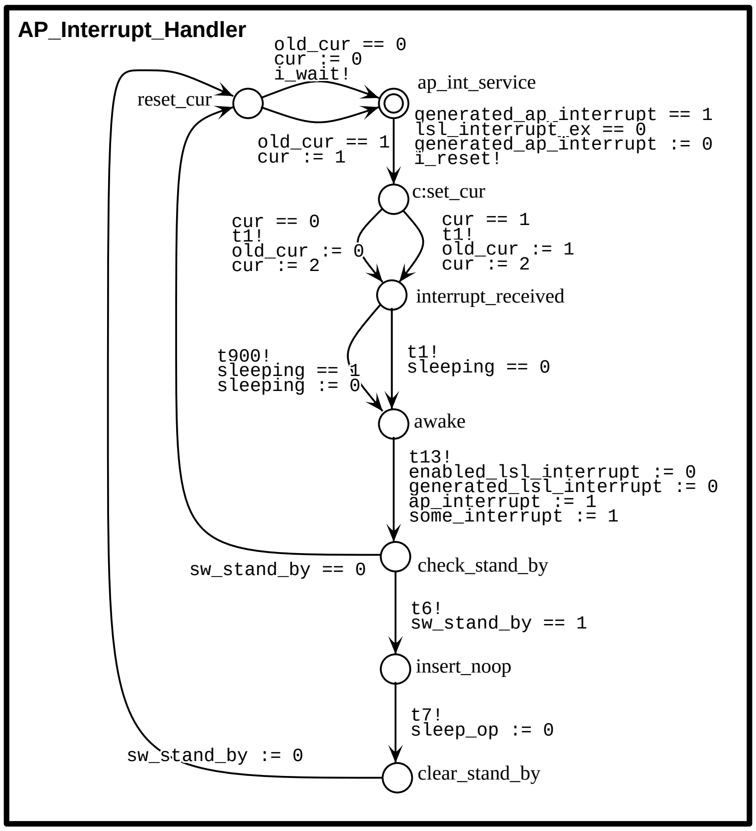

Figure 5.5: The AP Interrupt Handler

First of all, since there is no `enabled_ap_interrupt` variable there is no guard on this being 1. The AP interrupt handler is always enabled. Also, there is no guard on the `cur` variable, meaning that the AP interrupt handler basically can interrupt at any moment. However, not when the LSL interrupt handler is starting, represented by the guard requiring that `lsl_interrupt_ex` is 0. The AP interrupt handler cannot be interrupted itself, modeled by assigning 2 to `cur`, and later by disabling the LSL interrupt handler.

Second, before assigning the value 2 to the variable `cur`, the old value must be remembered – in the variable `old_cur`. This is necessary in order to later restore this value in `cur` when returning. Note that this is not necessary for the LSL interrupt handler since it can only interrupt the IOP and the driver, that is: when `cur` equals 0, while the AP interrupt handler can interrupt also the LSL interrupt handler (when `cur` equals 1). When returning, the value of `old_cur` also determines whether the timer should be signalled – `i_wait!` – since if it equals 1, it means that the LSL interrupt handler has been interrupted, and in that case the timer should just continue to deliver time slots to that. This will be explained in section 5.9.

5.6 The LSL Interrupt Generator

The interrupt generator, figure 5.6, generates LSL interrupts by assigning 1 to the variable `generated_lsl_interrupt`. Each such interrupt indicates the arrival of data on the low speed link. There is an upper bound on the number of such interrupts, in this case 2, and the local variable `no_lsl_interrupts`, keeps track on the number of interrupts generated. The bound is removed in certain of the verifications.

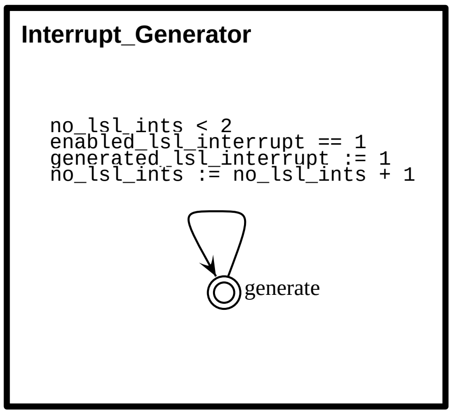

Figure 5.6: The Interrupt Generator

5.7 The LSL Driver

The LSL driver, 5.7, is triggered to execute when the IOP assigns values in the domain $\{1, 2, 3\}$ to the variable `lsl_command`. Recall the meaning of these values: 1 = "Initialize", 2 = "Close Down" and 3 = "Activate".

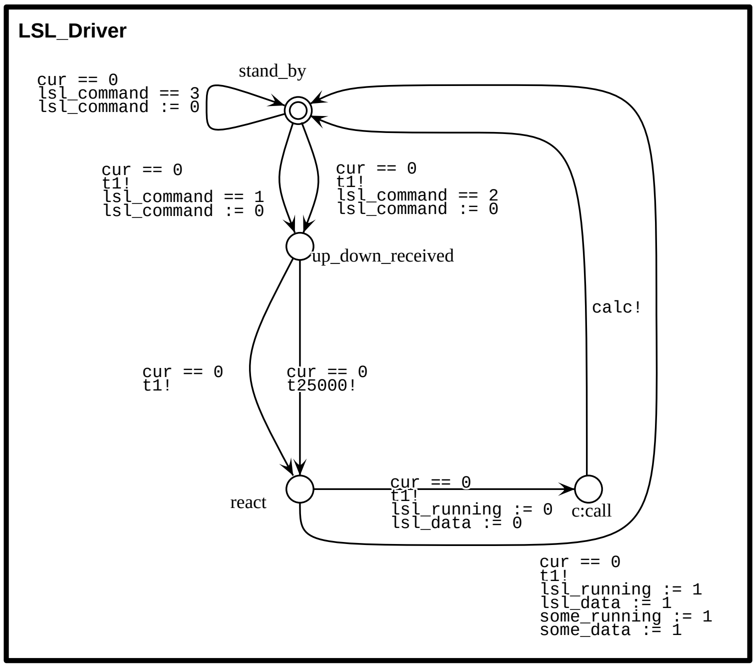

Figure 5.7: The LSL Driver

Note that the driver has lowest priority, and hence can only execute when the variable `cur` has the value 0. Now, in case `lsl_command` has the value 3 (Activate), the driver just stays in `stand_by` mode. This is an abstraction of the reality, indicating that we are not interested in the drivers response to such activation commands. When on the other hand `lsl_command` has the value 1 (Initialize) or 2 (Close Down), the driver goes to the node `up_down_received`. Note that its behaviour is identical in these two cases: it is supposed to check whether there are data on the link or not. This examination can terminate immediately (taking 1 μs in the model) or it can take 25000 μs if there are data, or noise on the link.

From the node `react` there are two edges leading out. The upper edge assigns 0 to `lsl_running` and `lsl_data`, meaning that no data have been detected on the link. The lower edge represents the fact that data have been detected, assigning 1 to these variables. The choice between these two edges is non-deterministic, corresponding to an abstraction from the details of the link: rather than modeling the arrival of data explicitly, we let the driver decide non-deterministically whether data have arrived or not.

The current version of the UPPAAL language does not allow edge guards that are disjunctions of predicates. For example, in certain places we need guards of the form (assuming a full model including the IIC component):

```
lsl_running or iic_running
```

and similarly:

```
lsl_data or iic_data
```

Since this is not allowed, we have introduced the two variables `some_running` and `some_data`, which we make sure represent the corresponding disjunctions at any moment. Hence, in guards we can instead write `some_running` and `some_data` instead of the above disjunctions. To update the variables `some_running` and `some_data`, we have introduced the component named Calc, and the driver communicates to this component via the `calc!` signal. The Calc component is only called when a *running* or a *data* variable is assigned the value 0, since when one of these is assigned 1, we know for sure that the corresponding *some* variable becomes 1.

5.8 The Calculator

The calculator, fig 5.8, calculates the value of `some_data` and `some_running` when triggered by a signal on the `calc` channel. All nodes are committed reflecting that this models a procedure that terminates instantly without consuming time. Note that the presented Calc is a reduction of a more complicated one involving also the corresponding IIC variables. Hence, in the current setting with only an LSL driver, the Calc component is not really needed.

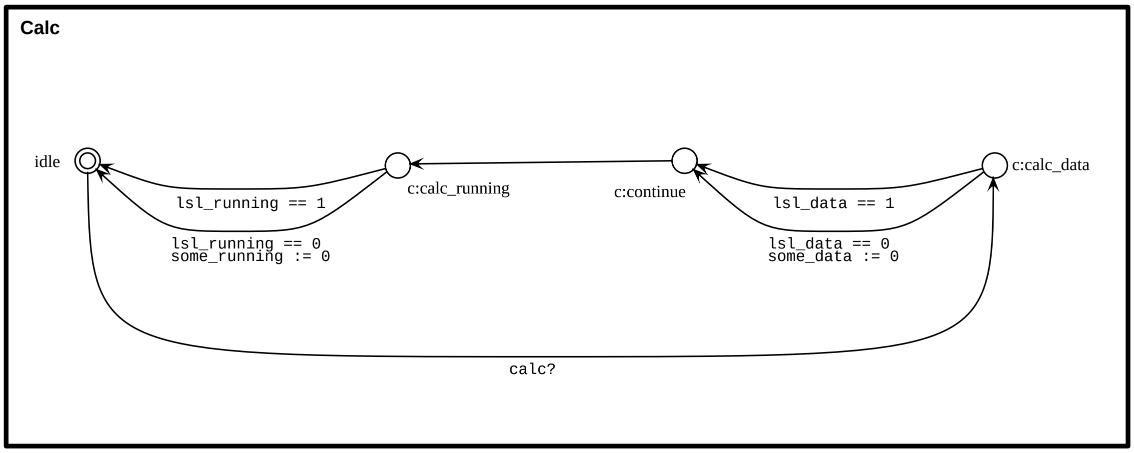

Figure 5.8: The Data/Running Calculator

5.9 The Timer

The Timer, figure 5.9, is in its basic form designed like the Timer in section 3.3, figure 3.3. That is, there is a node called `go`, in which time cannot progress, and from this leaves edges, one for each time slot to be delivered.

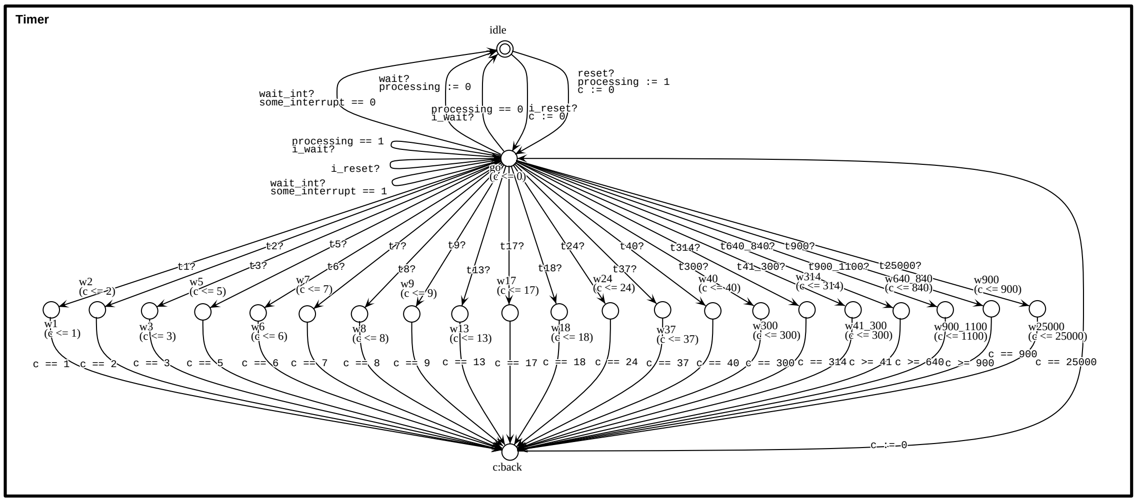

Figure 5.9: The Timer

In addition, there is a node `idle` in which time can progress without any restrictions. The idea is, that when the IOP is either active (in the node `active`) or is waiting for an interrupt (in the node `stand_by`), time can progress, and in order for that to be allowed, the timer must be "switched off" by putting it into the `idle` node. Initially the timer is in the `idle` node.

Furthermore there is a local variable called `processing`, which at any moment is *true* if and only if the IOP is waiting for an interrupt in the node `stand_by` or is powering up or down. In other words, it is *true* if the IOP is not in the node `active`. In case an interrupt occurs, and terminates, if `processing` is 1, the IOP must be started immediately. If on the other hand, `processing` is 0 when an interrupt terminates, the IOP is in the node `active`, and hence it should stay there without reacting immediately to the interrupt.

To model this behaviour, the switching on and off the timer is done by the IOP, and the interrupt handlers by communication over the following channels.
The IOP uses the channels:

```
reset
wait
wait_int
```

The interrupt handlers use the following:

```
i_reset
i_wait
```

The IOP starts the timer with a `reset!` signal (and `processing` becomes 1), and switches it off again with a `wait!` signal (and `processing` becomes 0) when the IOP goes back into the `active` node. If on the other hand the IOP goes into the node `stand_by` to wait for an interrupt, the timer is switched off with a `wait_int!` signal (which does *not* set `processing` to 0, hence `processing` will still be 1 since the IOP is not in the `active` node). Note, however, that if an interrupt has just occurred (`some_interrupt` equals 1) the `idle` node is not entered[1].

An interrupt handler resets the timer with a `i_reset!` signal. The difference between this and the `reset!` signal is, that `i_reset!` does not cause the variable `processing` to become 1, since the activation of an interrupt handler does not affect whether the IOP is in the `active` node or not. Finally, when an interrupt handler terminates, it yields a `i_wait!` signal. In case `processing` is 1, it means that the IOP needs to be started immediately, and in that case the timer stays in the `go` node. On the other hand, if `processing` is 0, the IOP must be in the `active` node, and hence should not react to the interrupt immediately, which is why the timer then goes back to the `idle` mode.

[1]This solution was introduced due to an error trace generated by UPPAAL in an early model where `wait_int!` *always* made the timer go into the `idle` node. This turned out to block the IOP.

5.10 The Configuration Declaration

The configuration declaration, figure 5.10, shows all the channels and variables with their types. They have been divided into sections, one for each component in the system. This division is informal and only shows to which component a variable belongs "most" in the authors opinion.

```
Config

// Global
int cur,
    sleeping,
    sleep_op,
    sw_stand_by;
// AP
chan ap_down;
urgent chan ap_active,
     ap_down_ack,
     ap_down_nack;
int generated_ap_interrupt,
    no_ap_ints;
// LSL_Driver
int lsl_command,
    lsl_running,
    lsl_data;
// AP_Interrupt_Handler
int old_cur,
    ap_interrupt;
// LSL_Interrupt_Handler
int enabled_lsl_interrupt,
    lsl_interrupt,
    lsl_interrupt_ex;
// Interrupt_Generator
int generated_lsl_interrupt,
    no_lsl_ints;
// Timer
chan wait, wait_int,
  i_wait, i_reset;
urgent chan reset;
chan t1, t2, t3, t5, t6, t7, t8,
  t9, t13, t17, t18, t24, t37, t40,
  t300, t314, t900, t25000,
  t41_300, t640_840, t900_1100;
clock c;
int processing;
// Calc
chan calc;
int some_running,
    some_data,
    some_interrupt;
system Power_Down_IOP, AP,
       LSL_Driver,
       AP_Interrupt_Handler,
       LSL_Interrupt_Handler,
       Interrupt_Generator,
       Timer,
       Calc;
```

Figure 5.10: The Configuration Declaration

Chapter 6

Analysis wrt. Selected Properties

In this chapter we shall present the results of analyzing in UPPAAL the properties formulated by B&O. The properties were originally named 1, 2a, 2b, 3, 4a, 4b, 5a, 5b, and 6 – 11, and we have maintained this numbering. For each property, we first quote the property as B&O originally formulated it, occasionally followed by a short explanation. Then follow the modifications to the model needed to formulate the property in the UPPAAL logic. For example, it may be necessary to add an observer, or to add new variables to the model, which then will occur in the property to be verified. This property is then formalized in the UPPAAL temporal logic. Finally, the result of the verification is described, including a commented error trace in case the property is not satisfied. In a few cases (properties 1 and 11) no verification has been done, since the property turned out to be trivially true – a fact that however only became clear to B&O after the model had been designed.

6.1 Property 1

6.1.1 Property

`sleeping` *must not change from 0 to 1 while* `sleep_op` *has the value 0.*

6.1.2 Result

The property is trivially satisfied, which can be seen by observing the transition system, since `sleeping` is only assigned the value 1 in one edge (within the IOP), and the guard for this edge is exactly `sleep_op == 1`. However, this only became clear for B&O after having built the model. It was decided not to verify the property due to its obvious truth.

6.2 Property 2a

6.2.1 Property

There must be a path from `active` *to* `stand_by` *and vice versa.*

6.2.2 Model Modifications

1. New declaration:

```
int stand_by_reached;
```

2. IOP changed: the assignment `stand_by_reached := 1` has been added to the edge from `w_stand_by` to `stand_by`, see figure 6.1 (up, right).

6.2.3 Property in Uppaal Logic

```
E<> Power_Down_IOP.stand_by

E<> Power_Down_IOP.active and stand_by_reached == 1
```

6.2.4 Result

The property is satisfied.

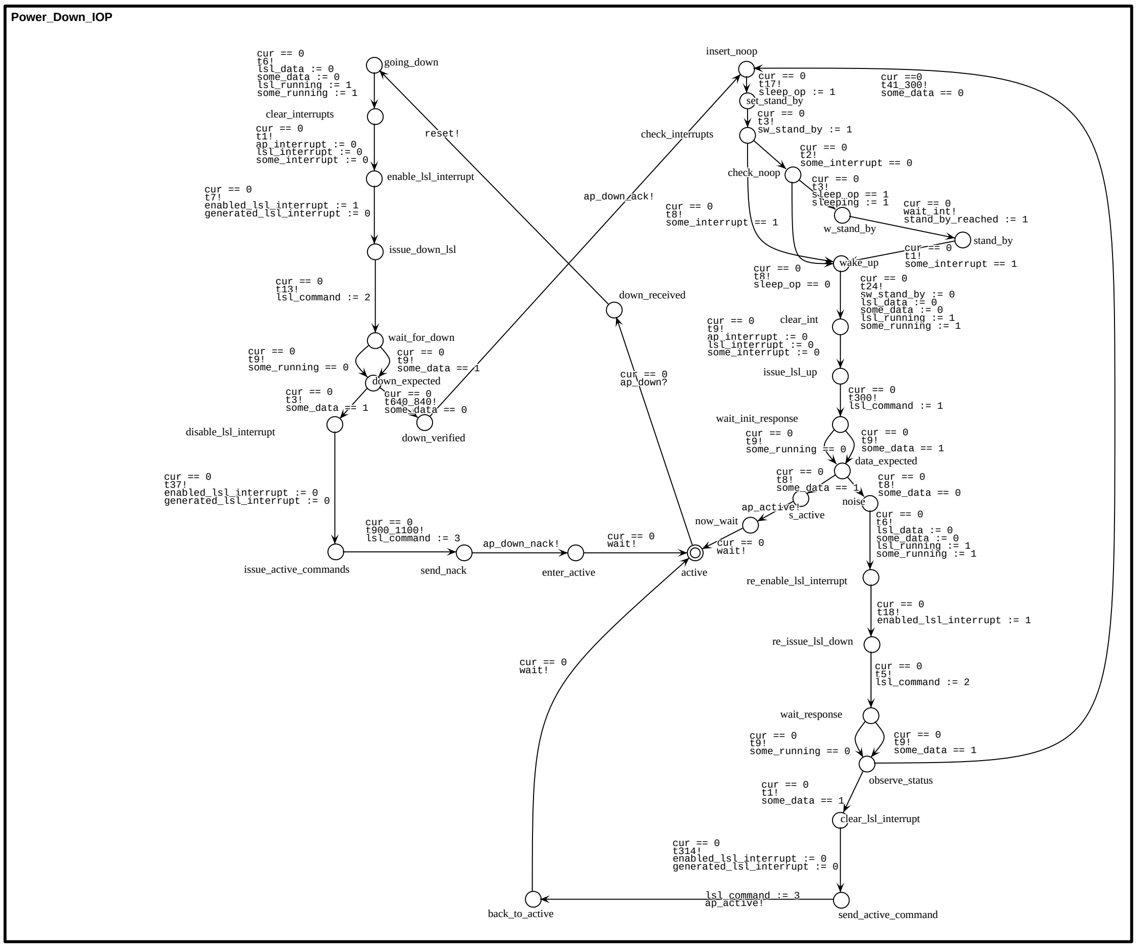

Figure 6.1: Q2a – The IOP

6.3 Property 2b

6.3.1 Property

Every path from `active` *to* `noise` *must pass through* `stand_by`.

6.3.2 Model Modifications

1. New declaration:

```
int active_debt;
```

2. IOP changed: the assignment `active_debt := 1` has been added to the edge from `active` to `down_reached`, see figure 6.2 (mid), and the assignment `active_debt := 0` has been added to the edge from `stand_by` to `wake_up` (up, right). The idea is, that when we leave from the `active` node, this variable is 1 until it becomes 0 when passing though `stand_by`. It should always be 0 then, when entering `noise`.

6.3.3 Property in Uppaal Logic

```
A[] Power_Down_IOP.noise imply active_debt == 0
```

6.3.4 Result

The property is not satisfied. It turns out that the property is not a desired property, and hence should not necessarily be satisfied.
The error trace generated demonstrates a sequence of events where the IOP reaches the node `insert_noop` (up, right) after having left the node `active`. At this position, an interrupt occurs (`some_interrupt == 1`), and the IOP is therefore directed from the node `check_interrupts` to `wake_up` without passing through `stand_by`. From there, the node `noise` is reached (and the variable `active_debt` still has the value 1 since we never took the edge leading out from `stand_by`).

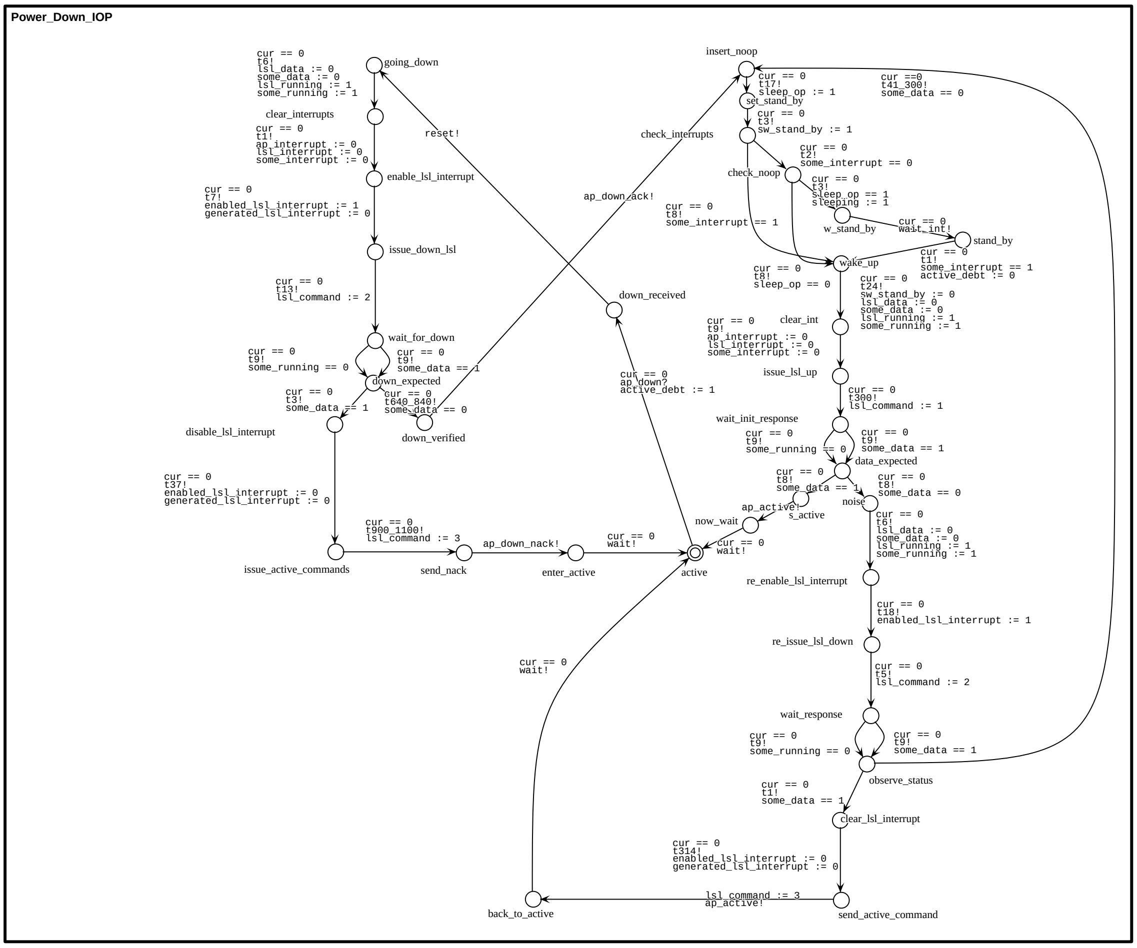

Figure 6.2: Q2b – The IOP

6.4 Property 3

6.4.1 Property

> *The variable* `sleeping` *must not change from 0 to 1 while* `lsl_interrupt` *is 1 or* `ap_interrupt` *is 1.*

6.4.2 Model Modifications

None.

6.4.3 Property in Uppaal Logic

```
A[] (Power_Down_IOP.check_noop and cur == 0 and
     sleep_op == 1 and sleeping == 0)
       imply (lsl_interrupt == 0 and ap_interrupt == 0)
```

This guard of the implication is the guard (except for `sleeping == 0`) of the only edge in the system which assigns 1 to `sleeping`, namely the edge from `check_noop` to `w_stand_by` in the IOP, see figure 5.2 (up, right).

6.4.4 Result

The property is satisfied.

6.5 Property 4a

6.5.1 Property

> *The shortest way from* `down_expected` *to* `wait_init_response` *does not take more than 1500 µs.*

We want to observe traces, which pass through the node `down_expected`, goes "right" through `down_verified` since `some_data == 0`, and then directly to `wait_init_response`. Whether the node `stand_by` is passed or not is unimportant, although an interrupt has to have occurred, otherwise the IOP will hang in the `stand_by` node. This is what is meant by *shortest way*. The time consumed on this route between `down_expected` and `wait_init_response` should not exceed 1500 µs.

Put differently, the property to verify is the following: if the IOP is in the node `down_expected`, and `some_data == 0` and `some_interrupt == 1`, then the IOP will be in the node `wait_init_response` within 1500 µs.

6.5.2 Model Modifications

1. New declarations:

   ```
   chan obs_begin;
   urgent chan obs_end;
   clock clk;
   ```

2. IOP changed: we choose the node `down_verified` as starting point for the traces we want to investigate, instead of the node `down_expected`, since we are really only interested in traces going through the former, see figure 6.3 (mid, left). In order to signal to an observer (see below) that the node `down_verified` has been entered while an interrupt has occurred, we add an edge leaving from, and going back to, this node with the label:

   ```
   some_interrupt == 1
   obs_begin!
   ```

   Second, the edge from `issue_lsl_up` to `wait_init_response` (mid, right) has been split up into two edges, inserting the new node `signal_obs`. This is done purely to add the new edge `obs_end!`, signalling the observer again, when entering the node `wait_init_response`.

3. An observer automaton has been added, see figure 6.4. As soon as this observer receives an `obs_begin?` signal from the IOP, it starts measuring the time through the variable `clk`. If more than 1200 µs passes before an `obs_end?` signal is received, the node `bad` is entered, indicating that the 1500 µs time limit will be passed before the node `wait_init_response` will be reached. The reason for only counting 1200 µs is due to the way communication with the Timer automaton works: the `obs_end!` communication will actually be performed before the time delay initiated with the preceding `t300!` communication (in the IOP) begins.

   Note the extra `obs_end?` edges leading from `start` to `good` and from `good` to `good` in the observer. These are introduced in order to avoid the observer to block the observed IOP. They have no importance for the verification otherwise.

4. The AP and the interrupt generator allow an unbounded number of interrupts, see figures 6.5 and 6.6, where the guards involving `no_ap_ints` and `no_lsl_ints`, respectively, have been removed.

6.5.3 Property in Uppaal Logic

```
A[] not Observer.bad
```

6.5.4 Result

The property is *not* satisfied. The error trace demonstrates a situation, where a single LSL interrupt and 18 AP interrupts are generated in between the nodes `down_expected` and `wait_init_response`, each interrupt consuming time. The error trace is as follows:

1. The IOP gets a `ap_down` signal from the AP, and starts the closing down procedure.
2. While the IOP is in node `enable_lsl_interrupt`, an AP interrupt occurs, just making it possible to later get around the `stand_by` node; that is, now `some_interrupt == 1`. This interrupt has no direct importance for the time failure demonstrated by this trace.
3. The node `down_verified` is entered and the observer is signalled to start counting time. The IOP continues to node `wake_up`, consuming 868 μs (840 + 17 + 3 + 8).
4. An LSL interrupt occurs.
5. The LSL interrupt is itself interrupted while in the node `insert_noop`, by an AP interrupt which consumes 28 (1+1+13+6+7) time units, since `stand_by == 1`. Then follows 16 more AP interrupts, each consuming 15 (1+1+13) time units, since now `stand_by == 0`, yielding 240 μs all together.
6. After all the AP interrupts, the LSL interrupt continues, and returns. The LSL interrupt itself has consumed 27 μs (1 + 13 + 6 + 7).
7. The IOP now continues to the node `clear_int`, consuming 24 μs.
8. Now yet an AP interrupt occurs, and consumes 13 μs(plus more of course).
9. At this point, 1200 μs have passed, and the observer automaton goes into the `bad` node.

The verification takes 45 minutes.

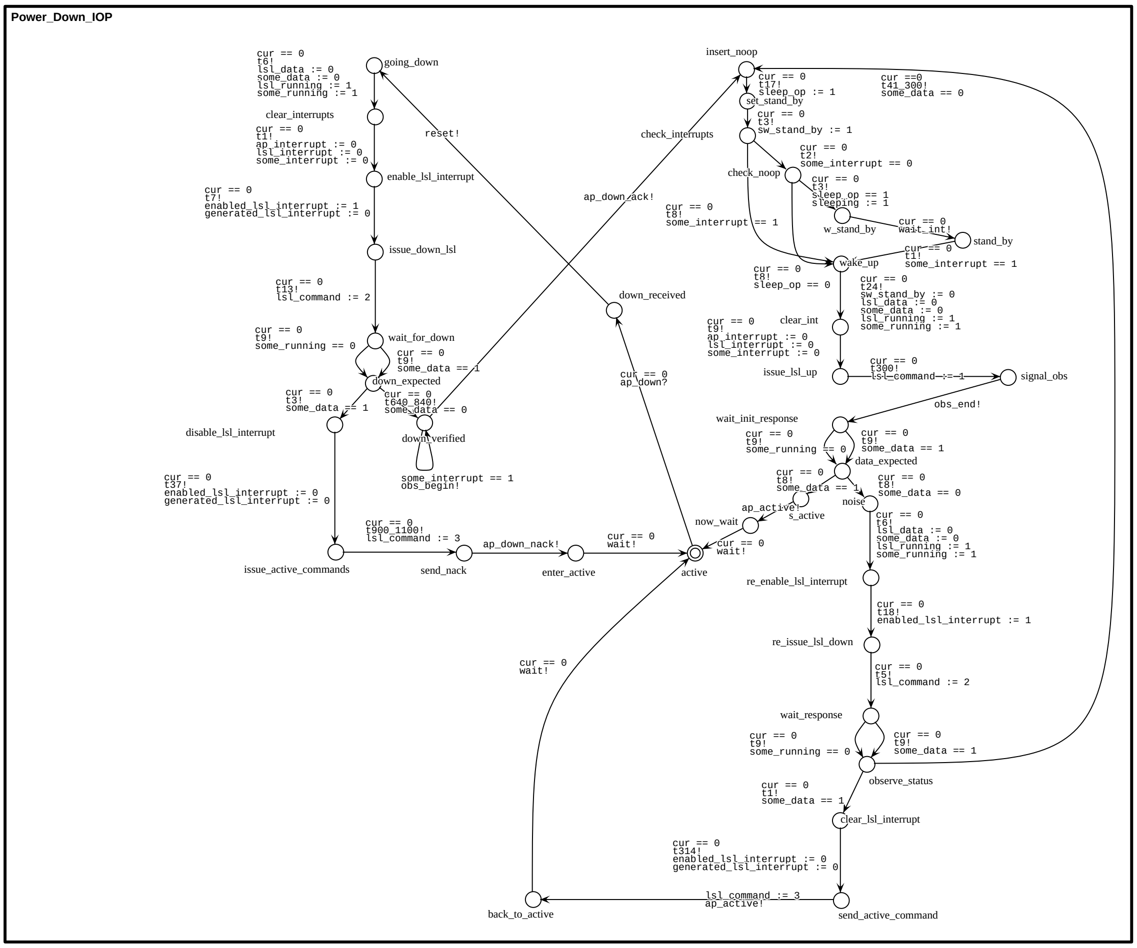

Figure 6.3: Q4a – The IOP

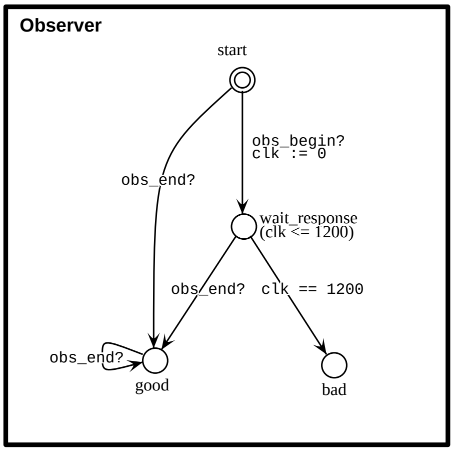

Figure 6.4: Q4a – The Observer

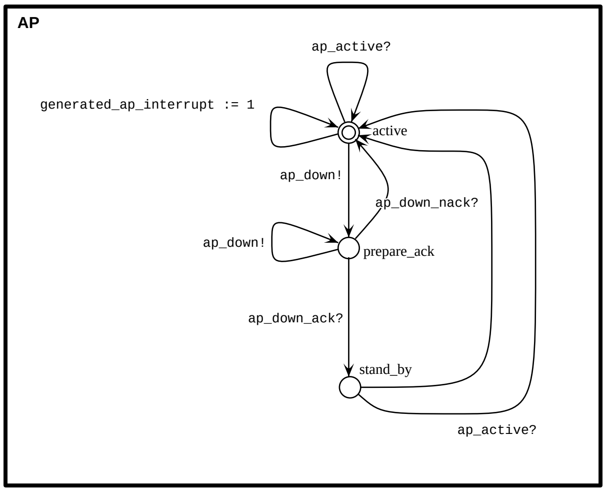

Figure 6.5: Q4a – The AP

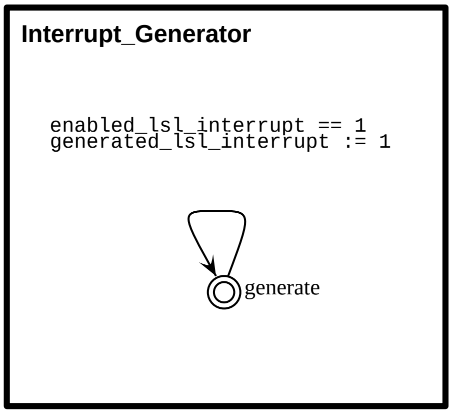

Figure 6.6: Q4a – The Interrupt Generator

6.6 Property 4b

6.6.1 Property

The shortest way from `down_expected` *to* `active` *does not take more than 1500 μs.*

This property is similar to property 4a described in the previous section, both what concerns formulation and solution. We want to observe traces, which pass through the node `down_expected`, goes "left" through `disable_lsl_interrupt` since `some_data == 1`, and then directly to `active`. This is what is meant by *shortest way.* The time consumed on this route between `down_expected` and `active` should not exceed 1500 μs.

Put differently, the property to verify is the following: if the IOP is in the node `down_expected`, and `some_data == 1`, then the IOP will be in the node `active` within 1500 μs.

6.6.2 Model Modifications

The same kind of modifications are done as for question 4a.

1. New declarations:

```
chan obs_begin;
urgent chan obs_end;
clock clk;
```

2. IOP changed: we choose the node `disable_lsl_interrupt` as starting point for the traces we want to investigate, instead of the node `down_expected`, since we are only interested in traces going through the former, see figure 6.7 (mid, left). In order to signal to an observer (see below) that the node `disable_lsl_interrupt` has been entered we add an edge leaving from, and going back to, this node with the label `obs_begin!`.

   Second, the edge from `enter_active` to `active` (mid) has been split up into two edges, inserting the new node `signal_obs`. This is done purely to add the new edge `obs_end!`, signalling the observer again, when entering the node `active`.

3. An observer automaton has been added, see figure 6.8. When the observer receives an `obs_begin?` signal from the IOP, it may start measuring the time through the variable `clk`. If more than 1500 μs passes before an `obs_end?` signal is received, the node `bad` is entered.

   Note the difference between this observer and the observer in figure 6.4 for property 4a. They should each serve the same purpose, but in fact, the one presented here is more correct since it will observe *any* `obs_begin?` signal from the IOP, while the one in figure 6.4 only will observe the first one.

4. The AP and the interrupt generator allow an unbounded number of interrupts, see figures 6.5 and 6.6 from property 4a's verification.

6.6.3 Property in UPPAAL Logic

```
A[] not Observer.bad
```

6.6.4 Result

The property is *not* satisfied. The error trace demonstrates a situation, where 24 AP interrupts are generated in between the nodes `down_expected` and `active`, each interrupt consuming time. The error trace is as follows:

1. The IOP gets a `ap_down` signal from the AP, and starts the closing down procedure.

2. The node `down_expected` is entered, and then the node `disable_lsl_interrupt`, where after the observer is signalled to start counting time. The IOP continues to node `enter_active`, consuming 1140 μs (3 + 37 + 1100).
3. Now 24 AP interrupts occur, each taking 15 μs (1 + 1 + 13), since `stand_by == 0`. This yields in total 360 μs.
4. At this point, 1500 μs have passed, and the observer automaton goes into the `bad` node.

The verification takes 7 minutes.

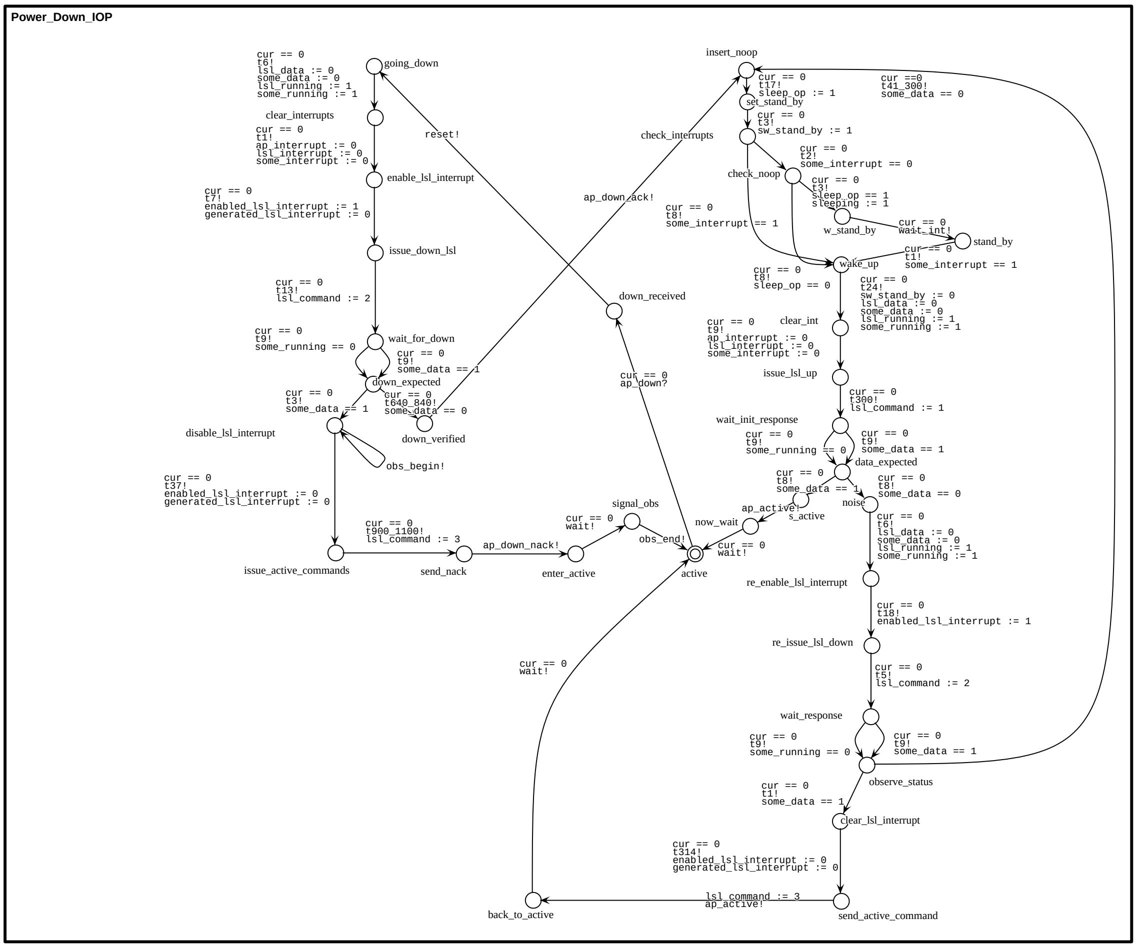

Figure 6.7: Q4b – The IOP

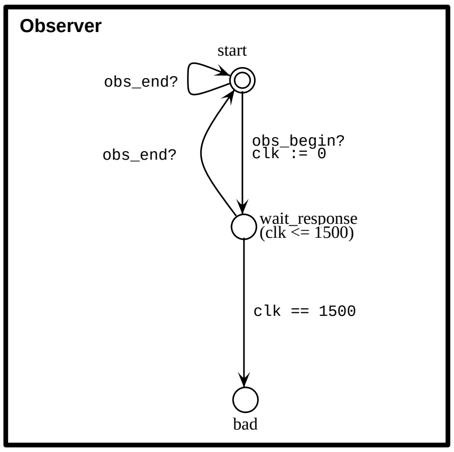

Figure 6.8: Q4b – The Observer

6.7 Property 5a

6.7.1 Property

The shortest way from `observe_status` *to* `wait_init_response` *does not take more than 1500 μs.*

This property is similar to properties 4a and 4b described in the previous sections, both what concerns formulation and solution. We want to observe traces, which pass through the node `observe_status`, goes "right and up" through `insert_noop` (up, right) since `some_data == 0`, and then down to `wait_init_response`. Note that an interrupt has to have occurred in order for the IOP to get past the `stand_by` node. This is what is meant by *shortest way*. The time consumed on this route between `observe_status` and `wait_init_response` should not exceed 1500 μs.

Put differently, the property to verify is the following: if the IOP is in the node `observe_status`, and `some_data == 0` and `some_interrupt == 1`, then the IOP will be in the node `wait_init_response` within 1500 μs.

6.7.2 Model Modifications

The same kind of modifications are done as for questions 4a and 4b.

1. New declarations:

   ```
   chan obs_begin;
   urgent chan obs_end;
   clock clk;
   ```

2. IOP changed: In order to signal to an observer (see below) that the node `observe_status` has been entered while an interrupt has occurred, see figure 6.9 (down, right), we add an edge leaving from, and going back to, this node with the label:

   ```
   some_interrupt == 1
   some_data == 0
   obs_begin!
   ```

   Note that we also require that `some_data == 0` since this will guarantee that *the shortest way* is chosen.

   Second, the edge from `issue_lsl_up` to `wait_init_response` (mid, right) has been split up into two edges, inserting the new node `signal_obs`. This is done purely to add the new edge `obs_end!`, signalling the observer again, when entering the node `wait_init_response`.

3. An observer automaton has been added, see figure 6.10. When the observer receives an `obs_begin?` signal from the IOP, it may start measuring the time through the variable `clk`. If more than 1500 μs passes before an `obs_end?` signal is received, the node `bad` is entered.

6.7.3 Property in UPPAAL Logic

```
A[] not Observer.bad
```

6.7.4 Result

The property is satisfied with a limited number of interrupts (1 AP interrupt and 2 LSL interrupts).

A verification with an unlimited number of interrupts were tried, but this had not terminated after 12 hours. An attempt with 15 AP interrupts and 1 LSL interrupt had not terminated after 5 hours. An attempt with 10 AP interrupts and 0 LSL interrupts had not terminated after 2 hours.

It is quite likely, that an error trace similar to those for properties 4a and 4b can be generated if there is no bound on the number of AP interrupts that can occur.

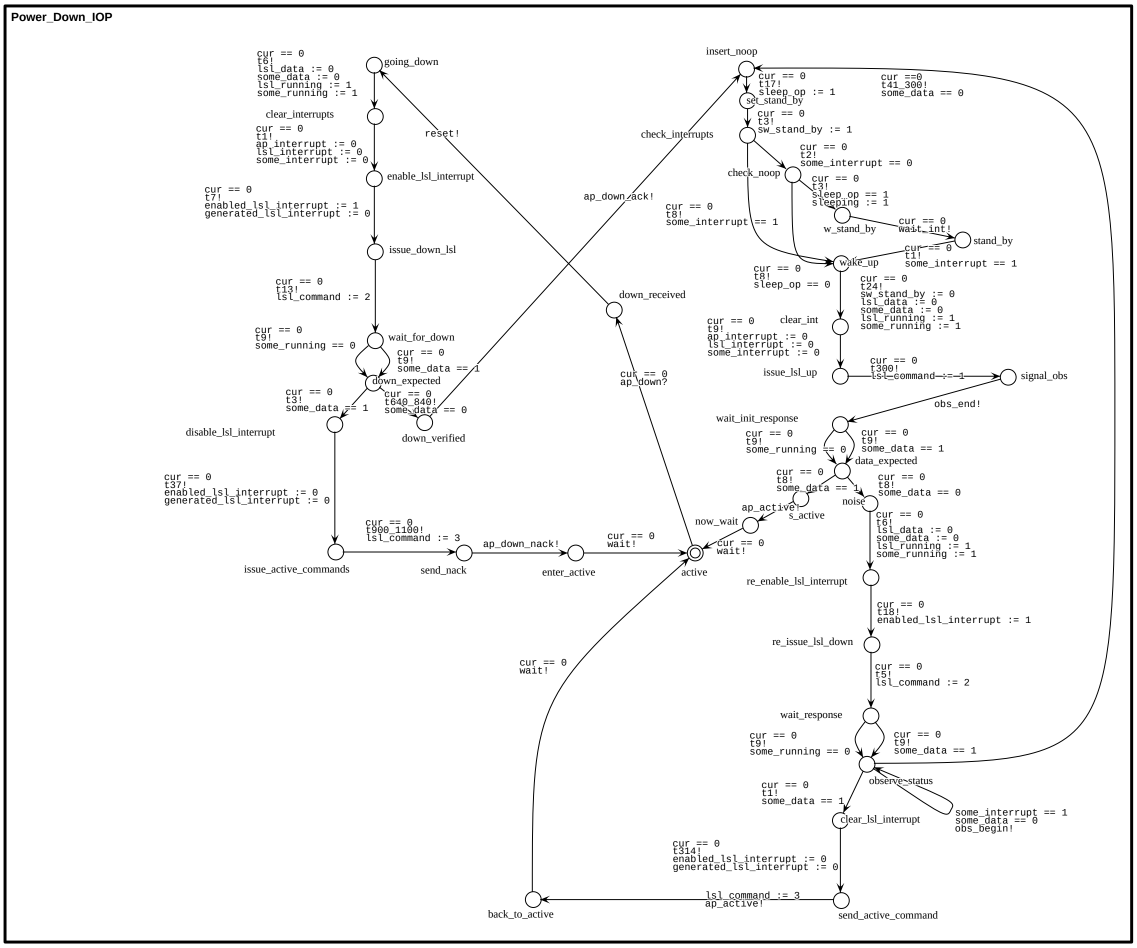

Figure 6.9: Q5a – The IOP

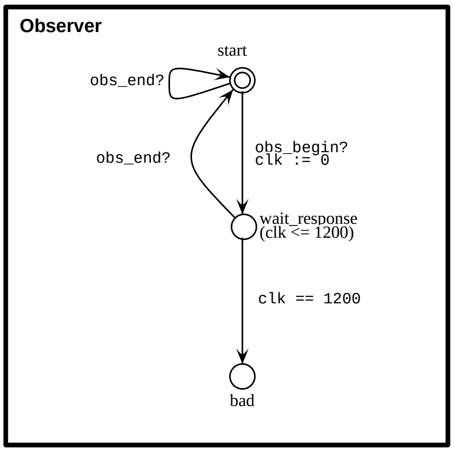

Figure 6.10: Q5a – The Observer

6.8 Property 5b

6.8.1 Property

The shortest way from `observe_status` *to* `active` *does not take more than 1500 μs.*

This property is similar to properties 4a, 4b and 5a described in the previous sections, both what concerns formulation and solution.
We want to observe traces, which pass through the node `observe_status`, goes "left and down" through `clear_lsl_interrupt` (down, right) since `some_data == 1`, and then up to `active`. This is what is meant by *shortest way*. The time consumed on this route between `observe_status` and `active` should not exceed 1500 μs.

Put differently, the property to verify is the following: if the IOP is in the node `observe_status`, and `some_data == 1` then the IOP will be in the node `active` within 1500 μs.

6.8.2 Model Modifications

The same kind of modifications are done as for questions 4a, 4b and 5a.

1. New declarations:

```
chan obs_begin;
urgent chan obs_end;
clock clk;
```

2. IOP changed: In order to signal to an observer (see below) that the node `observe_status` has been entered, see figure 6.11 (down, right), we add an edge leaving from, and going back to, this node with the label:

```
some_data == 1
obs_begin!
```

   Note that we also require that `some_data == 1` since this will guarantee that *the shortest way* is chosen.

   Second, the edge from `back_to_active` to `active` (mid) has been split up into two edges, inserting the new node `signal_obs`. This is done purely to add the new edge `obs_end!`, signalling the observer again, when entering the node `active`.

3. An observer automaton has been added, see figure 6.12. When the observer receives an `obs_begin?` signal from the IOP, it may start measuring the time through the variable `clk`. If more than 1500 μs passes before an `obs_end?` signal is received, the node `bad` is entered.

4. The AP and the interrupt generator allow an unbounded number of interrupts, see figures 6.5 and 6.6 from property 4a's verification.

6.8.3 Property in Uppaal Logic

```
A[] not Observer.bad
```

6.8.4 Result

The property is *not* satisfied. The error trace demonstrates a situation, where 79 AP interrupts are generated in between the nodes `observe_status` and `active`, each interrupt consuming time. The error trace is as follows:

1. The IOP gets an `ap_down` signal from the AP, and starts the closing down procedure.
2. The node `observe_status` is entered, where after the observer is signalled to start counting time. The IOP continues to node `clear_lsl_interrupt`, consuming 1 μs.
3. An LSL interrupt occurs, consuming 14 μs (1 + 13).
4. The IOP continues to `back_to_active`, consuming 314 μs.
5. Now 79 AP interrupts occur, each taking 15 μs (1 + 1 + 13), since `stand_by == 0`. This yields in total 1185 μs.
6. At this point, 1500 μs have passed, and the observer automaton goes into the `bad` node.

The verification takes 2hours and 40 minutes.

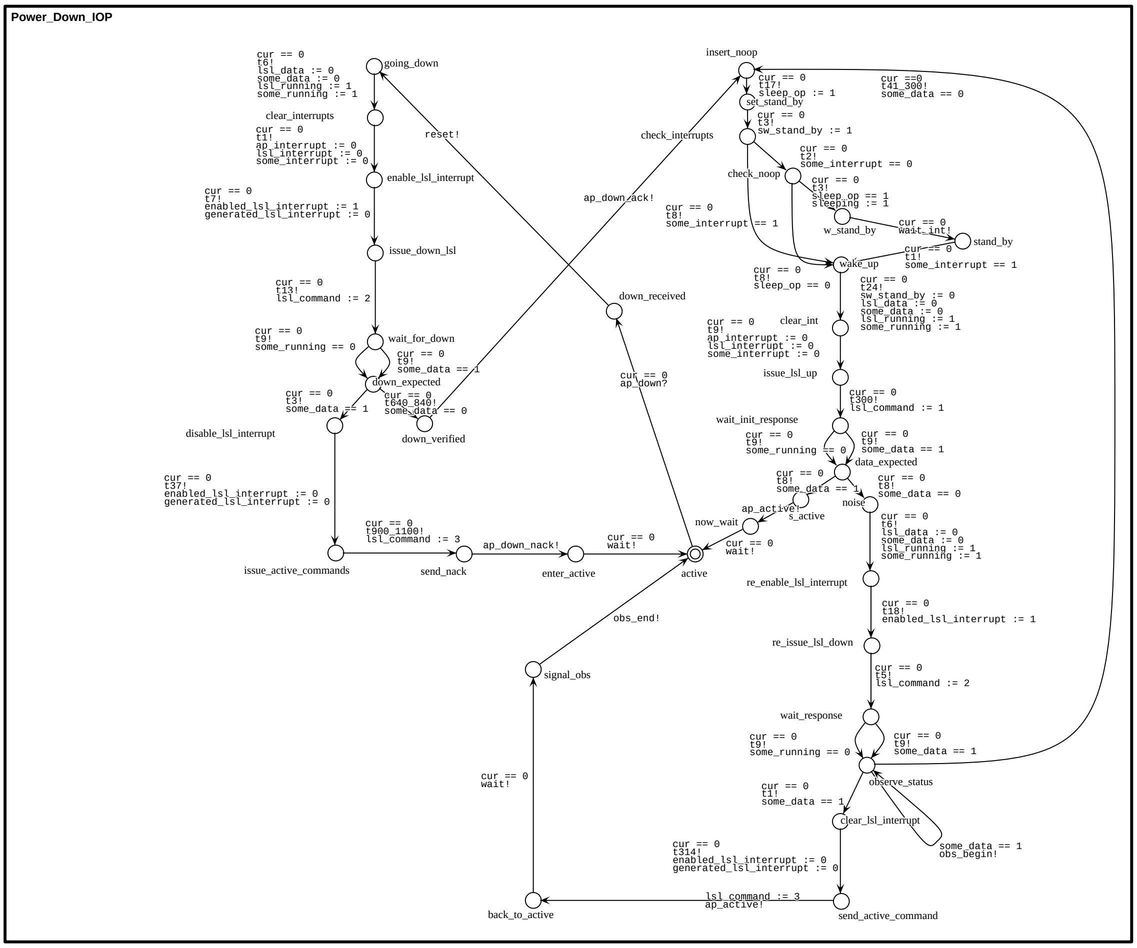

Figure 6.11: Q5b – The IOP

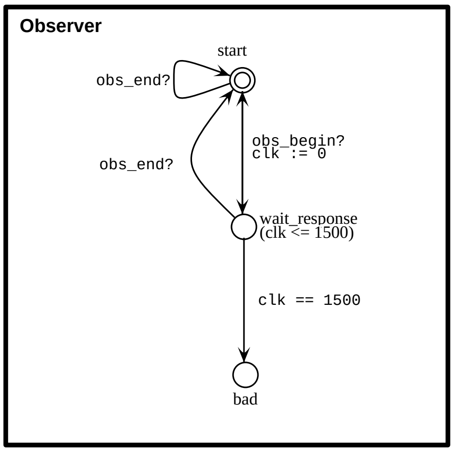

Figure 6.12: Q5b – The Observer

6.9 Property 6

6.9.1 Property

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.*

6.9.2 Model Modifications

1. New declarations:

```
int old_lsl_command;
```

2. IOP changed: The variable `old_lsl_command` has been assigned to in each of those 5 edges which assign a value different from 0 to `lsl_command` (`old_lsl_command` gets the same value). In this way, `old_lsl_command` will always hold the last value different from 0 assigned to `lsl_command`. See figure 6.13.

6.9.3 Property in Uppaal Logic

```
A[] (Power_Down_IOP.check_noop and cur == 0 and
     sleep_op == 1 and sleeping == 0)
       imply
         old_lsl_cmd == 2
```

The condition of the implication is the guard (except for `sleeping == 0`) of the only edge in the system which assigns 1 to `sleeping`, namely the edge from `check_noop` to `w_stand_by` in the IOP, see figure 5.2 (up, right).

6.9.4 Result

The property is satisfied.

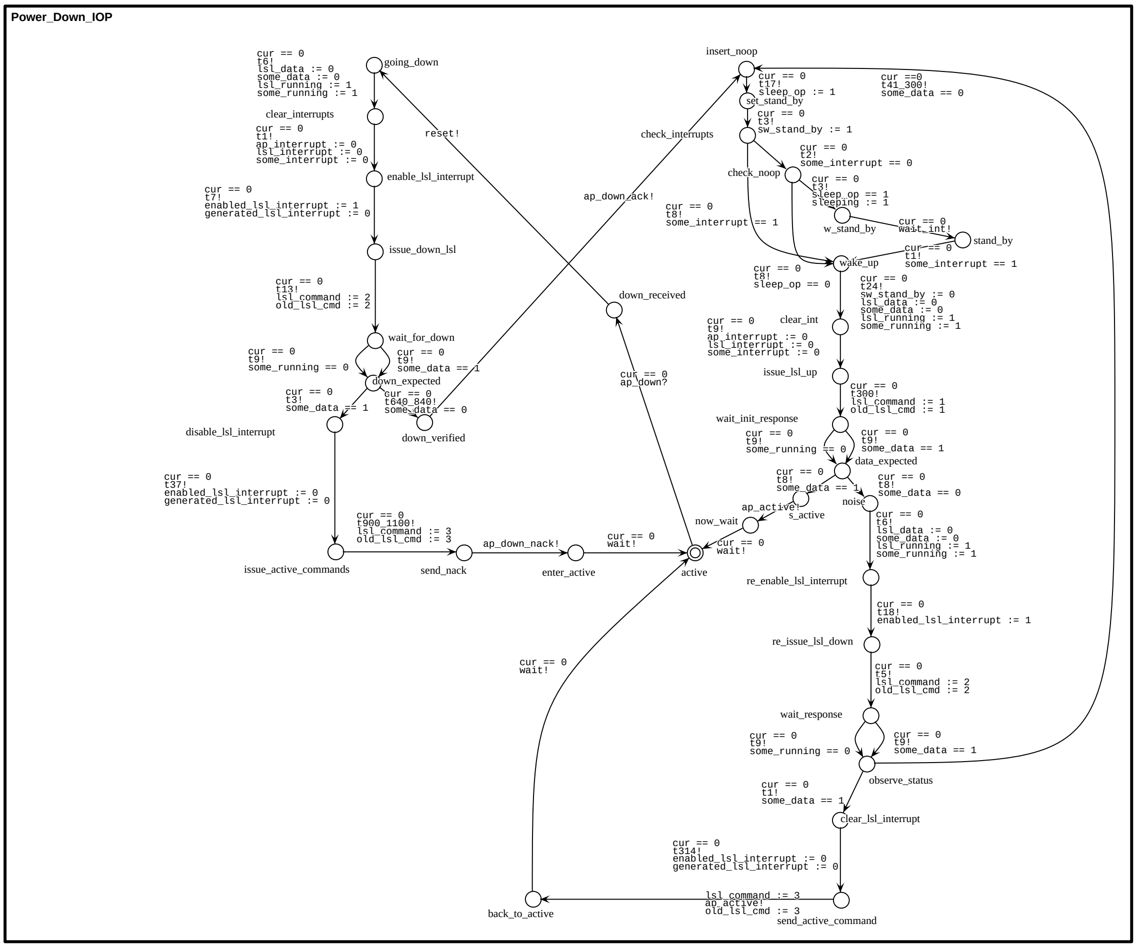

Figure 6.13: Q6 – The IOP

6.10 Property 7

6.10.1 Property

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.*

6.10.2 Model Modifications

1. New declaration:

```
int old_lsl_command;
```

2. IOP changed: The variable `old_lsl_command` has been assigned to in each of those 5 edges which assign a value different from 0 to `lsl_command` (`old_lsl_command` gets the same value). In this way, `old_lsl_command` will always hold the last value different from 0 assigned to `lsl_command`. See figure 6.14.

   Furthermore, consider those nodes from which there is an edge that assigns 1 or 3 to `lsl_command`. These are the nodes `issue_active_commands` (down, left), `issue_lsl_up` (mid, right) and `send_active_command` (down, right). If the property shall hold, then a pre-condition for each of these edges is that the last value of `lsl_command` is not in conflict with the new value (1 is in conflict with 3, and vice versa). Hence, for each of these nodes, an error node is introduced and an edge leading to it in case the pre-condition is violated. That is, the new nodes are: `bad_start_cmd1`, `bad_start_cmd2`, and `bad_init_cmd`.

   For example, take the node `issue_active_commands` (down, left) from which an edge assigns 3 to `lsl_command`. A new edge then leads to the new error node `bad_start_cmd1` in case the last value assigned was 1 (`old_lsl_command == 1`).

6.10.3 Property in Uppaal Logic

```
A[] not Power_Down_IOP.bad_start_cmd1

A[] not Power_Down_IOP.bad_start_cmd2

A[] not Power_Down_IOP.bad_init_cmd
```

6.10.4 Result

The property is satisfied.

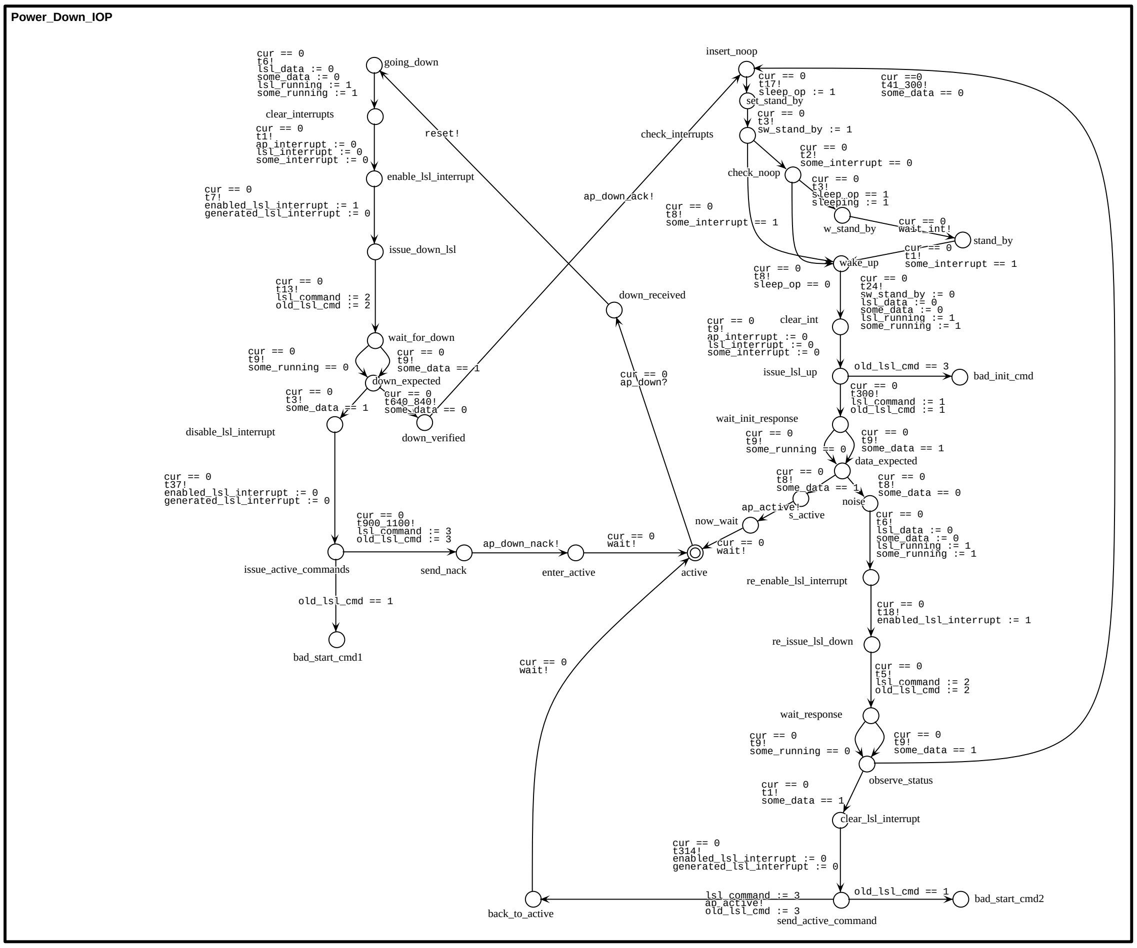

Figure 6.14: Q7 – The IOP

6.11 Property 8

6.11.1 Property

No more than 1500 μs must pass from an interrupt occurs until all drivers are active.

6.11.2 Model Modifications

1. New declarations:

```
clock clk;
chan lsl_interrupt_set,
     lsl_driver_on,
     lsl_driver_off;
urgent chan lsl_driver_is_on;
```

2. LSL interrupt handler changed: In order to signal to an observer (see below) that an LSL interrupt has occurred, the communication `lsl_interrupt_set!` has been added, together with the node `signal_obs`, see figure 6.15.

3. LSL driver changed: In order to signal to a driver status observer (see below) when the driver becomes active and when it becomes inactive, the communications `lsl_driver_on!` and `lsl_driver_off!` have been added on relevant edges, see figure 6.16. Note how the value of `lsl_command` determines whether the driver is turned off or not when leaving the driver, and that for the same reason the resetting `lsl_command := 0` has been moved to the end.

4. An LSL driver observer has been added, see figure 6.17. This automaton continuously shows whether the driver is active (node `on`) or inactive (node `off`) by responding to the signals `lsl_driver_on!` and `lsl_driver_off!` from the driver.

   In addition, a communication to an observer (see below) is always possible on the urgent channel `lsl_driver_is_on` when in node `on`.

5. An observer automaton has been added, see figure 6.18. Upon receiving an `lsl_interrupt_set?` signal from the LSL interrupt handler, it may start measuring the time through the variable `clk`. If more than 1500 μs passes before an `lsl_driver_is_on?` signal from the LSL driver status observer is received, the node `bad` is entered. The extra `lsl_interrupt_set?` edge leaving and going back to the `start` node allows the observer to measure *any* interrupt, and not just the first one. The extra edge on the node `good` ensures that the observer does not block the model it observes.

6. The interrupt generator only allows 1 interrupt, see figure 6.19.

7. IOP changed: It is necessary to add the guard `lsl_command == 0` on edges following an assignment of the form `lsl_command := 3`, in order to enforce the corresponding *activate driver* edge in the LSL driver to be taken. It concerns the edges starting in `send_nack` (down, left) and `back_to_active` (down, right), see figure 6.20. This is really a modification that should be added to all models, although it has no importance for the other verifications since activating the driver in those is just an identity edge (from the LSL driver node `stand_by` and back to that node).

6.11.3 Property in Uppaal Logic

```
A[] not Observer.bad
```

6.11.4 Result

The property is satisfied in a model restricted to allow 1 AP interrupt and 1 LSL interrupt.

A verification has been tried with 1 AP interrupt and 2 LSL interrupts, but that took a hour and 140 MB without any termination.

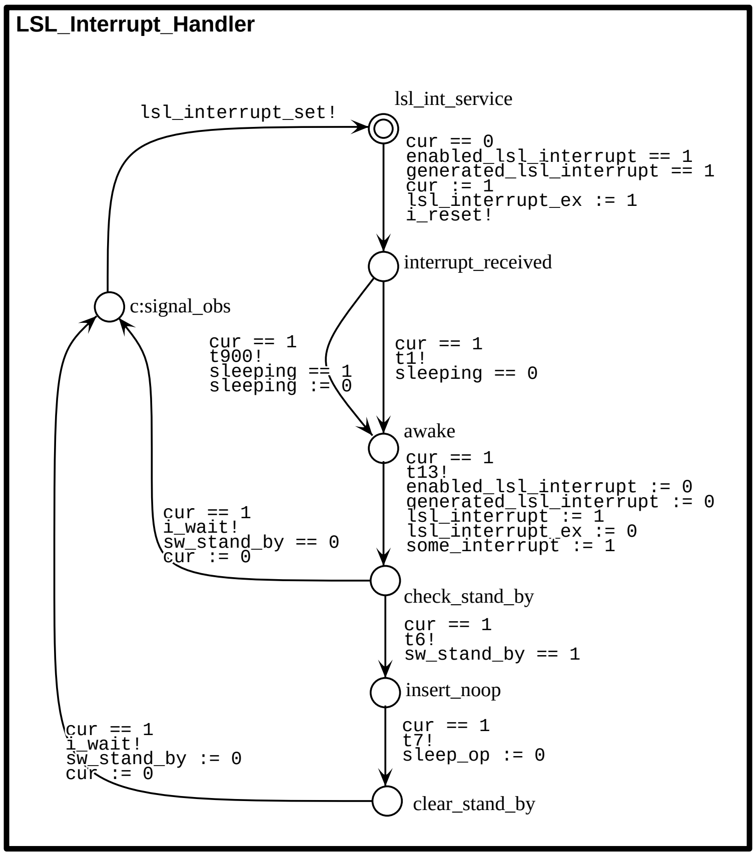

Figure 6.15: Q8 – The LSL Interrupt Handler

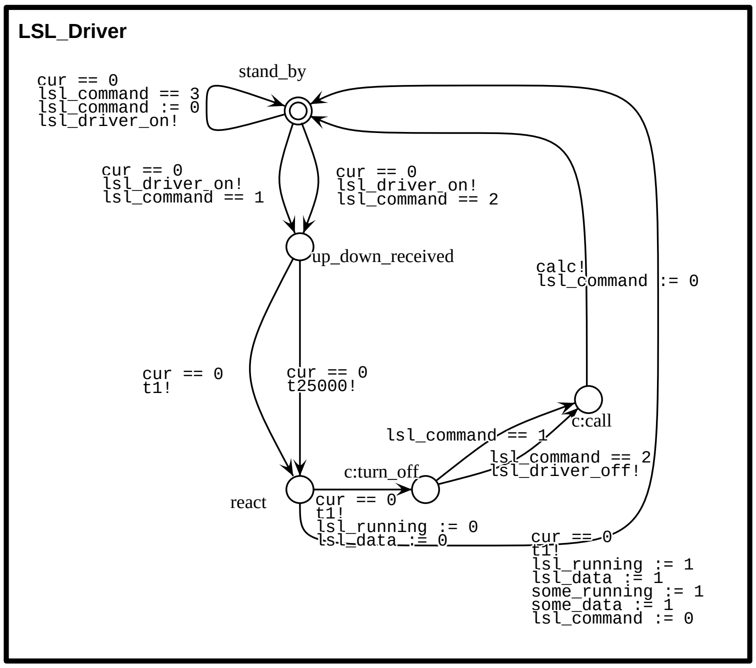

Figure 6.16: Q8 – The LSL Driver

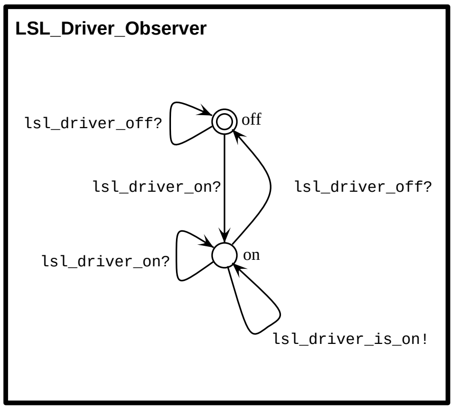

Figure 6.17: Q8 – The LSL Driver Observer

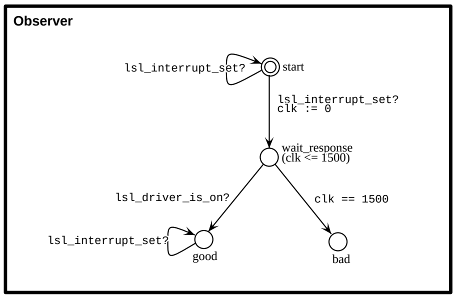

Figure 6.18: Q8 – The Observer

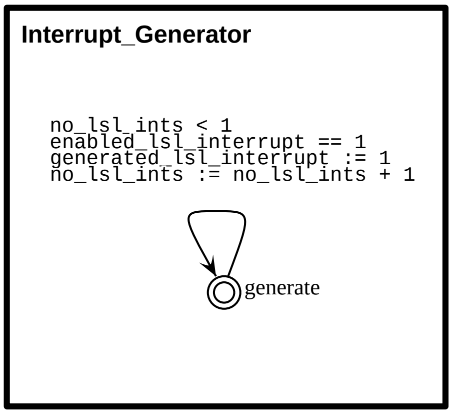

Figure 6.19: Q8 – The Interrupt Generator

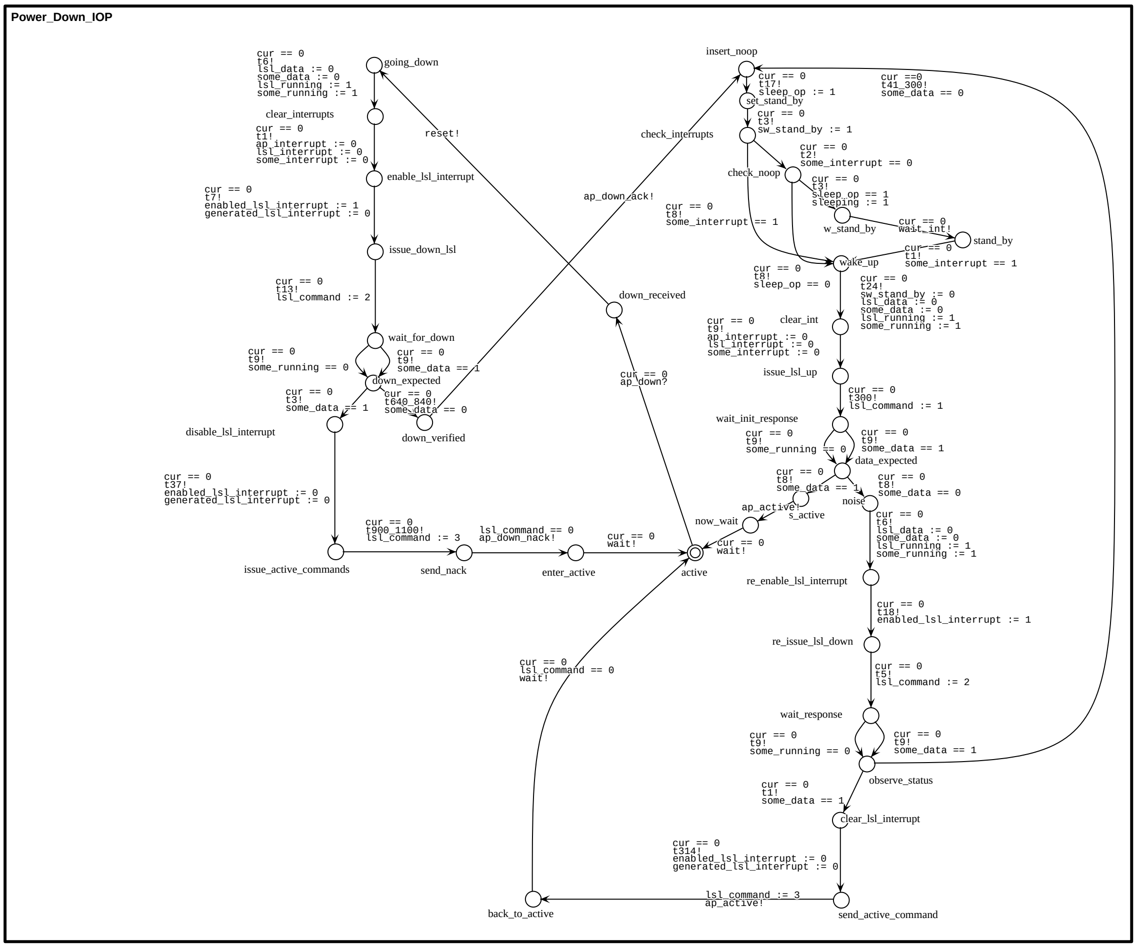

Figure 6.20: Q8 – The IOP

6.12 Property 9

6.12.1 Property

1. *It must be possible for both interrupt handlers to be in the node* `insert_noop` *at the same time, while in addition* `sleep_op` *is 0.*
2. *It must be possible for both interrupt handlers to be in the node* `insert_noop` *at the same time, while in addition* `sleep_op` *is 1.*
3. *If both interrupt handlers are in the node* `insert_noop` *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`*.*

6.12.2 Model Modifications

None.

6.12.3 Property in Uppaal Logic

```
E<> AP_Interrupt_Handler.insert_noop and
    LSL_Interrupt_Handler.insert_noop and
    sleep_op == 0

E<> AP_Interrupt_Handler.insert_noop and
    LSL_Interrupt_Handler.insert_noop and
    sleep_op == 1

A[] (AP_Interrupt_Handler.insert_noop and
     LSL_Interrupt_Handler.insert_noop)
       imply
         (Power_Down_IOP.set_stand_by or
          Power_Down_IOP.check_interrupts or
          Power_Down_IOP.check_noop or
          Power_Down_IOP.w_stand_by or
          Power_Down_IOP.stand_by or
          Power_Down_IOP.wake_up)
```

6.12.4 Result

1. Not satisfied.
2. Satisfied.
3. Satisfied.

It turns out that property 1 is wrongly formulated, and probably *should not* be satisfied, just as shown by the verification.

6.13 Property 10

6.13.1 Property

It must be possible to come from the node `noise` *to the node* `stand_by`.

6.13.2 Model Modifications

1. New declarations:

```
int noise_reached;
```

2. IOP changed: the assignment `noise_reached := 1` has been added to the edge from node `data_expected` to node `noise`, see figure 6.21 (mid, right).

6.13.3 Property in UPPAAL Logic

```
E<> Power_Down_IOP.stand_by and noise_reached == 1
```

6.13.4 Result

The property is satisfied.

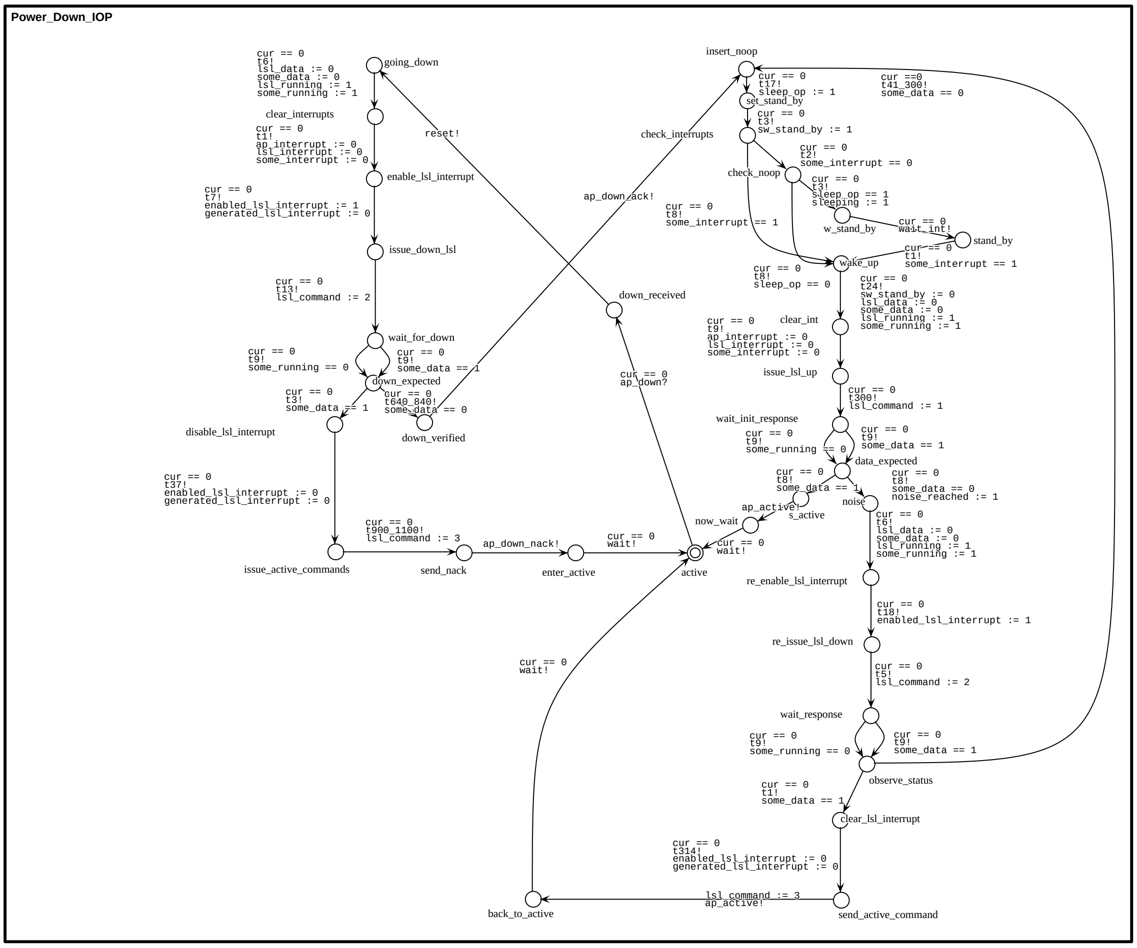

Figure 6.21: Q10 – The IOP

6.14 Property 11

6.14.1 Property

I should not be possible to come from the node `stand_by` *to the node* `active` *without synchronizing on the channel* `ap_active`*.*

6.14.2 Result

The property is trivially satisfied by observing the transition system. However, this only became clear for B&O after having built the model. It was decided not to verify the property due to its obvious truth.

Chapter 7

Evaluation

This chapter contains an evaluation of the project, as seen from each of the two partners: B&O and AUC.

7.1 B&O's Evaluation

This section contains B&O's comments on the project. It is mostly a direct translation of an email in Danish received from Johnny Kudahl. In certain places, we have added explanatory text [*in squared brackets*].

7.1.1 General Comments

Method and Language Syntax

The method and the language syntax is easy to work with, and it is relatively easy to formulate ones design/code in UPPAAL. In UPPAAL everything is executed in parallel and with the same priority. The model is perhaps missing some features for serial evaluation with different priorities. In general I find that UPPAAL as a method is "strong" and very usable. We have a big interest in following the future development of UPPAAL.

Model Building

Where I gained most from this work was during the building of the the model. During this phase, you are forced to consider many situations, that you would normally not otherwise have considered. During the development of the model, I found 3 errors, a fatal one, and two minor. The fatal error was that I did not reinitialize the drivers after a power down request to a driver that was processing data. The two minor errors were that I had wrongly swapped two return values in a function. These errors would definitely also have been found during the verification.

The Typing Phase

Because this phase was mainly done by you [*AUC*], I don't have a big experience with the tool. I, however, got the impression that the user interface was a bit heavy to work with. Another thing I noticed was, that there was no version control in the tool. It would be nice, if one could lock versions. Also, one has to make a new copy of the main model, for each property verified [*in order modify it, for example by adding an observer or extra variables*]. That makes it difficult to keep track of the models, and to maintain them when changes are made to the main model.

If one looked to the object oriented world, and regarded the main model as the *main class* instead, then all other models could then inherit from this main class, adding and removing the facets necessary to formulate the property. Changes in the main model will then affect all specializations of it. It should though be possible to "freeze" certain areas in the sub models, such that these areas are not changed by changes in the main model.

It would also be an advantage, if one could divide a system into subsystems. It would increase comprehensibility, and it would be possible to focus on part of a system or on the whole system at a higher level.

The Verification Phase

This is UPPAAL's other strength. After the model has been formulated it is relatively easy to add or change properties to be verified. That some properties take time to verify is not so important, as long as one gets an answer.

7.1.2 The Individual Properties

I have tried to prioritize the properties:

Very Important (affecting software and hardware). Such properties are 4a, 4b, 5a, 5b and 8. These properties are important since they give a clear indication of how much noise the module can stand without data are lost on the low speed link (said in other words: how many interrupts can the system tolerate without exceeding the famous 1500 μs limit).

This analysis gave the result, that in the worst case there could come 1 LSL interrupt and 18 AP interrupts [*this is the smallest number of interrupts leading to the 1500 μs limit being crossed*]. This gives a max AP interrupt frequency of 1500 μs divided by 18 which is 83 μs (12 KHz). It is now my task to maintain this frequency.

Properties 1, 3 and 11 also belong to this group. The difference between these and the previous is, that they more or less were answered during the development of the model. Before we started developing the model, it was not possible to see whether these properties were satisfied.

Important for the design of power down (software only). These are properties 6 and 7. If the properties were not satisfied, it would require a re-design of either the power down [the IOP] or the driver. The consequences would, however, only be software related – the hardware would still work.

Very Important (software and hardware) but "known" to hold. These are properties 2a and 9 (9.1 is, however, wrongly formulated). If these properties were not satisfied, the whole idea behind the power down design would be wrong. These properties have therefore been highly considered during the design of the power down. One can say that the properties were given to UPPAAL well knowing that they would be satisfied.

Ups, properties being wrongly formulated. These are properties 2b, 9.1 and 10. Concerning properties 2b and 9.1, I had expected another outcome than the verification gave. Concerning property 10, I have not yet been able to figure out what I was thinking when formulating it.

7.2 AUC's Evaluation

7.2.1 UPPAAL as a Communication Medium

The first session we had with B&O started with an informal discussion of the model, where basically Johnny Kudahl explained how it worked, and we asked questions. Within an hour we (AUC) felt a need to get a deeper understanding, and at that point it was suggested to start writing down parts of the model in UPPAAL, directly on the white board. Johnny Kudahl was instructed in UPPAAL's notation, it took 15 minutes, and thereafter we were basically all communicating in the same language. It appeared unproblematic to make Johnny Kudahl work with the notation, and as a communication medium between AUC and B&O it appeared excellent.

7.2.2 Verification Results

The protocol appeared to satisfy all the properties stated (at least those that were correctly formulated), and our feeling was, that the protocol was very securely designed. One of the potentially un-secure points was the consequences of too many interrupts during powering up and down. Some of the questions were in fact formulated to examine this. The result was a couple of constants suggested by UPPAAL, indicating the maximal number of interrupts allowed within certain intervals. B&O will now try to obey these constants. In other cases, properties were rejected, and then discovered to be wrongly formulated. Hence, in these cases, no bug was discovered, but an unexpected – and correct – behaviour of the system was observed. Such results help to understand the working of the protocol.

In general, we felt that formulating the model and verifying the properties increased B&O's confidence in the design. The only bugs identified were caught by B&O during the modeling. These errors would, however, have been caught during verification also, had they not been corrected beforehand.

7.2.3 The UPPAAL Language

As part of the result of this work, we have suggested ways of modeling timed transitions and interrupts. The modeling is relatively simple, and models are written in a style pretty close to the way one would probably write them, if UPPAAL had been designed to support these constructs. In fact, interrupts are easy to model in UPPAAL (using committed nodes for example); but it is the combination of timed transitions, time slicing and interrupts, that makes the need for the `cur` variable. Hence, it is really the combination of timed transitions and time slicing that causes the extra modeling. We do not know of any other model checker that provides these facilities as we needed them in this example.

7.2.4 The UPPAAL Environment

UPPAAL needs a version control system for deriving sub models from a full model. Sub models are derived for two reasons:

1. *abstraction:* when reducing a model that is too big for verification to a model that is small enough.
2. *verification:* when modifying a model in order to formulate a property to be verified, perhaps by adding an observer.

In fact, we worked with 3 models due to abstraction and 12 models due to verification. In parallel with the present work, a tool has been designed which automatically generates test-automata from a richer UPPAAL logic, hence, in the future it is expected that sub model derivation will mostly be caused by abstraction only. However, even in this case, version control seems very important, and in addition, a tool which supports defining the abstractions themselves seems quite useful. That is, basically a tool that allows to "throw out" code, restrict loops by putting an upper bound on the number of iterations allowed, etc., in an easy way.

Chapter 8

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 AUC and B&O, where the B&O representative visited AUC. During this week, a first sketch of the model was written down in UPPAAL's language. The model was based on examination of the C-code implementation. The work carried out during the following two weeks was mainly carried out by AUC.

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.

The UPPAAL notation appeared to be a good communication medium between AUC and B&O. 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 will be of help to B&O in their design process. Various unexpected, but correct, behaviours 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.

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. More generally, transitions that take time seems to be a useful concept. We also have frequent discussions as to what a model checking language should look like in order to be convenient to work with. The results of these discussions will be documented elsewhere.

Bibliography

[1] 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.

[2] 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.

[3] Johan Bengtsson, David Griffioen, Kåre Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang 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.

[4] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang 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.

[5] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang 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.

[6] A. Bouali, A. Ressouche, and V. Roy R. de Simone. The FC2Toolset. *Lecture Notes in Computer Science*, 1102, 1996.

[7] P.R. D'Arenio, 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.

[8] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. *In Proc. of* 7th International Conference on Formal Description Techniques, 1994.

[9] 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.

[10] 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*, Dec 1997. To appear.

[11] Pei-Hsin Ho and Howard 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.

[12] Gerard Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[13] 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.

[14] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller: an Industrial Case Study using UPPAAL. In preparation., 1997.

[15] R. Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, 1989.