Formal model of the Bluetooth Inquiry Protocol

Hugo Brakman Vincent Driessen Joseph Kavuma Laura Nij Bijvank
Sander Vermolen

April 2006

1 Introduction

Bluetooth is a widely used communication protocol these days. It is used in the communication between phones, computers, headsets and many more devices. In the year 1994 the Ericsson company decided it wanted a protocol that could be used to connect mobile phones to other devices. Jaap Haartsen, working for Ericsson, developed the protocol. The techniques were further developed by the Bluetooth Special Interest Group.

One of the things described in the protocol is the way two devices that are neither connected nor synchronized can try to find each other. This is called the Inquiry Response Phase, the first phase in the protocol, and should provide a way for the devices to synchronize in order to allow further communication.

We have looked closer at the specification of this phase as described in [1] and created an UPPAAL model to formally verify that after the Inquiry Response Phase indeed the devices will be synchronized.

In this paper, we will first give an informal explanation of the Inquiry Phase, the Inquiry Scan Phase and the Inquiry Response Phase. We will then look closer to how we modeled these phases in detail. We will then look at some of the problems we ran into and the decisions we have made to solve these. In the final part of the paper we will state our conclusions and make some suggestions for possible further research.

2 Informal description

When two Bluetooth devices want to start communicating they do that using the Inquiry Phases. In this phase one of the devices is assumed to be in master mode querying for other devices. The other device is assumed to be in slave mode. The master keeps sending packages and listening for responses. The slave will listen for a package from the master and respond to the master by sending a return package.

The devices do, however, change frequency during every phase. The frequencies used in Bluetooth are very common frequencies used in wireless phones, remote controls, garage doors and more. Therefore the devices change ("hop") their frequencies a lot. Probably the devices will not use the same frequency the first time and the communication attempt will fail. However, the hopping should be done in a way that at a certain point in time the devices will use the same frequency in the same time interval and further synchronization can be achieved using that.

There are quite a few tricks involved in order to get this to work properly. There is the hopping of frequencies, timing issues in sending, receiving and listening and some more.

We want to verify that indeed the devices will eventually synchronize in all cases if we follow the specification.

3 Detailed description

In this section, we will give a top-down overview of the formal model of the Bluetooth Inquiry Phases. The official specification introduces the terms *slave* and *master* to mean the following:

> "Although master and slave roles are not defined prior to a connection, the term master is used for the inquiring device and slave is used for the inquiry scanning device."

This is exactly the definition that we will use in this document. Informally, inquiring can be seen as the process of trying to find a Bluetooth device in the direct neighborhood. Similarly, inquiry scanning is the process of a Bluetooth device scanning the environment for inquiring devices that are trying to communicate. This is the integral process of the Bluetooth inquiry sub state—there will be no communication apart from "finding each other". After there has been a successful handshake, the devices advance to the next sub state of the Bluetooth protocol, in which we are not interested within this research.

In order to come up with a formal description of what is going on, we tried to follow the prescriptive Bluetooth Standard Specification [1] as close as possible. In doing this, we've initially made a simplistic model in which we abstracted from all tricky details and made unrealistic assumptions, in order to avoid complexity. Examples of this are discussed in Chapter 4. After that, we have gradually removed assumptions and tried to model the thereby newly introduced complex constructs into the existing model, bit by bit, persistently assuring ourselves that the key properties of the system would still hold.

In the following sections, we will start with a top-down overview, after which we will incrementally strike down alongside the building blocks that together form the model.

3.1 Bluetooth clocks

One of the key objects within each Bluetooth device is the (internal) Bluetooth clock, also called the *native clock*. Bluetooth devices have to synchronize with each other, using time slots with a duration of 312.5 μs. Virtually all the timing of the Bluetooth communication is defined in terms of these time slots, which indicates the smallest common time unit.

Of course, an exact value of 312.5 μs is not feasible in real time systems. Since the clocks will consist of hardware clocks at the lowest level, there will always be the problem of clock drift and jitter, which should be taken into account.

Per device, there is a native clock counter, which is an array of 28 bits, posing a total number of $2^{28} = 268{,}435{,}456$ possible clock values. Since the clock increments each 312.5 μs, a full clock cycle will take almost a day (≈ 83886.08 seconds).

Using UPPAAL, we represented the ticking of a Bluetooth clock like the way it is displayed in Figure 1. The template consists of only one state, in which UPPAAL waits until the clock should be incremented. Upon taking the transition, UPPAAL resets an internal clock (representing the hardware clock), `time` and executes the `tickClock()` function, which takes care of increasing the array of 28 bits. The relevant source code for this template is shown in Figure 2.
As can be seen, the array of `CLOCK_LENGTH` (a constant set to 28) bits is passed along with the instantiation of this template. It is this array of booleans that is incremented each step, by the given algorithm.
Drift and jitter are implemented by letting UPPAAL wait a non-deterministic amount of time. This is done using the typical UPPAAL construct for this: by introducing a state invariant `time <= CLK_MAX` and a transition guard `time >= CLK_MIN`. We took a clock period of 625 μs (which is 2*312.5 μs, because UPPAAL can't handle real numbers, only integers), and a jitter of 6 and drift of 1. All of this implies that the Bluetooth clock is incremented each 623 to 629 UPPAAL time units (non-deterministically).

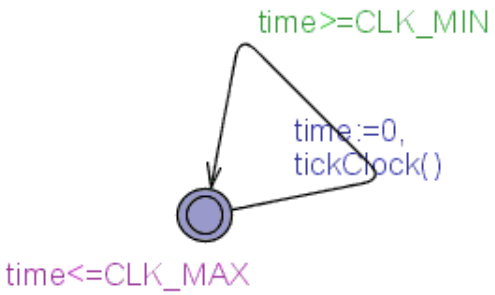

Figure 1: The Bluetooth Clock modeled in UPPAAL

3.2 Communication

Communication of messages must takes place at the start of a time slot and at no other time. Since two Bluetooth devices initially have no notion of each other, they have not yet synchronized their clocks with each other[1].

[1] In fact, this is not a part of the inquiry phase, but is done in a later stadium.

```
// Because some of the internal variables of this clock are needed elsewhere
// in the model, as we will discuss at a later point in time, the variables
// for the internal clock (CLKN) as well as the UPPAAL clock (time) are
// parameters of BluetoothClock
//
// Parameter definition:
// int[0,1] &CLKN[CLOCK_LENGTH], clock &time;

// Relevant global constants (from the global declarations file)
const int CLOCK_LENGTH = 28;    // Length of clock in bits
const int CLOCK_PERIOD = 625;   // Clock period times 2, because Bluetooth
                                // only supports discrete values

// Local (BluetoothClock) constants
const int HALF_JITTER = 3;
const int DRIFT = 1;
const int CLK_MIN = DRIFT + CLOCK_PERIOD - HALF_JITTER;
const int CLK_MAX = DRIFT + CLOCK_PERIOD + HALF_JITTER;

void tickClock()
{
        increment(CLKN);
}

void increment(int[0,1] &inputClock[CLOCK_LENGTH])
{
        for (i : int[0, CLOCK_LENGTH-1])
        {
                if (inputClock[i])
                        inputClock[i]:=0;
                else
                {       inputClock[i]:=1;
                        return;
                }
        }
}
```

Figure 2: Source code listing for the `BluetoothClock` template

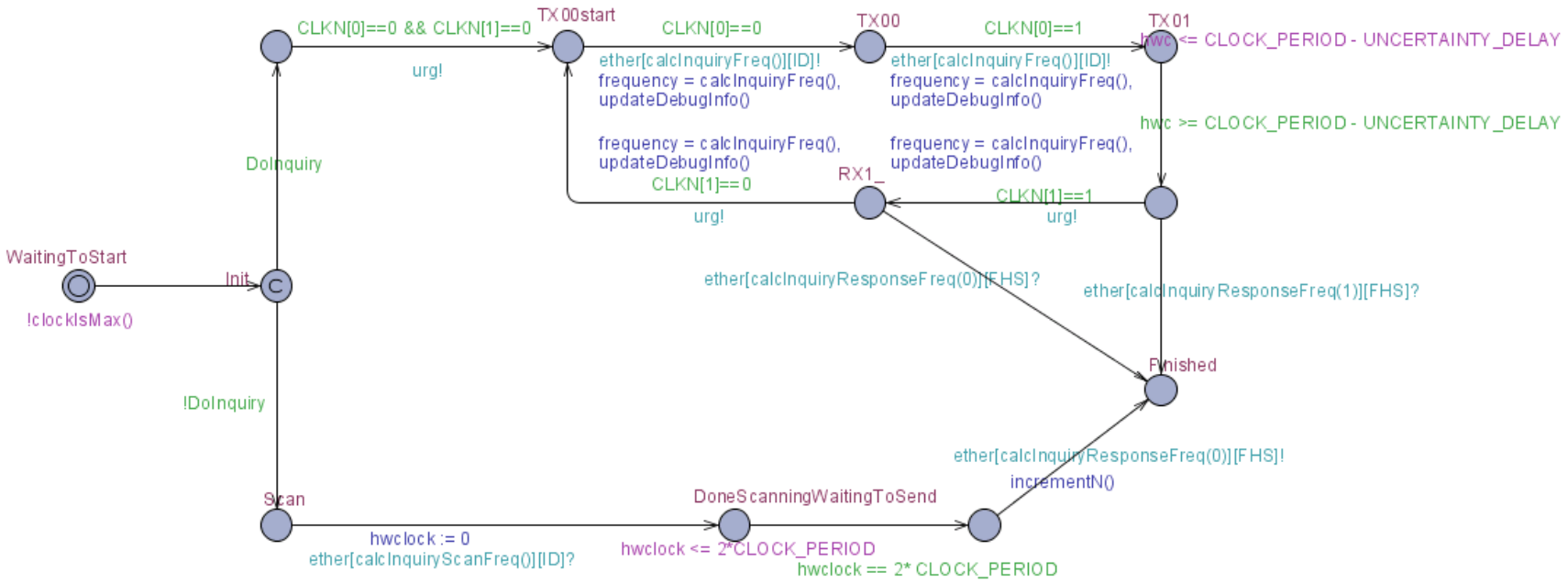

Figure 3: The full Device template as modeled in UPPAAL

Figure 3 displays the model of a Bluetooth device, which actually holds the model for both a master and a slave device, since most Bluetooth devices can be used as both a master and a slave. However, the choice for a device to become either one of them is a choice that implies a different behavior. The `DoInquiry` parameter to the template constructor actually determines for the instance of the device whether it will become a master or a slave. In our model, we have validated some properties of the system, using a simple communication model consisting of one master and one slave, each instantiated with a different value for `DoInquiry`.

Usually Bluetooth devices "contain" an internal BluetoothClock. Since UPPAAL does not fully support object oriented constructs like this, we have modeled this by defining two global native clocks (`int[0,1] CLKNs[NR_OF_DEVS]`) and passing corresponding references to these clocks to both a `Device` instance and a `BluetoothClock` instance.

The initial `WaitingToStart` state holds an invariant stating that the clock has not yet reached a certain maximal value. This technically introduces a maximal waiting time for the system to start, without having to introduce a new UPPAAL clock, which is effective avoidance of a state-space explosion. The waiting time itself avoids that the clocks have equal values along the way for a long time by waiting a random (maximal) amount of time.

In the next section, we will take a more detailed look at the implementation of the master and slave parts of the model.

3.3 The slave

To start with the simplest case, the slave life time goes through three states sequentially. First, the slave is ready to start a connection and starts listening in the environment for a master that is searching for it. After it receives a message from a willing master device, it will immediately go into a waiting state in which it has to wait for the second next clock tick to arrive, before it may send back a message to the master. This is a requirement because the master uses a send-send-listen-listen activity pattern, making the time space between a send action and the corresponding listen action exactly two Bluetooth clock ticks.

After this mandatory waiting time, the response packet is sent back to the master. Then, the inquiry phase is over and the slave's part is done here. This process is depicted in Figure 4.

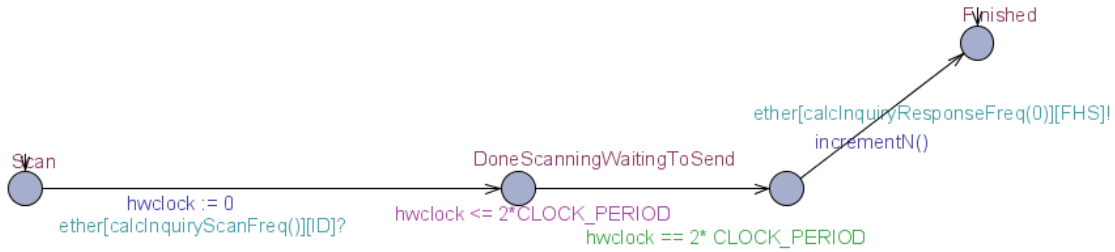

Figure 4: The device part relevant for slave devices

Of course, things are a bit more complicated then described here. Communication involves the sending of packets at certain defined moments in time and at certain broadcast frequencies.

Because a device can only listen or send at only one frequency at a time, and to avoid collisions of all Bluetooth devices (and others) in the same area using the same frequency, the Bluetooth specification defines a hopping sequence of frequencies, which follows a pseudo-random hopping pattern. The master follows a faster hopping sequence than the slave to broaden the changes to successfully find a slave device.

Urgent `ether` transactions

Furthermore, `ether` is defined to be a two-dimensional, urgent, broadcast channel. It is urgent to make sure that the transitions are taken as soon as they are possible. It is a broadcast channel to match reality (if a signal is broadcasted and nobody picks up the signal, the system should not deadlock). Finally, the two-dimensions of the channel definition are a construct to match frequency relevance on which the packet is broadcasted (signals broadcasted on frequency X should not be picked up by a device listening on frequency Y, if X is unequal to Y) and the type of the packet. For the inquiry phase of the Bluetooth protocol, the type of packet can either be an ID or an FHS-packet. For both packets, concerning our model, the contents of the packets is irrelevant within the inquiry phase[2]. In the inquiry sub state, the master sends out the first packet, which is an ID-packet, while the slave responds with an FHS-packet. We've abstracted away from their contents and replaced the packet types by constants.

3.4 The master

The master part is the most complex part of the device. The master is involved in complex timing of packet sending and listening. As described above, the task of the master can be described as sending an ID-packet over frequency X and Y rapidly, then immediately starting to listen if someone replies at frequency X and, if not, do the same for frequency Y. If it didn't get a response, it tries again on the next two frequencies. The frequencies are defined by the hopping sequence and change rapidly, trying a different frequency each time.

Although a slot in the inquiry phase is defined to be of a length twice as long as the clock tick period (so the slot length is 625 μs), the master performs two actions per slot. When the master sends messages, it is called a TX-slot (transmission), otherwise, it is a RX-slot (reception).

Figure 5 shows the part of the device model that is relevant for a master device. The first thing that is forced by the way the model is built, is that the master will wait until it starts the inquiry procedure until the least significant two bits of its native clock are zero. This is expressed by the guard `CLKN[0]==0 && CLKN[1]==0` on the urgent transition[3]. In a worst case scenario, this may take at most 3 clock ticks. The reason for this, is the need to make a choice when to initiate a TX-slot (at least, without having to record the clock value, which would be a lot of administration).

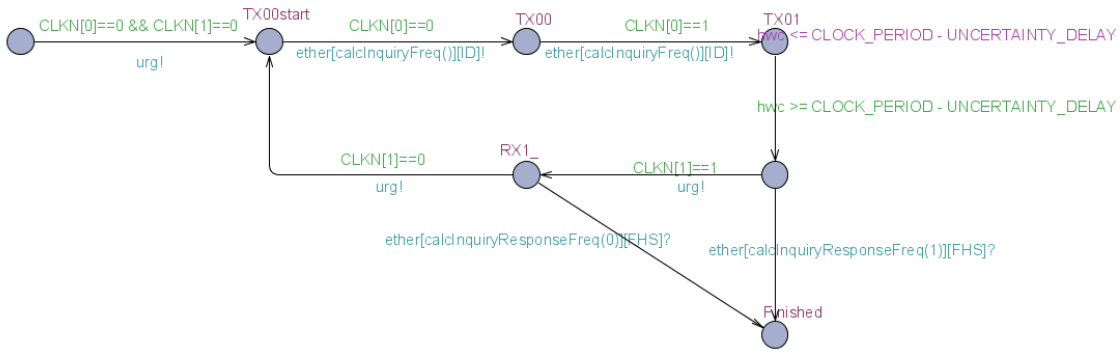

Figure 5: The device part relevant for master devices

After this urgent transaction, the first ID-packet is sent out urgently (see the discussion of the urgent `ether` channel in the previous section). These two transactions are always executed right at the same moment, because, once the least significant clock bit is 0, both guard on urgent transitions hold. Note

[2]The contents of the FHS-packet is used in the next phase to page the other device.

[3]The `urg!` synchronization channel is an urgent broadcast channel, introduced as a dummy for expressing that the transaction is urgent, nothing more. It has no side effects other that this.

the function call to `calcInquiryFreq()` here. This function calculates the hopping sequence, which results in a frequency, depending on the current value of CLKN, the native Bluetooth clock. Hopping sequences are detailed in Section 3.5.

Please note that, in contrast to what the specification implies, we have modeled the sending and receiving of packets to be instantaneously, i.e. sending does not take time and the packet is immediately done.

Next, we wait in the state TX00 (this naming convention should make clear that we are in a TX-slot, and the native clock's least significant bits are both zero). Then, again, immediately when the clock is incremented (i.e. when `CLKN[0] == 1`) we should send out another ID-packet. Because the hopping sequence calculation is executed again, and CLKN's value has changed, the frequency will be different here.

Once we are in the TX01 state we have to wait for the time slot to be finished in order to be allowed to act again. The master is now supposed to start listening for a response of a potential slave, on a corresponding frequency. However, since the clocks of both the slave and the master are subject to drift and jitter, we should start listening somewhat before the sending probably will take place.

Therefore, the Bluetooth specification introduces a so called *uncertainty delay*, which is a small amount of time before the clock tick arrives. The tricky aspects of this seemingly small exception are greater than one might expect at first glance. Not only because of the fact that we should manually use the hardware clock to time this event—which, from a conceptual point of view should not be necessary, ideally, because the Bluetooth clock is an abstraction made so we wouldn't have to use the hardware clock in the first place. Besides this being an ugly hack in our opinion, it has consequences for the way the listening frequency must be calculated, because this calculation depends on the current CLKN value. However, what we want is to use the CLKN value for the next time slot (only in this particular situation). This requires a hack from our side, which we solved by adding a boolean parameter to the `calcInquiryResponseFreq()` function call, stating whether we should increment the CLKN counter beforehand, or use it as it is.[4].

The rest of the master part is kind of straight forward. Nothing special happens in the remaining states or transitions. Once the TX01 state is left, we arrive at a nameless state, from which immediately is listened for a potential slave response. If that response is captured, we are finished. If not, we go to another state (RX1_, which represents the whole RX-slot where the least significant bits of CLKN are both 10 and 11) in which is listened for a response (on two frequencies, since the CLKN value changes during this state).

3.5 Hopping sequence calculation

Calculating the Bluetooth hopping frequencies mainly consists of calculating one large function. This function (the basic hop selection kernel) is described in a high level of detail in the Bluetooth specification. To represent this function in our model, we have implemented a series of UPPAAL functions that exactly executes the individual parts of the selection kernel. There are no differences between the calculation of the hopping sequences by our model and the calculation by the specification. Even the naming used in our model is nearly the same as the one used in the specification. So there is no need to focus on the details of the UPPAAL calculation here. Those can be found in the specification, or in the comments between the code of our model.

4 Assumptions

We have made some assumptions in our model. Some were already stated earlier. In this section we will list and discuss the assumptions made.

[4] The code is not getting any more elegant due to this.

- In figure 2.10 on page 81 of the specification: The master-to-slave slots always occur at `CLKN[1] = 0`. It is not stated that it should be this case, but it makes the model much more clear this way and it doesn't cause further problems because of the way the rest of out model works. The difference between the clocks of the devices might still be one, two, three etc. up to the maximum introduced.
- In our model, the slave sends a response-FHS-package and then assumes the package arrived at the master and is finished with this phase. Actually the FHS-package can be lost or disturbed. The Inquiry Scan Physical Channel specification doesn't say what to do in this case. Probably, the listening cycle has to start over for the slave, but the exact drill is unclear so far.

4.1 Hopping sequence calculation

As we stated in the previous chapter, the specification gives a detailed and almost complete description of the hopping sequence calculation. There is however one part where the specification is not entirely clear and seems to contradict itself. At page 88 section 2.6.2.4 the second addition of the selection kernel is explained. The specification says:

> *The addition operation only adds a constant to the output of the permutation operation.*

When we look at Figure 2.16 on page 86 of the specification, we can seen that the second addition has Y_2 as one of its inputs. This seems to be a contradiction, since Y_2 depends on the clock (Table 2.2 page 91) and is therefore variable instead of constant.

To solve this issue, we have ignored the above quote of the specification and have used the following calculation for the second addition:

$$(\text{InputStream} + Y_2 + E + F) \bmod 79$$

In which the input stream is the 5-bit result from the permutation operation.

Furthermore, Table 2.2 on page 91 states that for the inquiry phase $D = A_{18-10}$, we assumed this to mean $D_8 = A_{18}$ and $D_0 = A_{10}$.

There was one more issue we had to resolve while constructing the model. This has to do with the A-train and B-train switches. The trains themselves are described in much detail on page 92, 93 and 94. However when to switch between the trains cannot be found in this part of the specification. Some searching through the other parts resulted in (page 331 of volume 3):

> *A single train shall be repeated for at least $N_{inquiry} = 256$ times before a new train is used.*

When we assume that a train will start when `CLKN[3-0]=0`, then only `CLKN[3-0]` will change while processing a train, since a train only consists of 16 different frequencies. We have defined our `Xi` calculation in such a way that our assumption is correct. We can now switch between the trains on the switch of `CLKN[12]`, since at that point in time the train will have been used exactly 256 times. We could of course increase this number as the specification states, but this would cost more time in simulation and verification. Since the specification states (on the same page as the quote) that at least three switches between the trains must have taken place to be sure a connection has been made, there is no reason to assume the value of $N_{inquiry}$ is very large, because this would only increase the connection time.

5 Conceptual problems and Design decisions

We started with separate master and slave models (templates). But because the hopping-sequence-calculation should be in both, and it is more natural because a device isn't only a master or only a slave initially, we decided to make a one device model (template).

We only verified the model with 2 devices, one being master and the other slave. For the checking if the devices will synchronize on the hopping sequences we don't need more devices. We didn't check if there can be interference between more devices in the inquiry phase.

We had at some point a problem when the clock of the master and the slave would remain ticking exactly at the same time. In practice this will never occur, but in our model it was a possibility. When the slave receives the ID packet from the master at the first sending-try of the master in a slot, it waits for 625 μs. Then UPPAAL has to choose which of the two clocks to tick first. When it chooses the master-clock, everything goes alright. However when it chooses the slave-clock, the slave starts sending the response-FHS package before the master starts listening. The master in this way just misses the (beginning of) the FHS package. This problem is solved by implementing the uncertainty window which we had to do anyway.

5.1 Clock decisions

We used a Bluetooth-clock of only 17 bits instead of 28. Because for the hopping sequence calculation (and in our model) you only need bits 0 to 16, and this way the verification is much faster.

For the initial state the devices can only wait for the first 8 bits to be raised to 1 then it has to leave the initial state. The waiting is done for the two clocks (the master-clock and slave-clock) to differ. Actually we wanted it to be possible to let them differ for all the clock-bits. But this took too long to verify and we did not wait for the results.

5.2 Hopping sequence calculation

To represent the different global variables of a Bluetooth device, we had two possibilities. The first was to use integers and the second was to use bit-arrays. The advantage of numbers is that they might be processed a lot faster by UPPAAL. However, we are not entirely sure of this, because the actual state space remains the same as with bit arrays. So because of the fact that using bit arrays is easier to code and comprehend and much more intuitive to read, we decided to use the arrays. This has resulted in a model that looks a lot like the specification and has just one array-to-integer conversion function.

The specification of the inquiry and inquiry scan sub state does not explicitly state the bit length of the internal device variable *N*. So we have just assumed this to be 5 (the same as the length of the bit stream of the selection kernel). This might be incorrect, but that does not pose any threats to the correctness of the model. The reason is, that in any execution of the model, *N* can only be incremented by at most one. So *N* will alway be zero or one. Although this has not been a problem for us, if the model would be extended to a situation in which more than two packages will be transmitted, the length of *N* might become an issue.

6 UPPAAL problems

6.1 Documentation in a node

When we add some documentation in state `DoneScanningWaitingToSend`, UPPAAL gives some syntax errors: "Missing initial state" and "Missing system tag" We don't know why this is and it is only a problem in this state. It seems to be a bug in UPPAAL.

6.2 Urgent broadcast channels

At some point we wanted to leave a state as soon as a guard would hold. We first tried to make the state urgent, but this didn't work because it wanted to leave the state as soon as possible without waiting for the guard to hold, causing a deadlock. (Obviously, this could be expected since time is not allowed to proceed in an urgent state.) So we changed the state back to not urgent and introduced

an urgent broadcast channel. Now, it takes the transition as soon as the guard holds and doesn't have to synchronize because of the broadcast.

6.3 "Maybe satisfied"

In the UPPAAL language one can define boundaries to integers. This is quite useful in verifying the correctness of your code and most likely it is useful to prevent a state explosion. We had also implemented these boundaries and used them on a variable that stored the result of a modulo operation. We had expected this result to be positive, but in UPPAAL the modulo function appeared to be able to return negative values. Not aware of this flaw, we tried the code in the verifier. After some calculation, to our astonishment, UPPAAL resulted in the message: "Property maybe satisfied". Since the boundary error only rarely occurred, the simulator did not help much. So after some exhaustive searching we eventually found the flaw and were able to verify our properties correctly.

6.4 Clock guard on urgent transactions

As already mentioned in the detailed discussion on the slave implementation, UPPAAL does not allow for expressing guards on urgent transactions if the guard is a predicate over a UPPAAL clock. As we have tried to model the slave transaction from the `DoneScanningWaitingToSend` state to the state where the response is sent as an urgent transaction, UPPAAL complains about not being able to compile. This is depicted in Figure 6.

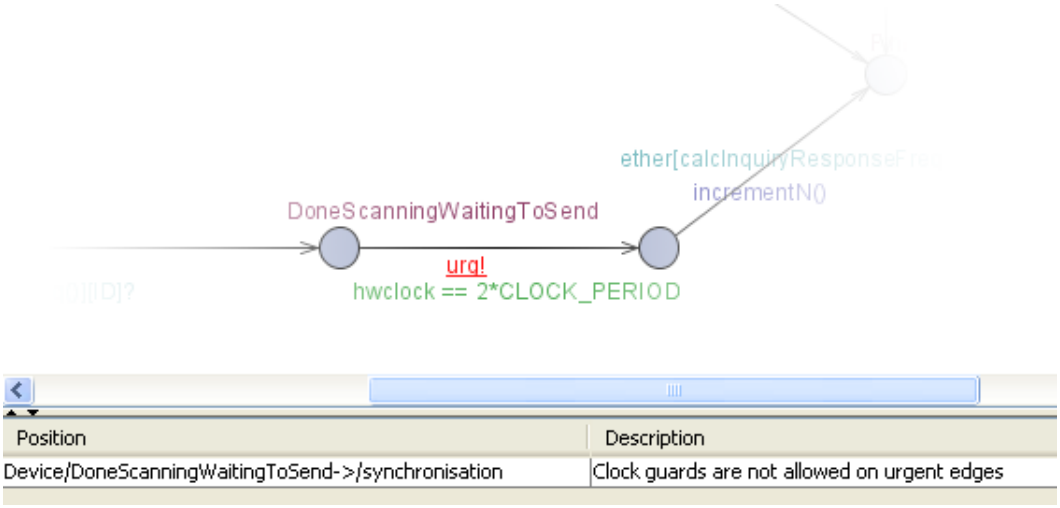

Figure 6: Errors in the UPPAAL model when adding clock guards to urgent transactions

To avoid this error message, we manually recoded the model to contain an invariant (`hwclock <= 2*CLOCK_PERIOD`) that should hold until the guard can finally be taken and does not hold after the guard does not hold anymore. Of course, this is not the preferred way of doing things.

6.5 Hopping sequence calculation

When having understood the selection kernel, implementing it in UPPAAL is not as easy as this might seem. UPPAAL has a very limited language syntactically as well as semantically. For functions that consist of just a few lines of code, the language is sufficient, but for a function such as the selection kernel, this requires some creativity and will certainly result in code that is not as beautiful as one would like it to be. I will not discuss all detailed restrictions of UPPAAL here. Those can be found in the manual. However, I will name the most problematic one:

UPPAAL can not handle variable sized arrays as parameter to its functions. In the selection kernel there are a lot of operations that are executed more than once with different bit stream sizes. For example the various addition operators. One would like to specify some kind of binary adder, and use this throughout the code. But this is impossible due to the restrictions of UPPAAL. We have solved this issue, by specifying a different function for every box of the selection kernel and copying some of the code. This results in code that is still usable and intuitive, but it certainly is not code that is easily maintainable.

7 Conclusions and Further research

We created a model of which we think it is sufficiently close to reality to be used in the verification of some properties of the Inquiry Response Phase.

In total, we have tried to prove two properties of the system, representing system liveness and safety. These are:

- $A\Diamond$ `Master.Finished` $\wedge$ `Slave.Finished`
  This property actually expresses that the system always eventually will reach the "finished" state for both devices, i.e. it expresses that there will always be communication. Actually, this property is the desired property the developers of Bluetooth would want to satisfy under all conditions. We have validated this important property for a whole variety of initial clock values. Besides that, we have been able to verify these properties, too, for both ideal Bluetooth clocks as well as clocks that were subject to drift and jitter.

  Also, the simplification of bringing down the CLKN array down to only 17-bits of length, as mentioned in Section 5.1, did not have effect on the validity of this property, as we expected beforehand. This makes us think the simplification is a harmless one. Verification was significantly faster after this simplification, which in turn accounted to make complex improvements easier to verify.

- $A\Box$ `not deadlock`
  This property actually expresses a system invariant, stating that the system as a whole will never deadlock. This property is very important in getting a confidence that the specification is correctly modeled.

Modeling the Inquiry Response Phase in UPPAAL worked pretty well. It gave us good insight in the phase and some questions surfaced that we could not answer easily. We even found a strange remark in the specification that to our opinion is incorrect as discussed in section 4.1.

Although it is arguable whether the specification is well written, at least we could, with some effort, all agree on what we think the specification specifies.

We implemented a jitter of at most $\frac{6}{625}$ and a drift of at most $\frac{1}{625}$ of the "real" time and some arbitrary initial clock difference of the first 8 bits of the Bluetooth clock. For this situation we could verify that always eventually the master device will receive a return packet from the slave. This means in practice that synchronization is accomplished. Probably we could verify more than this but it takes quite some time so that is left for the next group.

One of the things we did not look closer at of the Inquiry Response Phase but that might still be interesting is the duration of a transmission. Currently we assume that a transmission, if the receiver listens in time, will arrive completely and without errors or not arrive at all. Time does not elapse while sending or receiving. In reality this is not the case and it might be something to look closer at.

Something else we did not look at but probably will be relatively easy to do using the model is verify properties for more than two instances (master, slave, slave for example).

Our research focused on the Inquiry Response Phase, leaving out other phases. Obviously these could be interesting.

References

[1] Baseband Specification. In *Bluetooth–Core Specification v2.0 + EDR*, pages 55–210. 2004. URL http://bluetooth.com/NR/rdonlyres/1F6469BA-6AE7-42B6-B5A1-65148B9DB238/840/Core_v210_EDR.zip.