Experiences with Modelling the USB Hub Protocol Using Spin*

Dennis Dams, Rob Gerth
Dept. of Math. & Comp. Science
Eindhoven Univ. of Technology
PO Box 513
5600 MB Eindhoven
{wsindd,robg}@win.tue.nl

Loe Feijs
Philips Research Labs. &
Eindhoven Univ. of Technology
Prof. Holstlaan 4
5656 AA Eindhoven
feijs@natlab.research.philips.com

Gertjan Kamsteeg
Dept. of Comp. Science
Leiden Univ.
PO Box 9512
2300 RA Leiden

Abstract

This paper describes our ongoing experiences during the a-posteriori formalisation of the USB hub protocol in Promela, and subsequent simulation and verification experiments. Such an activity forces one to consider all chosen aspects with equal (and great) precision. The paper focusses on the technical aspects of the model, the choices we had to make, and the problems we encountered. Many unclarities, inconsistencies and errors in the original text of the specification were uncovered.

1 Introduction and Motivation

The USB hub protocol is a recently developed protocol which is used for building up networks consisting of one computer and a number of peripheral devices. In this paper we report on our experiences during the a-posteriori formalisation of the USB hub protocol in Promela, and subsequent simulation and verification experiments. Our interests that motivated the research conducted in the present paper are threefold:

- to improve our understanding of the USB hub protocol,
- to find out in how far a state-of-the-art model checker like Spin [4, 5] would be capable of dealing with a complex real-life protocol,
- to experiment with modeling principles that would allow for a modular separation of concerns such that (ideally) the Spin model would be a faithful model of the electrical, logical, and timing-related aspects of the protocol, while still allowing efficient simulation and verification.

The USB hub protocol is part of an emerging industry standard called Universal Serial Bus (USB), which is meant as a cost-effective solution for the problem of connecting a variety of low- and medium-speed peripheral devices to personal computers. The USB allows for hot plug-and-play of devices, which means that the user can at any time add or remove a device from the USB network, without disturbing the operation of the rest of the network. Moreover USB allows the PC to put one or more devices into a state of rest (typically for reasons of power consumption), and still maintain the possibility that later the PC or the device itself wakes up and reactivates the pertinent part of the network. The USB protocol is particularly suited for multimedia applications because the network will carry data of different kinds in a time-division multiplex manner. In particular, it will carry interrupt messages with a guaranteed latency time, it will carry stream-like data such as audio-streams and (restricted-bandwidth) video streams with a guaranteed throughput, and it will also carry bulk-like computer data such as texts to be sent to a printer. The requirement to USB of being cost-effective, also implies that the protocol must be robust for errors caused by malfunctions due to hardware faults in (possibly cheap) peripheral devices.

All these requirements have contributed significantly to the complexity of the USB protocols. Although the interconnecting wires and plugs are very simple and can be produced cheaply, the protocols are very complex. In particular the hub protocol (the hub is the network component which performs splitter and repeater functions) has become very complex. The hub plays a key role in the automatic configuration and re-configuration of the network (the hot plug-and-play), in the detection and isolation of malfunctioning peripheral devices, and in the suspend/resume behaviour of the bus.

Throughout our case study we use the protocol design language PROMELA, as proposed by Holzmann, together with the model checker tool Spin. The language Promela has a C-like syntax with a limited number of concrete data types, conventional assignments, Dijkstra's guarded commands and CSP-like send and receive statements. Promela models can

*This research was supported by the Netherlands Computer Science Research Foundation (SION).

be executed directly (simulation), or can be analysed by means of model-checking, that is, the entire state space is searched for violations of certain temporal properties. The temporal properties can be expressed by so-called never-claims, which roughly speaking represent formulae of linear-time temporal logic.

1.1 Architectural aspects

In Figure 1 we show a typical USB configuration with one PC, a keyboard, a mouse, and two loudspeakers. There are two hubs, and as can be seen, one

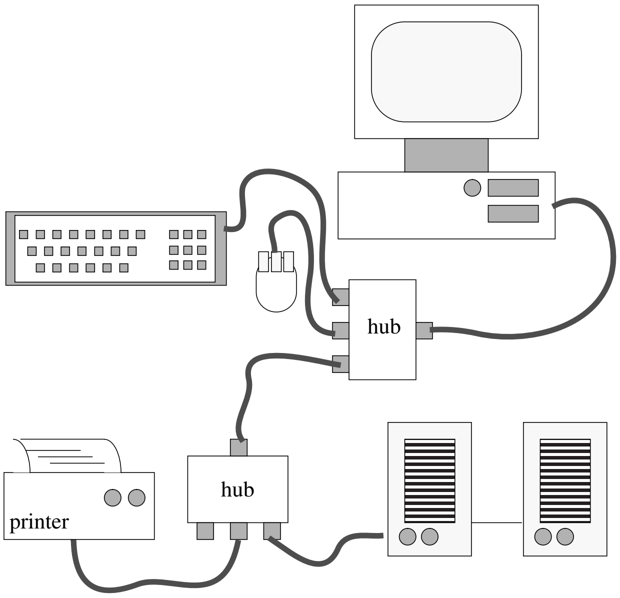

Figure 1: USB Network with two hubs

hub is configured as a subnode of another hub, and in this way the whole network has a tree structure. In general, the tree may have a width and breadth of up to five. In practice, one or more of the hubs are built-in to one of the peripheral devices[1], e.g. the keyboard, rather than being separate devices. But from a logical point of view that makes no difference, and in order to avoid confusion we leave it as in Figure 1.

1.2 Informal description of the protocol

The protocol is a kind of master-slave protocol in which the PC is the master. It is called the *host* and plays a special role. It assigns time slots to other devices, and it makes sure that the entire bus operates in a certain rhythm based on so-called *frames*.

[1] We follow the terminology of [3] in that the term *device* is used for any node in the USB tree except the host. Thus, also a hub is a device. A device that is not a hub is also called a *function*.

We ought to explain two important USB concepts. A *packet* is a short sequence of consecutive bits that can be transmitted over the USB. Packet length may vary from 24 bits up to 1026 bytes. A packet has a clearly indicated beginning, characterised by a unique synchronisation sequence, and a clearly indicated end, characterised by a certain state of the bus, already recognisable at the electrical level. We use the abbreviations SOP for 'start of packet' and EOP for 'end of packet'.

A *frame* is a period of time of one millisecond. Within one frame there is enough time for hundreds of packets (the bus rate is 12.000.000 bits per second). The notion of frame is important for the adjustment of local clocks and for detection of malfunctioning devices. The idea is that the PC signals the beginning of a new frame by means of a special packet, started by a different synchronisation sequence, called SOF (start of frame). When one millisecond has almost passed, which the devices are supposed to know by having counted almost 12.000 clock ticks on their local clock, the bus is supposed to be silent. If it is not, some device has failed to abort its packet in due time, and such a device is said to be *babbling*. A device which persists in babbling, will eventually be disconnected from the bus by its "parent" hub. The same happens to a device that starts a packet and then silences until the end of the frame — this is called *loss of activity (LOA)*. The hub's state diagram is given

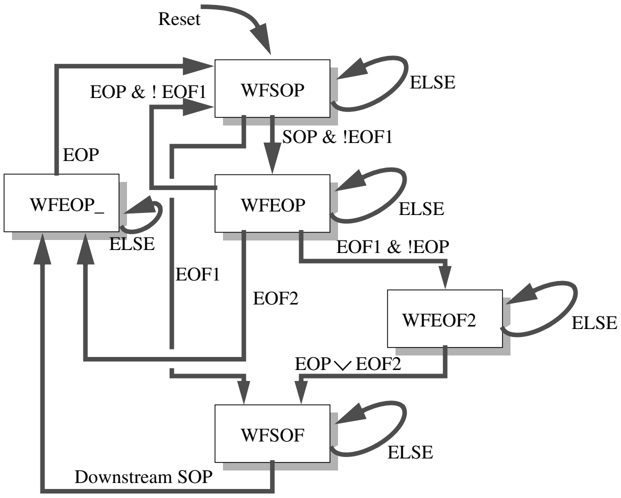

Figure 2: Repeater state machine (as used in first model)

in Figure 2, which is obtained as a first step of debugging and adapting the diagram presented in [3]. This state diagram is concerned with the repeater function of the hub, and during its typical operation, the hub cycles through this diagram precisely once for each packet transferred by the hub. The four main states are named as follows. WFSOP means 'wait for start of packet', WFEOP means 'wait for end of packet', WFEOF2 means 'wait for EOF2 point', and finally WFSOF means 'wait for start of frame'.

When we started with our initial modeling attempts, taking place in August 1996, we started from a slightly different diagram, as provided in the USB Revision 1.0 specification [3] (Fig. 11-6). While building our Promela model, we found that the diagram from USB Revision 1.0 specification could impossibly be right and we had to devise an adapted diagram ourselves.

The state-diagram in [3] has a direct transition from the state WFSOF to to the state WFSOP when a downstream SOP event is sensed. This seemed wrong as a proper 'start of frame' packet (SOF) ends with a EOP token. In addition, it was unclear what should happen if the repeater is in state WFEOP when the EOF2 event occurs. Accordingly, in our own first model, there is a special state WFEOP_, which is almost the same as WFEOP, except for the fact that in this state, the packet whose end is being waited for is known to be the SOF packet; at least if the host performs well, which is an assumption throughout the hub specification.

We still ought to explain the so-called EOF1 and EOF2 points. These are particular points in time, to be signalled by local timers in each device, which occur when the end of frame is approaching. EOF1 occurs first: at EOF $-$ 32 bits. EOF2 occurs next: at EOF $-$ 10 bits.

Yet another diagram was published by [6], as shown in Figure 3, which came to our attention a few months later. Notice that the transition from state

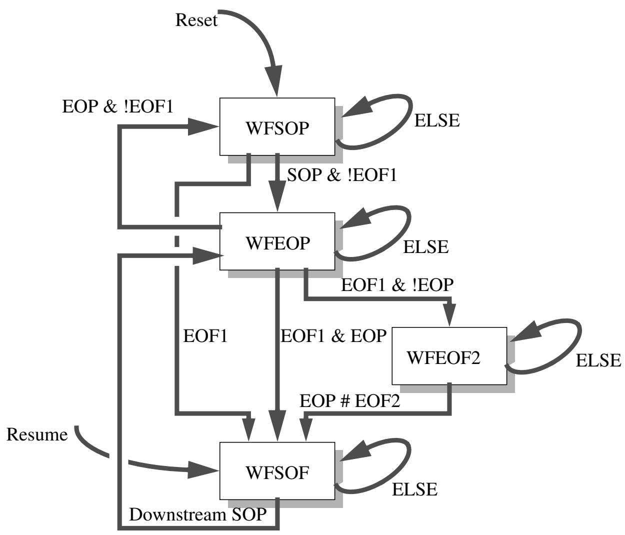

Figure 3: Repeater state machine (as given by Jaff and Morris)

WFSOF now properly goes to an intermediate state to await an EOP token. On the other hand, there is no additional WFEOP_.

2 A first Promela model

Below we describe a small Promela model of the hub protocol. The following protocol aspects are modeled:

- precisely four downstream ports[2],
- hubs can be instantiated and connected by cables,
- SOP and EOP messages,
- EOF1 and EOF2 events,
- relaying and broadcasting of messages,
- ports can (only) be disabled or enabled,
- full hub repeater state machine (cf. Figure 2).

There are also a number of aspects which are not modeled:

- Suspend/resume matters (for putting devices into a state of rest).
- (Quantitative) timing aspects.
- Parametrisation of the USB configuration.
- The port statuses "disconnected" and "suspended".
- Packet contents: packets are modeled as a pair consisting of a SOP and an EOP.
- The enumeration protocol, where each hub and each device is assigned an address.
- The interpretation of hub commands such as resets.
- Line events such as frame errors.
- Low/high speed matters.
- Power switching.
- The USB protocol from Chapter 8 of [3].

We omit the obvious definition of constants for Booleans (`false`, `true`), for packet-related events (`SOP`, `EOP`), for timer-related messages (`START`, `EOF1`, `EOF2`), for repeater states (`WFSOP`, `WFEOP`, `WFEOP`, `WFSOF`, `WFEOF2`), and for hub port states (`DISABLED`, `ENABLED`). The first two macros (`port_enable`, `fr_error`) allow the control of hub port states. After that we have various hub 'subroutines': `enabled`, `snd`, `broadcast`, `FrameTimerReset`, `any_EOP`, `any_SOP`. `FrameTimerReset` sends a `START` over channel `time`. In order to ensure that synchronisation with the `timer` process, the `EOF`s may have to be skipped first.

[2] The ports in the direction of the leaves of the USB tree are called *downstream*; the (single) port towards the host is the *root port*.

```
#define port_enable(P)                              \
  if                                                \
  :: (pstate[P]==DISABLED) -> pstate[P]=ENABLED  \
  :: (pstate[P]!=DISABLED) -> skip               \
  fi

#define fr_error(P)                                 \
  if                                                \
  :: (pstate[P]==ENABLED) -> pstate[P]=DISABLED  \
  :: (pstate[P]!=ENABLED) -> skip                \
  fi

#define enabled(P) (pstate[P]==ENABLED)
#define snd(P,M)                                    \
  if                                                \
  :: enabled(P) -> if                               \
                   :: P==0 -> p0!M                  \
                   :: P==1 -> p1!M                  \
                   :: P==2 -> p2!M                  \
                   :: P==3 -> p3!M                  \
                   :: P==4 -> p4!M                  \
                   fi                               \
  :: else skip                                      \
  fi
#define broadcast(M)                                \
  snd(1,M); snd(2,M); snd(3,M); snd(4,M)

#define FrameTimerReset                             \
  if                                                \
  :: time?EOF1 -> time?EOF2 -> time!START        \
  :: time?EOF2 -> time!START                     \
  :: time!START                                  \
  fi

#define any_EOP                                     \
  p0?EOP :: p1?EOP :: p2?EOP :: p3?EOP :: p4?EOP
#define any_SOP                                     \
  p0?SOP :: p1?SOP :: p2?SOP :: p3?SOP :: p4?SOP
```

After these preparations the main hub model can be given. It is described in an SDL style, where each of the main states (variable `state`) has a list of alternative receive statements, one for each possible message. For each alternative message, there is a transition which consists of assignments to the five port states, the directivity of the packet transfer, as well as outgoing messages and the next state. The `timeout` construct is a test that becomes enabled when all other processes are blocked.

```
proctype hub(chan p0, p1, p2, p3, p4, time)
{
  short state;       /* repeater main state         */
  short pstate[5];   /* port states                 */
  short src;         /* connectivity indicated by
                        source                      */
  short x;           /* buffer for don't care value */

  port_enable(1);
  port_enable(2);
  FrameTimerReset;
  do
  :: (state==WFSOP) ->
     if
     :: p0?SOP -> src=0; state=WFEOP; broadcast(SOP)
     :: p1?SOP ->
        if
        :: enabled(1) -> src=1; state=WFEOP; p0!SOP
        :: else skip
        fi
     :: p2?SOP ->
        if
        :: enabled(2) -> src=2; state=WFEOP; p0!SOP
        :: else skip
        fi
     :: p3?SOP ->
        if
        :: enabled(3) -> src=3; state=WFEOP; p0!SOP
        :: else skip
        fi
     :: p4?SOP ->
        if
        :: enabled(4) -> src=4; state=WFEOP; p0!SOP
        :: else skip
        fi
     :: any_EOP
     :: time?EOF1 -> state=WFSOF
     :: time?EOF2
     fi
  :: (state==WFEOP) ->
     if
     :: any_SOP
     :: p0?EOP ->
        if
        :: src==0 -> state=WFSOP; broadcast(EOP)
        :: else skip
        fi
     :: p1?EOP ->
        if
        :: enabled(1) && src==1 -> state=WFSOP; p0!EOP
        :: else skip
        fi
     :: p2?EOP ->
        if
        :: enabled(2) && src==2 -> state=WFSOP; p0!EOP
        :: else skip
        fi
     :: p3?EOP ->
        if
        :: enabled(3) && src==3 -> state=WFSOP; p0!EOP
        :: else skip
        fi
     :: p4?EOP ->
        if
        :: enabled(4) && src==4 -> state=WFSOP; p0!EOP
        :: else skip
        fi
     :: time?EOF1 ->
        if
        :: (src>0) -> state=WFEOF2; p0!EOP
        :: else skip
        fi
     :: time?EOF2 ->
        if
        :: (src==0) -> state=WFEOP_; FrameTimerReset
        :: else skip
        fi
     fi
  :: (state==WFEOP_) ->
     if
     :: any_SOP
     :: p0?EOP -> state=WFSOP; broadcast(EOP)
     :: p1?EOP
     :: p2?EOP
     :: p3?EOP
     :: p4?EOP
     :: time?EOF1
     :: time?EOF2
     fi
  :: (state==WFEOF2)->
```

```
    if
    :: any_SOP
    :: p0?EOP
    :: p1?EOP ->
       if
       :: enabled(1) && src==1 -> state=WFSOF
       :: else skip
       fi
    :: p2?EOP ->
       if
       :: enabled(2) && src==2 -> state=WFSOF
       :: else skip
       fi
    :: p3?EOP ->
       if
       :: enabled(3) && src==3 -> state=WFSOF
       :: else skip
       fi
    :: p4?EOP ->
       if
       :: enabled(4) && src==4 -> state=WFSOF
       :: else skip
       fi
    :: time?EOF1
    :: time?EOF2 -> fr_error(src); state=WFSOF
    fi
  :: (state==WFSOF) ->
    if
    :: p0?SOP ->
       FrameTimerReset; state=WFEOP_; broadcast(SOP);
    :: p1?SOP
    :: p2?SOP
    :: p3?SOP
    :: p4?SOP
    :: any_EOP
    :: time?EOF1
    :: time?EOF2
    fi
  od
}

/* test drivers, stubs and timers */

/* The host sends, successively, an SOP and EOP and
   then waits for a similar reply from a device.
   If there is none, it continues. */
proctype host(chan c)
{
     do
     :: c!SOP -> c!EOP -> if
                          :: c?SOP -> c?EOP
                          :: timeout
                          fi
     od
}

/* An eager device replies to the SOP \& EOP sent by
   the host. */
proctype eager_device(chan c)
{
     do
     :: c?SOP -> c?EOP ->
        c!SOP -> c!EOP
     od
}

/* A lazy device does not reply. */
proctype lazy_device(chan c)
{
     do
     :: c?SOP -> c?EOP
     od
}

proctype timer(chan c)
{
     do
     :: c?START -> c!EOF1 -> c!EOF2
     od
}
```

2.1 Simulation of the first model

```
chan cable[9] = [0] of { mtype };
chan time [2] = [0] of { mtype };

/* example topology:
 *
 *                  H
 *                  |
 *                 [0]
 *                  |
 *             HUB__REPEATER----time[0]---TIMER
 *             |   |   |   |
 *            [1] [2] [3] [4]
 *             |   |   |   |
 *             |   D
 *             |
 *       HUB__REPEATER----time[1]---TIMER
 *        |   |   |   |
 *       [5] [6] [7] [8]
 *        |   |   |   |
 *        D   D!!
 *
 */

init{
  atomic{
    run timer(time[0]) priority 1;
    run timer(time[1]);

    run host(cable[0]);
    run hub (cable[0],cable[1],cable[2],
             cable[3],cable[4],time[0]);
    run hub (cable[1],cable[5],cable[6],
             cable[7],cable[8],time[1]);
    run lazy_device (cable[5]);
    run eager_device(cable[6]);
    run lazy_device (cable[2]);
  }
}
```

We refer to Figure 4.

2.2 Conclusions of the first modelling

The following conclusions can be drawn from this initial modelling activity. First, simulation of the PROMELA model shows many modelling artefacts, caused by the way in which events are mapped to PROMELA constructs and by the level of abstraction of the model. For example, the decision to model time by communication of messages (`START`, `EOF1`, and `EOF2`) causes a discrepancy between the actual condition "`SOP & !EOF1`" from `WFSOP` to `WFEOP` in Figure 2 and its implementation (see the case `state==WFSOP` of the `hub` proctype). The latter may also transition

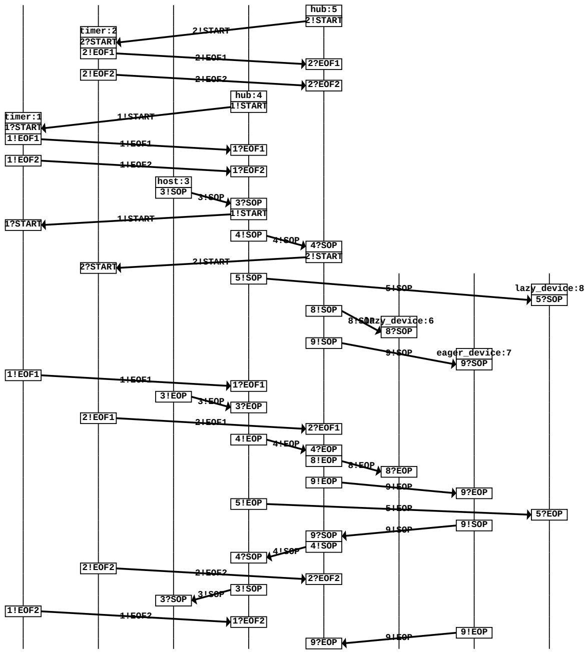

Figure 4: Simulation run of the first model

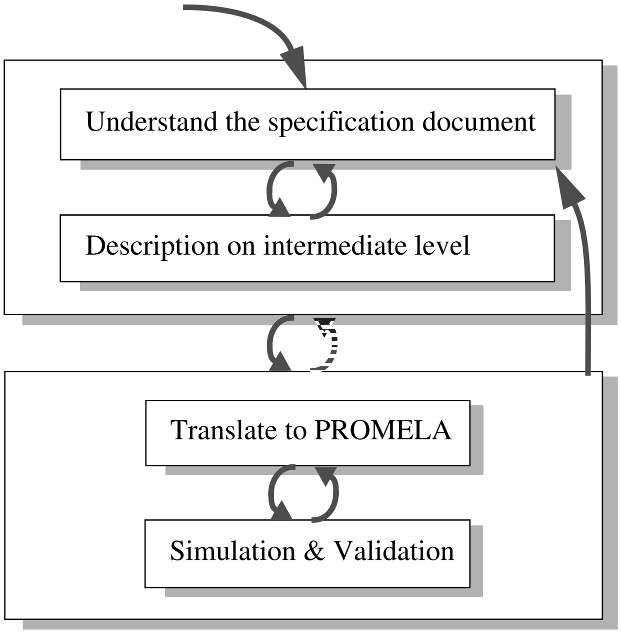

Figure 5: Approach

to the `WFEOP` state if `SOP` and `EOF1` occur simultaneously. Indeed, the first model does not deal with real time aspects. There are local timers, but Spin does not know whether a single clock tick, or 100.000 clock ticks have passed between e.g. a START and an EOF1. Yet this may be important to verify quantitative timing properties.

Second, an important aspect of the hub's functionality has been ignored: all matters of suspend/resume signalling have been left out. This is one of the more complex parts of the hub which furthermore exposes some behaviour that is particularly challenging from the point of modelling, like using the possibility to drive a wire from two sides (e.g. a PC and a hub, or a peripheral device and a hub). At the electrical level this is possible, provided both sides drive the same signal (high Voltage or low Voltage). It is clear that the above Spin model is far too abstract for covering these things, and yet they are essential for the suspend/resume protocol.

Finally, we were faced with a number of striking unclarities in the official specification.

3 Modelling More

These conclusions have instigated us to make a second, more elaborate model of the hub, while applying a phased approach as outlined in Figure 5. We have decoupled the phases of understanding the specification and constructing the Promela model by inserting a phase in which a description on an intermediate level is given, seeking a means of description that is more formal than the specification document, but still flexible enough to allow for frequent changes. This was expected to reduce the amount of work involved in iterating between the text of the specification and our understanding of it as reflected in the more formal description. Only after this phase stabilises, the intermediate level description would then be translated into Promela, filling in additional details of the modelling and bringing it to a completely formal level. The activity of performing simulation and verification runs usually first brings out many shortcomings which are specific to the mapping to Promela, resulting in iterations indicated in the lower part of Figure 5. Only after these have stabilised, mistakes in the higher-level modelling decisions, and possibly in the original specification, will be brought to the light.

The additional aspects of the hub taken into account are:

1. suspend/resume behaviour of the repeater;
2. more faithful timers;
3. a more general model of the hub architecture, allowing for an arbitrary number of ports;
4. ports can be disconnected, disabled, enabled, or suspended;
5. host requests to the hub controller (to control the ports).

The subsections below correspond to the approach outlined in Figure 5. We abbreviate upstream by u.s. and downstream by d.s.

3.1 Understanding the specification documents

A good understanding of the specification[3] is a vital first step in modelling the hub protocol. However, it also proved to be the main difficulty so far, for a number of reasons. Information is scattered throughout the document, and not seldomly overlapping pieces of information contain inconsistencies. The style is informal (plain English with explanatory pictures), which need not be a problem in itself, were it not that a certain degree of rigour is lacking. Finally, it is often only explained *how* things are done, but not *why*. More conceptual insight would be helpful, e.g. in solving ambiguities.

To have a better starting point for the model, we have extracted and ordered the relevant parts from the specification documents while correcting mistakes, explicitly listing design and modelling decisions, and giving additional conceptual information. The following top-down description of the hub functionality is based on [3] and [7], with additional information from [6] and [1]. We stress that it is not meant as a substitute for the original texts, but rather as an addition.

1. Decomposition of a hub A hub contains the following subcomponents: the repeater, the controller, and a number of downstream ports.

The subcomponents operate in parallel. Each type (repeater, controller, port) is responsible for several aspects of the hub's functionality. [3] and [7] are not always clear on which aspects precisely each subcomponent covers. We will clearly indicate our modelling decisions concerning the assignment of functionality to subcomponents.

The repeater takes care of connectivity and exception handling (such as bus fault detection and connect/disconnect detect); see paragraph 1.1 below. The downstream ports keep track of whether or not devices are connected and if so, control the status of the connection (enabled, disabled, suspended); see paragraph 1.2. The controller serves as the host's peer in host-to-hub communication; see paragraph 1.3.

Modelling decisions: Suspend/resume signalling will, as much as possible, be modelled as part of the repeater functionality ([3] suggests that resume signalling is an aspect of connectivity behaviour, which is a task of the repeater).

1.1. Repeater Connectivity behaviour of the repeater differs depending on whether it is propagating packet traffic, resume signalling, or in the idle state. In the idle state all ports are waiting for the start of a next packet. As such, there is no difference between this idle state and the logical states WFSOP and WFSOF[4] of the repeater. For this reason, we do not consider the idle state separately.

1.1.1. Propagation of packet traffic This is either upstream or downstream connectivity. Upstream packets are only propagated from the port that initiates the traffic (the *initiating* port) to the root port. Downstream packets are forwarded from the root port to all enabled d.s. ports.

The propagation of packet traffic is slightly complicated by the fact that the repeater is responsible for certain types of exception handling, namely babble and loss of activity (LOA), during u.s. connectivity. Without this feature, the repeater would just react to SOP and EOP packets, transitioning back and forth between the states WFSOP and WFEOP. However, babble or LOA could then cause deadlocks.

In order to gracefully recover from u.s. babble/LOA, the frame timer mechanism is used. The SOF packets are used by the hub to synchronise its own frame timer, which may be skewed with respect to the host's clock. These SOF packets hold the highest priority access to the bus, and are generated (by the host) regardless of the other bus activity. It is the responsibility of the hub to clear the way and allow for host access to the bus. Each hub has a frame timer that is reset at the start of each new frame[5]. It is then used by the hub to determine the point at which the bus must be silent in order to allow the next SOF. The idea is that those d.s. ports that are still taking up the bus (either because they keep babbling or because LOA has occurred) when the timer is getting near the end of the frame, are disabled. However, this strategy should be implemented with care so as to prevent "good" hubs to be shut off inadvertently. For example, consider a chain of hubs, with the top hub connected to the host and the bottom one to a babbling device. If each hub in the chain would just let the babble propagate upstream until the "time-out point", then as a result all hubs, except for the top one, will be shut off[6], because the top hub disables its downstream port to which all others are connected. Instead, a 2-phase time-out strategy is followed: at EOF1, each hub must terminate its own upstream connectivity — so as to save itself from being cut off — while at a later point EOF2, the initiating d.s. port is checked for babble/LOA and possibly disabled. The timing of the EOF1 and EOF2 points are computed in [3] and in [7]. Here, we do not con-

[3] In particular, we refer to Chapter 11 of [3], and its proposed revision, [7].

[4] The WFSOF state is renamed into WFDSOP in [7].

[5] Actually, up to 2 SOFs may be missed.

[6] Actually, as a result of host-hub skew, the scenario may also be different.

sider timing aspects.

Thus, apart from the states WFSOP and WFEOP, between which the repeater alternates within one frame, the repeater has a WFSOF state to which it transitions near the end of the frame (at EOF1 and provided the connectivity is not u.s., to be precise) and in which it awaits the next d.s. SOP (which is supposed to be the start of the SOF packet). Furthermore, there is a WFEOF2 state in which the repeater ends up when at EOF1 the u.s. traffic has not been ended by an EOP. In WFEOF2 it waits until the u.s. EOP still is detected, or otherwise until EOF2 occurs, in which case the initiating d.s. port has to be disabled. We arrive at the corrected repeater state machine shown in Figure 6. While we have devised

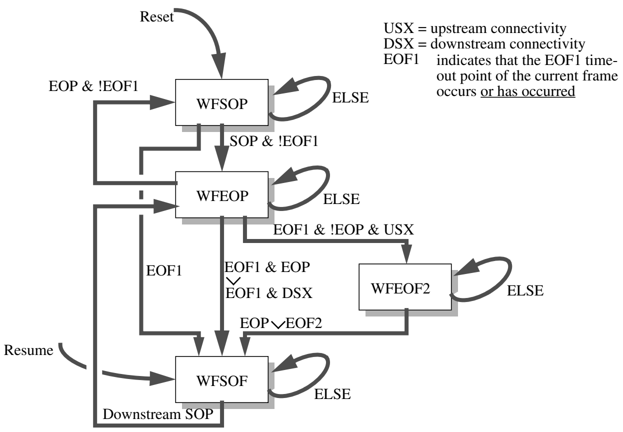

Figure 6: Repeater state machine (as used in extended model)

this improved version on the basis of the text[7] of [3], it is interesting to note that the revision of Chapter 11 proposed in [7] has independently come up with a very similar corrected picture.

1.1.2. Suspend/resume signalling The repeater displays different connectivity behaviour when resume signalling is being propagated through the bus. The suspend/resume mechanism is provided in order to place the USB, or parts of it, into an energy-saving state (suspend, or sleep). This is implemented by the possibility for each individual device (including hubs), independently of others, to transition to a suspended state. Two distinctive features of the mechanism are the following.

hub suspend vs. port suspend The host can send (via the hub controller) suspend requests to individual ports. This allows it to suspend certain segments (namely, the segment below a certain port) only. Thus, besides the suspended/awake state of a hub itself, also each of its individual ports may be suspended or awake. The suspended state of a port is a distinguished logical state. Any other logical state then implies that the port is awake (although for ports in the disabled or disconnected states this does not mean much). Likewise, we introduce a logical state called "suspended" to the hub repeater state diagram. Again, the hub is awake when it is in any other state. The suspend state of the port and the suspend state of the entire hub (or of a device) should not be confused. They are independent — indeed, any of the 4 combinations of possibilities may occur in practice.

host-initiated vs. device-initiated resume Resume signalling may originate not only from the host, but also from a device. This allows devices to independently wake up the bus, without the host having to monitor them. For example, think of a key on a keyboard being hit. Thus, resume signalling may be *host initiated* or *device initiated*. Independent from this, resume signalling may flow upstream and downstream. We will see that in the case of device initiated resume, both flow directions occur.

The general rules then, for any device, are as follows.

- For suspend: Any device[8] that sees a constant idle on its root port for more than 3.0 ms, transitions to the suspend state (from any state). It is not clear from the documentation whether a suspend is caused by a constant idle on the *root port* or a constant idle on *all ports*[9]. We assume that a constant idle on the root port is enough, with the idea that the host will not silence when a device is still talking — this is then a property of some higher level. So it will not be the case that a hub sees a constant idle on its root port while some (non-disabled) d.s. port is still signalling.

- For resume: A device can return from its suspend state[10]:

  – Upon seeing normal bus activity on any port. We assume that a resume may be initiated by (1) an idle-to-resume on the root port, and (2) *any* activity on a d.s. port.
  – Via reset signalling.

[7] [1], on p. 132, says the following about the discrepancy between text and figure: "The state diagram in the specification differs from the written description and it is the author's opinion that the written description is correct."

[8] Recall that a hub is a special type of device.

[9] E.g., compare [3]:p. 234 ([7]:p. 235), 11.5.1, second sentence with [3]:p. 119, 7.1.4.4, third and fourth sentences.

[10] This process is called resume, but there is no unique resume state. The logical state after resume may depend on the state before suspending and on the behaviour of the device.

– When activated by actions that are associated with internal functions (think of hitting a key on a keyboard) — this is called *remote wake-up*.

Below, we concentrate on suspend/resume behaviour of *hubs*.

1.1.2.1. Suspend By ceasing activity on a bus line, all devices below it will enter the suspend state. A bus line may be silenced by putting the d.s. port to which it is connected in the suspended state (via a SetPortFeature(PORT_SUSPEND) request — which is only understood by an awake hub. Of course the host may also decide to stop sending any signalling down the USB; in this case the whole USB will enter the suspend state. This is called *global suspend* in [3]. Suspending parts of the USB only, through port suspend, is called *selective suspend* (device level). We view global suspend as a special case of selective suspend.

Upon suspending, a hub: floats all output drivers; maintains static values of all control and status bits; preserves current state information for all d.s. ports; turns off its clocks.

1.1.2.2. Resume Resume behaviour is more complex. Here are the general rules:

initiation of resume: host/device Resume is initiated by the host or by a device. A host-initiated resume is effected by a reset or by resumption of d.s. bus activity. D.s. bus activity may either be resumed from the host directly (in case of a global suspend), or by a hub in response to a request from the host[11]. A device initiated resume is effected by a device that starts bus activity (e.g. in response to actions associated with internal functions) or by a hub that detects a connect/disconnect.

scope of resume A host initiated resume wakes up the entire segment of the bus below the point where the bus activity is resumed, except those parts "behind" suspended ports. In this case, the resume scenario only involves d.s. traffic. A device initiated resume wakes up the following segment of the USB: all hubs and ports that can be reached by going u.s. from and including the point where the resume is initiated up till (but not including) the first awake hub that is encountered, and from all these hubs, all segments that can be reached by going d.s. along enabled ports. In this case, both u.s. and d.s. traffic occur.

[11] Note that such a request can only be sent to, and only be received by, an awake hub.

end of resume Any resume scenario should result in a situation where the segment that has woken up is ready to receive d.s. bus signalling. In particular, the woken-up segment must be able to start taking part in the periodic reception of the SOF pulse that is used to synchronise its clocks and to keep it from suspending again. Any resume scenario ends with a d.s. resume signal being propagated for 20 ms, followed by a low speed EOP.

Modelling decision: We model the suspend-to-resume transition of a hub as an instantaneous action. In reality, this takes an amount of time (mainly because clocks have to be restarted, which may well last several frames.

1.1.2.2.1. Down-stream resume signalling A hub that sees a resume signal on its root port (this is not necessarily caused by a host initiated resume, but may also occur in those segments where d.s. resume signalling is generated in a device-initiated scenario (see below)) reflects this signal d.s. on all its enabled ports. We assume that the resume signal is ended by an EOP.

1.1.2.2.2. Up-stream resume signalling Here, the general scenario is as follows: The resume signalling propagates u.s. until it reaches an awake hub (called the *top hub* henceforth) or the host. Along the way, each *suspended* hub that sees the u.s. directed resume signal not only propagates it u.s., but also sends it *downstream* along all enabled ports, and also reflects it to the d.s. port on which the resume was received. We assume that it is the responsibility of the host to alter (or leave) the status of that port, i.e. to move it to the enabled state through a ClearPortFeature(PORT_SUSPEND) request, after inspecting the status bits.

We describe the scenario in more detail. We distinguish

1. the *bottom hub*, which is the one directly above the device that initiates the resume, but only if it is a suspended hub (if it is not suspended, this means it is the top hub);
2. the top hub (see above); and
3. all intermediate hubs, that lie on the path from bottom to top hub.

bottom hub and intermediate hubs Each of these hubs (which are all suspended), exposes the following behaviour. In case it sees a *signal which is not a connect or disconnect* on a d.s. port (called the *signalled port*), then regardless whether this port itself is enabled or suspended, a *resume* signal is generated u.s. to its rootport, to the signalled port, as well as to all other *enabled* ports. In case the signalled port is enabled, no status & interrupt bits are set. In case it is suspended, the status & interrupt bits are set, in order to notice to the host that this port should be enabled again.

In case the hub sees a *connect* or *disconnect* on a d.s. port, a *resume* signal is generated u.s. to its rootport, as well as to all *enabled* ports, except for the signalled port itself. Status & interrupt bits are set in this case. Note that we model the connect and disconnect as instantaneous signals. In reality, both depend on the duration of certain signals, and the hub has to return to an awake state first, with its clock started (it is stopped when the hub suspends). In order to model this more faithfully, we might have to perform the detection of (dis)connect into two separate phases: in the first, a possible occurrence of a (dis)connect is detected and the hub starts returning to an awake state; in the second, after the clock has started, the detection of the (dis)connect can be finished.

Note that the resume signalling that is generated in the d.s. direction causes hubs to react as specified in paragraph 1.1.2.2.1 above.

top hub The top hub sees a resume, a connect, or a disconnect on a *suspended port*. Then it needs to handle this without interfering with other traffic that may possibly be propagated via its other ports. Therefore, in case of a resume signal, it reflects it d.s. (in order to acknowledge) only on the signalled port; and in case of a connect/disconnect, nothing is reflected but only status & interrupt bits are set.

1.2. Ports Modelling of a single port is along the description in [3]. In principle, a port only maintains an internal state and, in certain cases, sets status and interrupt bits on reception of certain signals on its associated bus. One exceptional case is when a suspended port on an awake hub detects activity on its associated bus. In this case it takes care of reflecting a RESUME signal. Logically speaking, this would be the task of the repeater. However, as the hub is awake, the repeater may be occupied by other ongoing traffic through the hub — whence we have put this task as part of the port's functionality.

1.3. Controller Just as any other device, also the hub should be able to communicate with the host. It is the controller that takes care of this functionality. It intercepts those packets that are addressed to the hub from the bus and interprets them; this is in contrast to the repeater which just "blindly" propagates any packet regardless of its contents.

Currently, the only types of messages that our modelling of the controller considers are the port requests (of the form SetPortFeature/ClearPortFeature). The controller distributes these to the right ports.

For communication to the controller, we have introduced a separate channel for host-to-hub communication, which is independent from the bus. This is justified because at a more conceptual level the host-to-hub communication is also described as if it took place via a separate channel — we have to be cautious of possible modelling artifacts though.

3.2 Intermediate-level description

Before modelling the hub in PROMELA, we have fixed a description in an intermediate format in the style of *action systems* ([2]). While this (ad-hoc) notation allowed for a more concise and precise specification of the state-machines of the individual components, it still allowed for a certain looseness in description by having no formally defined syntax or semantics. Our decision to use such an intermediate format was based on the experience that the translation from a specification in natural language to a formal language is a step which typically involves a number of iterations. The process of formalisation exposes ambiguities, incompletenesses and mistakes in the natural-language specification which are incrementally corrected and taken into account at the more formal level. The more pedantic are the rules at this formal level, the more time these back-and-forth steps take. The intermediate level, with a more restrictive format than natural language, but not imposing the strait-jacket of a programming language, enforced enough rigidity so that we had to rethink the original specification over and over, but without involving the overhead of book-keeping too many details. A fragment of the action-system notation is shown in Figure 7. It describes the possible actions of the repeater when it is waiting for an `SOP`. Several aspects have been modelled by variables, like the logical state of the repeater (`repeater_stat`), the signal on the buses connected to the root port (`rootport_bus`) and to the downstream ports (`port_bus[i]`), and the port statuses (`port_status[i]`). On the other hand, things have been left informal, like the condition `''rootport_bus has been silent for at least 3.0 ms''` in the last action. Also, while the time has been modelled by a variable (`t`), it is not updated but assumed to proceed implicitly.

Indeed, Section 3.1 above is the result of this itera-

```
/*********/
/* WFSOP */
/*********/

repeater_stat = WFSOP & rootport_bus = (father,SOP)
& t < EOF1
  -> PARALLEL i : i in DPORT
      & port_status[i] = ENABLED :
        port_bus[i] := (hubid,SOP)
      ENDPARALLEL ;
      src := 0 ; repeater_stat := WFEOP

CHOICE i : i in DPORT :
  repeater_stat = WFSOP & port_bus[i] = (son[i],SOP)
  & port_status[i] = ENABLED & t < EOF1
    -> rootport_bus := (hubid,SOP) ;
        src := i ; repeater_stat := WFEOP
ENDCHOICE

repeater_stat = WFSOP & t = EOF1
  -> repeater_stat := WFSOF

repeater_stat = WFSOP & ``rootport_bus has been silent
for at least 3.0 ms''
  -> repeater_stat := SUSPENDED
```

Figure 7: Fragment of intermediate level description

tive process reaching a stable state. Stabilisation also indicated that it was time to proceed to the next step, namely transforming the description into PROMELA.

3.3 Translation to Promela

The first and foremost decision is how to model time. The HUBs in an USB topology operate in an essentially synchronous mode. Conceptually, a common global clock is maintained by the USB devices; indeed, the (local) clock synchronization imposed at the start of each frame is there to ensure a common time frame. During a clock cycle each device senses the bus segments to which it is connected, reacts to it and possibly drives its bus segments to a new state which is only sensed by other devices during the next cycle. Also, two devices cannot simultaneously (i.e., within a clock cycle) drive the same bus segment to different values.

The PROMELA model is structured accordingly: each device is modelled as a separate process instance which cycles through a Read phase during which the bus is read and a Write phase during which the bus can be written (but not read). No process can enter a phase until all other processes are ready to enter the same phase. This structure can be seen in Figure 8 which list the somewhat simplified code corresponding to the first two choices of the fragment in Figure 7. `RSTEP(prio)` and `WSTEP(prio)` are macros[12] that enforce the above mentioned synchronization. Because of this phasing, we can schedule the processes *de-*

[12] I.e., C–preprocessor `#defines`

```
            /* Read phase: */
RSTEP(prio);
ReadClEvent(my_t,clock_out[my_id]);
i = 1;
do
:: i <= ndports ->
   ReadBus(my_bus[i],bus[lport + i - 1]);
   i++
:: else break
od; i = 0;
ReadBus(my_bus[0],bus[my_id]);

                /* Write phase: */
if  /* WFSOP */
:: device_stat[my_id] == WFSOP
   && my_t != EOF1 ->
   /* Randomly select a port
      with activity. */
   src = (BusContent(my_bus[0],
          father_id,SOP) -> 0 : -1);
   i = 1;
   do
   :: i <= ndports ->
      if
      :: BusContent(my_bus[i],lport+i-1,SOP)
         && port_stat[lport+i-1]==ENABLED) ->
         if :: src != -1 :: src = i fi
      :: else
      fi; i++
   :: else break
   od;
   if
   :: src == 0 -> /* Downstream */
      i = 1;
      do
      :: i <= ndports ->
         if
         :: port_stat[lport+i-1] == ENABLED ->
            WriteBus(bus[lport+i-1],my_id,SOP)
         :: else
         fi;
         i++
      :: else -> break
      od;
      device_stat[my_id] = WFEOP;
   :: src > 0 -> /* Upstream */
      WriteBus(bus[my_id],my_id,SOP);
      device_stat[my_id] = WFEOP;
   :: else /* No activity */
   fi;
```

Figure 8: Fragment of Promela code

terministically, thus reducing the number of separate computations to analyse. Indeed, there is an explicit scheduler process and the `prio` arguments determine the scheduling order.

Global time, then, is measured as the number of Read/Write cycles and the local HUB time as the number if cycles since the last `RESET` occured at start-of-frame; note that we are ignoring clock skew for the moment.

In this set-up, the bus segments are most naturally modelled as shared variables. Unfortunately, the 'message sequence charts' that the Spin verifier can produce only depict communication events. For this reason we opted to use 1-buffered communication channels. Some added complexity was hidden by using macros (e.g., `ReadClEvent`, `BusContent`, etc.)

Still, a simple comparison of these two fragments gives a strong argument for using an intermediate description first.

4 Conclusions

We have given an account of our experiences in modelling the USB hub using PROMELA. While this is an ongoing project, some conclusions can yet be drawn. Our main effort so far has been aimed at understanding the functioning of the hub's components in detail. Being forced to describe this in a formalism is an excellent way to be sure to consider all chosen aspects with equal (and great) precision. Many unclarities, inconsistencies and errors in the original text of the specification have been uncovered during this process. The use of an intermediate level between the specification document and the PROMELA model proved useful to separate the concerns of understanding the hub on the one hand, and modelling this using PROMELA's capabilities on the other.

Now that the hub's functioning, as we understand it, has been fixed, we currently focus on simulation and verification in SPIN. One difficulty here is that the description of the hub, as extracted from the description in [3], has become a single monolithic object. Interestingly, it is not so much SPIN that cannot handle its overall size — at least there have not been any indication of this so far. Rather, it makes debugging a labour-intensive job. Although the repeater, ports and controller have been modelled as separate PROMELA `proctype`'s, it proves to be difficult to further modularise these in tracking errors. Hence, scaling down these components is a major concern of ongoing work.

References

[1] Don Anderson. *Universal Serial Bus System Architecture.* The PC System Architecture Series by MindShare, Inc. Addison-Wesley Developers Press, Reading, Massachusetts, 1997.

[2] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd ACM SIGACT–SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, New York, 1983. ACM.

[3] Compaq, Digital Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC, and Northern Telecom. Universal serial bus specification. Downloaded from `http://www.teleport.com/~usb/present.htm`, January 15 1996. Revision 1.0.

[4] Gerard Holzmann. Spin package. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[5] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series. Prentice-Hall International, London, 1991.

[6] Kosar Jaff and Jeff Morriss. Hub architecture. Handout of the USB Developers Conference February, 1996. Downloaded from `http://www.teleport.com/~usb/present.htm`.

[7] Universal Serial Bus specification Revision 1.1 RC 2. Downloaded from `http://www.teleport.com/~usb/specup.htm`, May 1997.