Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL

Henrik Ejersbo Jensen *Kim G. Larsen* *Arne Skou*

BRICS*, Aalborg University, Denmark, E-mail: {ejersbo,kgl,ask}@iesd.auc.dk.

Abstract

This paper compares the tools SPIN and UPPAAL by modelling and verifying a Collision Avoidance Protocol for an Ethernet–like medium. We find that SPIN is well suited for modelling the untimed aspects of the protocol processes and for expressing the relevant (untimed) properties. However, the modelling of the media becomes ackward due to the lack of broadcast communication in the PROMELA language. On the other hand we find it easy to model the timed aspects using the UPPAAL tool. Especially, the notion of *committed locations* supports the modelling of broadcast communication. However, the property language of UPPAAL lacks some expessivity for verification of bounded liveness properties, and we indicate how timed testing automata may be constructed for such properties, inspired by the (untimed) checking automata of SPIN.

1 Motivation

During the last few years, the SPIN tool [Hol91] has attracted much interest from university people teaching formal methods and from industrial developers. Its merits include a simple yet powerful design language based on asynchronous channels, as well as an expressive logic for property verification. However, until recently [TC96], it has not been possible to apply SPIN to verification of real time systems.

In this paper we compare the existing (untimed) SPIN with a recent real time tool, UPPAAL, which is based on timed automata specifications. This is done through application of the tools on a small protocol[1] for collision avoidance on an Ethernet-like broadcast medium. The untimed properties are verified on a PROMELA model and the real time properties are verified on a timed automata model.

*Basic Research in Computer Science, Centre of the Danish National Research Foundation.

[1]Our example is inspired by a recent paper by Karsisto and Valmari [KV96]

1.1 The Example

We assume that a number of stations are connected on an Ethernet-like medium, see Figure 1, that is, the basic protocol is of type CSMA/CD. On top of this basic protocol, we want to design a protocol without collisions[2], that is, we want to guarantee a lower bound on the transmission delay of a buffer - assuming that the medium does not loose or corrupt data and also assuming that the stations function properly[3]. The basic (obvious) idea of the protocol is to introduce a dedicated master station, which in turn asks the other stations if they want to transmit data to another station. However, the master has to take into account the possible buffer delays within the receiving stations. Hence, we want the protocol to enjoy the following properties:

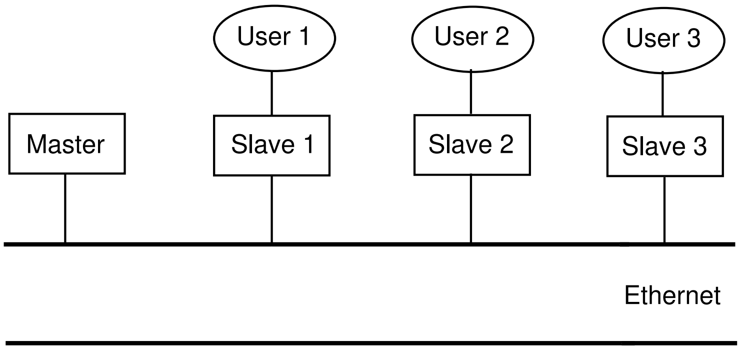

Figure 1: The Ethernet

- Collision cannot occur.
- The transmitted data eventually reach their destination.
- Data which are received, have been transmitted by a sender.
- Assuming error-free transmission, there is a known upper bound on the transmission delay.

Assuming that we know the buffer delays introduced by the medium and the slave stations, it should not be difficult to make an reasonable estimate of the upper bound by hand — assuming that the master makes enquiries according to a round-robin strategy. However, if we want to exploit the potential parallelism, it might very well be that one can find a more intrinsic strategy, which decreases the upper bound. Hence we would like to check for the following additional property:

- Does there exist a slave schedule with an upper bound being smaller than the sum of the individual slave delays?

[2]Applicable for e.g. real time plants.
[3]Well known classical protocols exist to handle these error cases.

1.2 Paper Organization

In section 2 we present a PROMELA model for the protocol, and we verify the above untimed properties. In section 3 we introduce UPPAAL and present the protocol model — applying the notion of committed locations to model broadcast. We verify the remaining timed properties and we indicate how one may extend UPPAAL with a more expressive logic than the existing one. Finally, in section 4, we sum up and compare the two tools on different aspects.

2 Modelling and Verification in SPIN

In this section we describe the modelling and verification of the Collision Avoidance Protocol using the design language PROMELA [Hol91] and the validation tool SPIN [Hol91, Hol]. We first present the PROMELA design and we then discuss the verification of this. The interesting verifications include verifying that no collisions occur and that data is delivered correctly between users.

2.1 The PROMELA design

Our basic modelling of the protocol consists of four slave processes and one master process. Master and slaves communicate using an independent process which models the communication medium (the Ethernet). The need for an independent process to model the medium is influenced by the need to model the mediums broadcast behaviour and the fact that we want the medium to be able to 'detect' collisions. The medium is modelled to be erroneous in the sense that it can loose messages, and to model the fact that slaves can loose messages independently, loss is modelled explicitly in the slaves. Finally the above model exists in a 'testing environment' consisting of one user process per slave. Slaves ask their user processes for data when they are enquired from the master, and in response user processes can either send data or indicate that they are not interested in sending anything. In this case the enquiry moves on to the next slave in round–robin fashion.

Message Format. All inter process communications are modelled by message passing on channels. Four different types of channels are used, each of zero capacity implying rendez-vous communication, see Figure 2. As we have used an independent process to model the medium, it is quite natural to model the communication between the master and the medium and between the slaves and the medium as synchronous. Messages from master and slaves to the medium are communicated on the channel `to_medium`, and messages from the medium are broadcasted using the channels `from_medium[N]` where `N` indicates the number of slaves plus one (the master). Messages between slaves and their users are communicated on channels `in[N]` and `out[N]`.

All messages passing the medium are formatted using three fields indicating the sender, the receiver and the type of data sent. The master's id is 0 and the slaves are numbered from 1 to `N`−1. Data types include `ENQ` indicating an enquiry from the master, and numbers in the range 0 to N−1 modelling data sent between users.

```
chan to_medium = [0] of {byte, byte, byte}
 /* channel to the medium : {sender, receiver, data} */

chan from_medium[N] = [0] of {byte, byte, byte}
/* channels from medium to each of master and slaves */

chan in[N] = [0] of {byte, byte}
/* channels between user and slave parts */

chan out[N] = [0] of {byte}
/* channels between user and slave parts */
```

Figure 2: Message Channel Formats

The Master. The master process, see Figure 3, passes enquiries round to slaves in round–robin fashion. Having sent an enquiry the master waits until it receives data and then it sends an enquiry to the next slave. If messages are lost the master will wait until 'silence' is detected and then it will send the next enquiry. Detecting silence is modelled by the `timeout` statement of SPIN which by default blocks until nothing in the full system is executable and then it becomes executable itself. Actually the master always wait for `timeout` before sending the next enquiry. This is because the silence of the system either is due to message loss as described above, or that a message has been correctly delivered at all slaves and at the master. In the later situation all processes will be listening for data i.e. no statements are executable besides `timeout`.

```
proctype master()
{
        byte sender;
        byte receiver;
        byte data;
        byte next;

        next=1;
        to_medium!0,next,ENQ;     /* enquiring the first slave */

        do
        :: from_medium[0]?sender,receiver,data
        :: timeout -> next=next%(N-1)+1;to_medium!0,next,ENQ
        od
}
```

Figure 3: The Master

The Medium. As mentioned we use an independent process to model the behaviour of the communication medium; the Ethernet. The fundamental property of the medium is that it broadcasts messages to all processes connected to the medium. In PROMELA there exists no broadcast primitive, so we model this explicitly using an approach where the medium, when receiving data from either the master or a slave, sends this data to all other processes connected to the medium in turn. Of course we would like this sequence of events to be atomic, but although PROMELA features an `atomic` primitive to ensure atomicity on a sequence of statements, this can not be used in our situation, as in general no blocking statements are allowed in `atomic` enclosings. The atomicity requirement says that no processes connected to the medium may start sending before the broadcast sequence of the medium is done. We model this by not allowing the medium to listen before the end of the broadcast. However, communications between users and slaves are allowed to interleave the broadcast.

The medium process itself can not loose messages (ignore them). This is because we want the possibility of the individual slaves to loose messages and therefore loss is modelled explicitly in the slaves.

The part of the `medium` code at the `accept_collision` label, see Figure 4, is for verification and will be discussed further in section 2.2.

The Slaves. The slave processes listens for messages at the medium, and whenever a message is communicated the slaves either ignores it (loss) or reads it. In the later case the slaves now determine what type of message is received and whether this message is addressed to them or not. If the message contains data (not enquiry) addressed to the slave, the slave sends the data along to its user. If the message is an enquiry from the master, the enquired slave asks its user whether it is interested in sending messages or not. In the former case the slave passes the messages on to the medium. See Figure 5.

The Users. The final part of the PROMELA design consists of the environment of the protocol, namely the user processes communicating with the slaves. The users can be seen as a sort of an testing environment for verifying the protocol, and further commenting will be given in section 2.2.

During the development of the above design we used the simulation facility of the validation tool SPIN to support early fault detection. Especially the message sequence chart (MSC) facility was very supportive in examining the communication pattern of our design.

2.2 Verification in SPIN

Using the simulation facilities of SPIN gives a first confidence in the correct behaviour of the design, as we can 'run' the protocol and see that there exists behaviours of our model that meet our expectations. Running the simulations we can for instance see, that there is a possibility that messages get lost and that the master in these situations sends on the enquiry to the next slave.

```
proctype medium()
{
        byte sender;
        byte receiver;
        byte data;
        byte i;

        do
        :: to_medium?sender,receiver,data ->

           i=0;
           if
           :: (1) -> do
                     :: i<=N-1 ->
                          if
                          :: i==sender -> skip
                          :: i!=sender -> from_medium[i]!sender,receiver,data
                          fi;
                          i=i+1
                     :: i>N-1 -> break
                     od
           :: to_medium?sender,receiver,data -> goto accept_collision
           fi
        od;

accept_collision: do
                  :: (1) -> skip
                  od                    /* collision detected */
}
```

Figure 4: The Medium

Fortunately, we can also see that messages (data) can be sent correctly from user to user.

The main aim of the designed protocol is to avoid collisions in the medium, so we want to verify this property. Furthermore, we want to verify that when data is sent between users – and not lost – then receiving users will actually get the data that is addressed to them. That is, messages can not get miss–directed in the medium.

Verifying that collisions are avoided is done by forcing the `medium` to enter an acceptance cycle if collisions occur, see Figure 4. The `medium` process will enter the acceptance cycle labelled `accept_collision` if `medium` can participate in two consecutive synchronizations on channel `to_medium`. By consecutive we mean with no broadcast delivery in between the receivings. Using SPIN to perform a full state space search including partial order reductions verifies that no acceptance cycles exist in the design.

Verifying that messages are correctly delivered to user processes is done using the testing environment consisting of the `user` processes. On request from their slaves users either indicate that they are not interested in sending data, or they send along data to their 'successor' which is simply interpreted as

```
proctype slave(byte id)
{
        byte sender;
        byte receiver;
        byte data;
        byte ny_receiver;
        byte ny_data;

        do
        :: from_medium[id]?sender,receiver,data ->
           if
           :: data!=ENQ ->
                if
                :: receiver==id -> out[id]!data
                :: receiver!=id -> skip
                fi
           :: data==ENQ && receiver==id ->
                in[id]?ny_receiver,ny_data;
                to_medium!id,ny_receiver,ny_data
           :: data==ENQ && receiver!=id -> skip
           fi
        :: from_medium[id]?sender,receiver,data         /* message loss */
        od
}
```

Figure 5: The Slaves

the user with process id one greater than the sending user. To guarantee that a correct unique receiving can be verified, the data send to the successor will be the id of the successor. Unique user id's are passed to user and slave processes on instantiation. Now, the `user` processes are forced into an acceptance cycle if they receive data from their slaves that do not correspond to the user id. See Figure 6. Using SPIN we verify that no acceptance cycles of the above nature exists.

3 Modelling and Verification in UPPAAL

The main purpose of the Collision Avoidance Protocol is to ensure an upper bound on the user communication delay in the Ethernet by avoiding collisions. In this above context it is obviously interesting to include *timing* in the design and to verify timing properties of the protocol. The PROMELA/SPIN framework does not yet allow the modelling of quantitative time.

In this section we describe the modelling and verification of the Collision Avoidance Protocol including *real–time* using the verification tool UPPAAL. We consider the process of transforming our PROMELA design to UPPAAL format. First, we introduce the UPPAAL tool and underlying model. Then, we

```
proctype user(byte id; chan cin,cout)
{
        byte data;

        do
        :: cout?data ->
           if
           :: data!=id -> goto accept_wrong_data
           :: data==id -> skip
           fi
        :: cin!0,0                              /* not interested */
        :: cin!(id%(N-1)+1),(id%(N-1)+1)
        od;

accept_wrong_data:

        do
        :: (1) -> skip                          /* wrong data received */
        od
}
```

Figure 6: The Users

consider the design of the timed protocol and finally we consider the verification of timing properties.

3.1 The UPPAAL tool

UPPAAL is a tool suite for automatic verification of safety and bounded liveness properties of real-time systems modeled as networks of timed automata extended with data variables [YPD94, LPY95a, BLL$^+$95], developed during the past two years. In this section, we summarize the main features of UPPAAL, applications to various case–studies and provide pointer to the theoretical foundation.

UPPAAL consists of a graphical user interface based on Autograph, that allows system descriptions to be defined graphically and a model-checker that combines *on-the-fly* verification with a *symbolic* technique reducing the verification problem to that of solving simple *constraint systems* [YPD94, LPY95a]. The current version of UPPAAL is able to check for invariant and reachability properties, in particular whether certain combinations of control-nodes of timed automata and constrains on variables are reachable from an initial configuration. Bounded liveness properties can be checked by reasoning about the system in the context of a testing automata. In order to facilitate debugging, the model-checker will report a *diagnostic trace* in case the verification procedure terminates with a negative answer [LPY95b].

The current version of UPPAAL is implemented in C++. An overview of UPPAAL is shown in Figure 7, and contains the following:

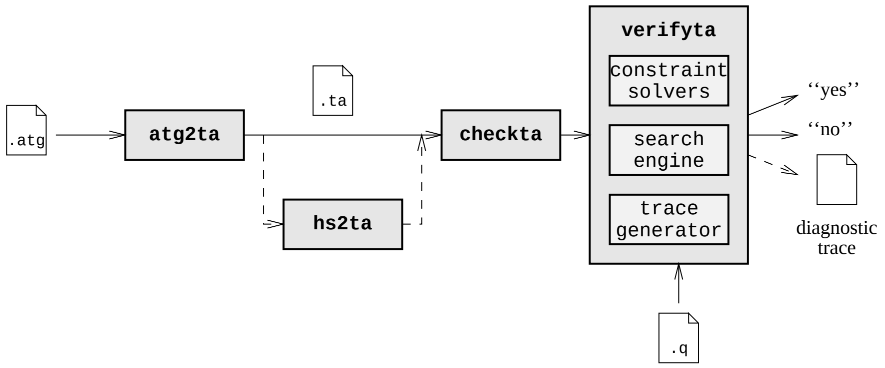

Figure 7: Overview of UPPAAL

`atg2ta` A compiler from the graphical representation (`.atg`) of a network of timed automata, to the textual representation in UPPAAL (`.ta`).

`hs2ta` A filter that automatically transforms linear hybrid automata where the speed of clocks is given by an interval into timed automata [OSY94], thus extending the class of systems that can be analyzed by UPPAAL.

`checkta` Given a textual representation (in the `.ta`-format) of a network of timed automata, `checkta` performs a number of simple but in practice useful syntactical checks.

`verifyta` A model-checker that combines *on-the-fly* verification with constraint solving techniques [YPD94, LPY95a].

During the past year, we have applied UPPAAL to a number of case-studies. To meet requirements arising from the case studies, the UPPAAL model and model–checker have been further extended with a number of new features:

Committed Locations. UPPAAL adopts hand-shaking synchronization between components in a network. A very recent case-study on the verification of Philips Audio Control Protocol with bus-collisions [BGK$^+$96] shows that we need to further extend the UPPAAL model with *committed locations* to model behaviors such as atomic broadcasting in real-time systems. The notion of committed locations is introduced in [BGK$^+$96]. Our experiences with UPPAAL show that the notion of committed locations implemented in UPPAAL is not only useful in modeling real-time systems but also yields significant reductions in time- and space-usages in verifying such systems.

Urgent Actions. In order to model progress properties UPPAAL uses a notion of *maximal delay* that requires discrete transitions to be taken within a certain time bound. However, in some examples, e.g. the Manufacturing Plant [DY95], synchronization on certain channels should happen immediately. For this reason the UPPAAL model was extended with *urgent channels*, on which processes should synchronize whenever possible [BLL$^+$95]. The notion of urgent channels (also known as *urgent actions* in the literature) has been implemented in both HyTech and Kronos.

Diagnostic Traces. Ideally, a model-checker should be able to report diagnostic information whenever the verification of a particular real-time system fails. UPPAAL reports such information by generating a *diagnostic trace* from the initial state to a state violating the property. The usefulness of this kind of information was shown during the debugging of an early version of Philips Audio-Control Protocol [LPY95b].

UPPAAL has been applied to a number of case-studies and benchmark examples, including: several versions of Fischers Protocol [AL93], two version of Philips Audio-Control Protocol [BPV94, LPY95b, BGK$^+$96], a Steam Generator [Abr95], a Train Gate Controller [HHWT95], a Manufacturing Plant [DY95], a Mine-Pump Controller [JBW$^+$96] and a Water Tank [OSY94].

The growing list of succesfully completed real–size verification case–studies and recently initiated collaboration with danish industry makes us believe that the UPPAAL is reaching a level of maturity where it can be applied to real industrial case–studies.

3.2 The UPPAAL design

Having already made the PROMELA protocol design actually made the modelling using timed automata a relatively easy task. The time spent on the timed automata design has been considerably less than the time spent on the initial PROMELA design.

The fundamental automata design is quite similar to the PROMELA design in the sense that the same type of processes are modelled. That is, each PROMELA process is matched by a timed automaton in the new design. The main differences in the models, besides the timing, considers the way communication between processes take part and the way the broadcasting behaviour of the medium is expressed. In the timed automata model used in UPPAAL there is no channel primitive and the only means of interaction between automata is by pure synchronization of atomic actions. Consequently, we use a combination of shared variables and synchronization to simulate the message passing that would actually take place in a real system.

The master. The master process is modelled as the timed automaton depicted in Figure 8[4]. We consider a lossy communication medium and therefore the master is equipped with a timer, see Figure 9, to guarantee that new enquiries will be sent in the precense of message loss.

The master starts by sending to the medium an enquire addressed to the first slave. This is modelled by the initial transition from `m0` to `m1`. The master sets the shared variable `data:=0` on this transition indicating that the message is an enquiry. Having sent the enquiry and without further delay, the master sets its timer and starts waiting until the message has been broadcasted to all

[4]This figure shows the actual input to UPPAAL.

slaves, indicated by an `empty` synchronization with the medium. This ensures that the master will not receive its own message.

Now, the master will either receive data broadcasted by a slave or it will timeout, if nothing is received within a certain time limit. In either case the master sends an enquiry along to the next slave indicated by increasing the shared variable `next`, setting `data:=0` and performing the output action `to_medium!`.

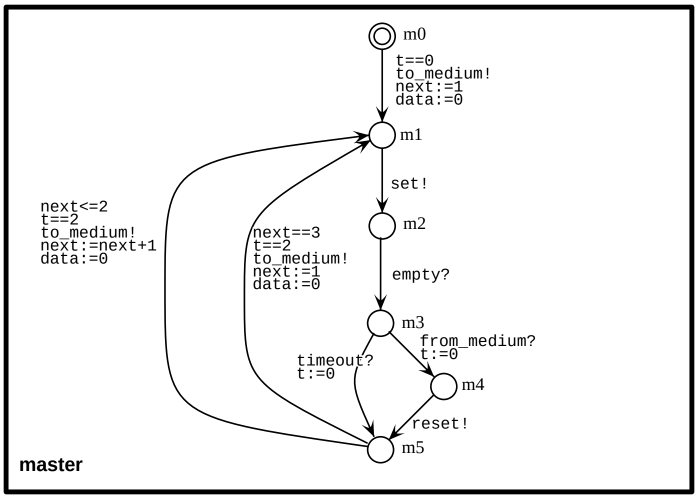

Figure 8: The Master with Timer

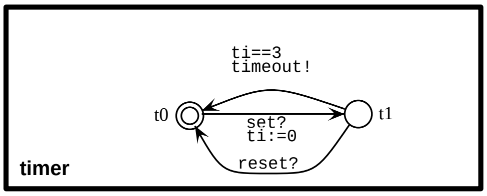

Figure 9: The Timer

The Medium. As mentioned the basic means of interaction between timed automata is by binary synchronization. No basic broadcasting primitive exists, but the notion of committed locations, see section 3.1, can be used to model the broadcasting in a simple way. Having received a message by synchronizing on the input action `to_medium?`, the medium delays the message for one time unit and then it starts broadcasting, see Figure 10. The broadcast consists of synchronizing in turn with each of the not–sending processes connected to the medium. Atomicity of the synchronization sequence is ensured by labelling each node that participates in the synchronization sequence as committed. This guarantees that no actions can interleave the broadcast.

The node labelled `col` will be entered upon collisions in the medium, and it serves the same verification purpose as the `accept_collision` state in the PROMELA model, see Figure 4.

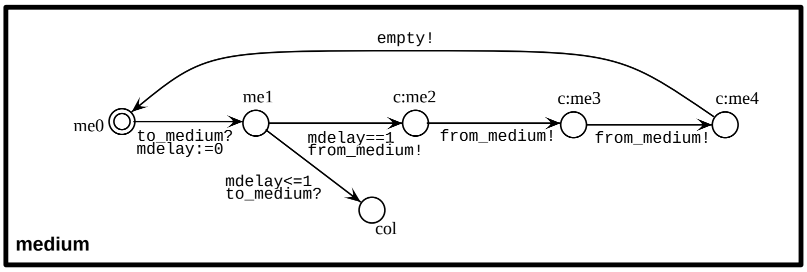

Figure 10: The Medium

The Slaves. In the UPPAAL model we need to model each slave as a unique timed automaton. In Figure 11 one of the almost identical slaves is depicted. The slaves synchronize with the medium on input action `from_medium?` and either they loose messages or they receive correctly, in which case they now determine what type of data is sent and to whom. Depending on the outcome, slaves either return to their initial state, sends data along to their users or asks their users for data to be send. In the last two situations the slaves will delay some amount of time and during this period they will not be able to detect messages sent to them. This is modelled as the 'ignoring' `from_medium?` input actions at the nodes `s1_2` and `s1_4` of Figure 11.

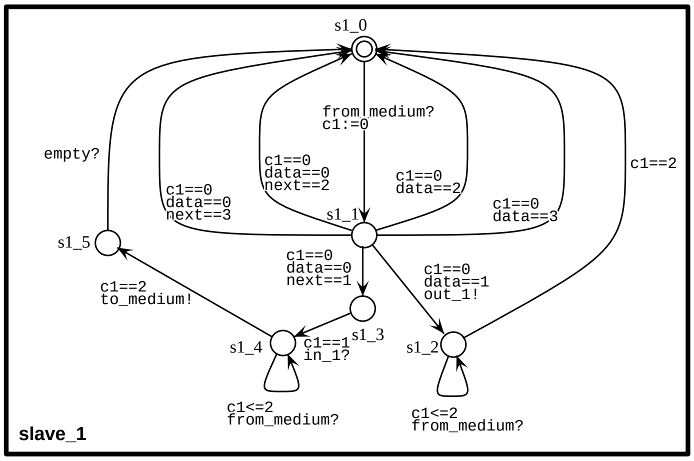

Figure 11: A Slave

The Users. As for the slaves we need to model each user process as a unique timed automaton. In Figure 12 the user automaton of the slave in Figure 11 is depicted. Users are always ready to either responding to enquiries from their slaves or receiving data sent from other users. Responding to enquiries is done by sending data to another user. The committed locations in the user automaton are for verification purposes and will be explained in section 3.3.

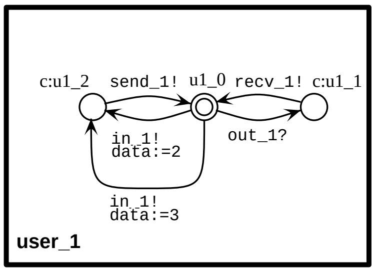

Figure 12: A User

3.3 Verification in UPPAAL

The primary correctness criteria that we want to verify for the protocol design explained in section 3.2 is that no collisions will ever occur. As the medium delays messages for one time unit, two messages sent to the medium within one time unit or less will eventually collide and this scenario will force the medium automaton in Figure 10 in the node `col`. What we need to verify is that it holds invariantly, that the protocol can not reach a state where the medium automaton is in state `col`. Stated as a property in the logic of UPPAAL this becomes:

$$\forall\Box(\texttt{not medium.col})$$

The satisfaction of the above formula is dependent upon the actual timeout limit in the timer. UPPAAL succesfully verifies the property if we consider a perfect medium, i.e. not lossy. But when an erroneous medium is introduced as in section 3.2 the timeout limit influences the possibility of collisions. If a timeout occurs to soon, the master interprets this as a situation where data is lost and all slaves are waiting for messages. But obviously this need not be the case as a slave can actually be in the process of enquiring its user. If this happens the slave will try to send data from its user and the master will try to send a new enquiry. If these two messages arrive at the medium within the one time unit delay of the medium, they will collide. We discover by repeated verification attempts that timeout limits greater than or equal to 3 will ensure that no collisions can occur. Also we verify that for a timeout limit of 2, a collision actually can occur, and the diagnostic trace facility of UPPAAL gives us a possible trace leading to collision.

Assuming a perfect medium (not lossy) and assuming that data is sent from users in round–robin fashion (all user are interested) we want to verify that the user–to–user communication delay is bounded by some constant. Also, we want to verify an upper bound on the delay between users sending data. This actually implies a bound on the delay between enquiries from the master, as all users are interested in sending. The above properties are examples of *bounded liveness* properties which can not be expressed directly in the logical property language of UPPAAL. To express the properties we introduce a seperate test automaton that probes the user processes in the protocol design. The test automaton will be designed to enter a 'bad node' if it tests an unwanted behaviour of the protocol. This approach is quite analogous to the never–claims used in the PROMELA language.

The test automaton for the properties described above is depicted in Figure 13. The automaton probes the sending and receiving of data in the user processes by synchronizing on actions `send_1` and `recv_1` (for user 1), see Figure 12. When a message is sent the test clock `s` is started and if the data sent is not received within a certain time limit, the test automaton is forced in the state `bad1`. Similarly, if a new sending is not performed within a certain time from the last receiving, the bad state `bad2` can be entered. The property verified using UPPAAL is:

$$\forall\Box\ \texttt{not}\ (\texttt{check\_1.bad1 or check\_1.bad2})$$

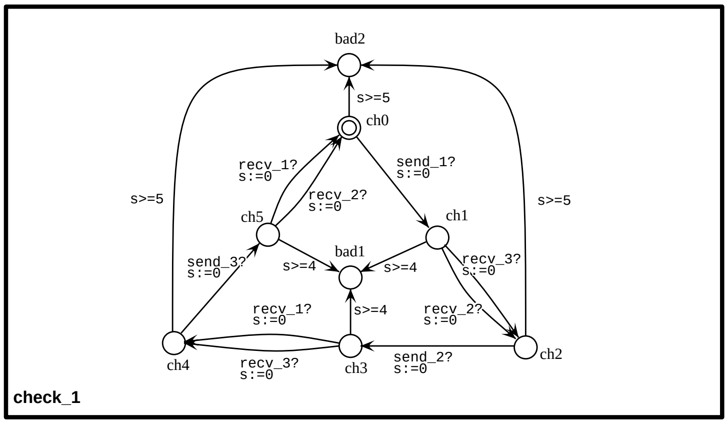

Figure 13: Bounded liveness test automaton

Using a similar approach as above we verify that there exists a round–trip time bound for the protocol. We use the test automaton of Figure 14 to verify that there exists a round–trip, modelled as user 1 having performed two sends, within a certain time bound. We verify:

$$\exists\Diamond(\texttt{check\_2.ch2 and s} \leq 18)$$

Also we verify that the following does not hold:

$$\exists\Diamond(\texttt{check\_2.ch2 and s} \leq 17)$$

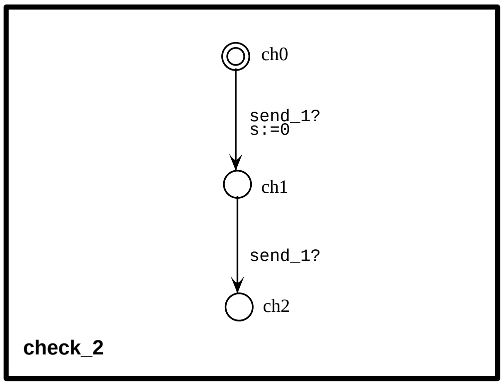

Figure 14: The round–trip time test automaton

That is, there exists no (initial) round–trip time of less than 18 time units.

As indicated on Figure 8 the master waits for two time units before sending out a new enquiry. This time limit guarantees that all slaves have finished their internal buissness and will be ready to receive data. I.e. messages will not be lost because of slave not ready to receive. To increase the round–trip performance of the protocol we could consider to let master send out new enquiries without delaying at all. Obviously this would require redesigning the enquiring strategy of the master s.t. it will not send enquiries to slaves having just received data from other users. We believe that a new strategy for the master benefiting from the parallelism in the slaves, will be possible, but we have not considered the actual design. However, we have verified that changing the waiting time in the master to zero, will allow for faster round–trip times. Obviously changing only the waiting time of the master does not guarantee that messages will not be lost, but it gives a clue as to how the enquiring strategy can be optimized.

3.4 Test Automata Generation

In the previous section we verified a number of bounded liveness properties by establishing reachability properties in the context of a testing automata. To allow the testing automata to 'observe' the system via communication, the system was in most cases extended with suitable probe actions.

Similar to SPIN's ability to generate never–claims directly from Linear Temporal Logic properties, it is possible to derive testing automata automatically from logical properties of the Safety and Bounded Liveness Logic introduced and studied in [LPY95a, LPY95b]. Here we indicate the automatic derivation of testing automata for a somewhat simpler logic STL (Simple Timed Logic) derived from the logic TML introduced in [HLY92].

The properties of STL is given by the following abstract syntax, where a ranges over actions and N over natural numbers (extended with ∞) :

$$\varphi \ ::= \ \mathrm{tt} \ \mid \ \mathrm{ff} \ \mid \ \mathrm{INV}\varphi \ \mid \ \langle a \rangle_{\leq n} \ \mid \ [a]_{\leq n}\varphi \ \mid \ \varphi_1 \wedge \varphi_2$$

The properties of STL are interpreted with respect to the behaviours (i.e. timed transition systems) of (networks of) timed automata.

The properties of STL are interpreted with respect to the behaviours (i.e. timed transition systems) of (networks of) timed automata. The interpretation of the propositional part is standard, and INVφ requires as expected that any reachable state of the timed automata must satisfy the property φ. The time–quantified action modality $\langle a\rangle_{\leq n}$ describes informally that the system must be able to perform an a–action no later than (the observer has experienced) a delay of n [5]. Similarly, $[a]_{\leq n}\varphi$ requires that any a–transition occurring before a delay of n must lead to a new state satisfying φ. We write $A_1 \mid \ldots \mid A_n \models \varphi$, when a network of timed automata, $A_1 \mid \ldots \mid A_n$ satisfies an STL formula φ.

Now, for any STL formula we may construct a testing timed automata T_φ with a designated location l_φ such that

$$l_\varphi \text{ is } un\text{reachable in } (T_\varphi \mid A_1 \mid \ldots \mid A_n)$$

if and only if

$$(A_1 \mid \ldots \mid A_n) \models \varphi$$

Thus model–checking STL properties may be reduced to deciding reachability questions. The testing automata T_φ is defined by the structure of φ and given in Figure 15.

In section 3.3 we essentially wanted to check that a sender can not send messages too frequently: at least 18 time units between two consecutive send's must elapse. The following property:

$$\text{INV}([send_1]_{\leq\infty}[send_1]_{\leq n}\text{ff}) \tag{1}$$

expresses precisely that at least n time units must elapse between two consecutive $send_1$ actions. Using the constructions described above we obtain the testing timed automata of Figure 16. That is we may check for the property (1) by checking for reachability of location l when the testing automata is combined with the system under consideration. The testing automata that was used in the actual verification was a slight simplification of the one that is obtained by the general construction.

4 Comparison of SPIN and UPPAAL

In this section we summarize our experiences with the two tools SPIN and UPPAAL based on our experience from the case study concerning the Collision Avoidance Protocol for an Ethernet.

Considering the design phase, the basic structure of the design was very easily obtained in the UPPAAL model because this was the last design made and the PROMELA/SPIN model was of great benefit as a structural basis for our UPPAAL model.

[5]The observer experiencing a delay of n means that the system may perform a sequence of delay steps intermixed with internal computations such that the delay steps accumulate to a total of n.

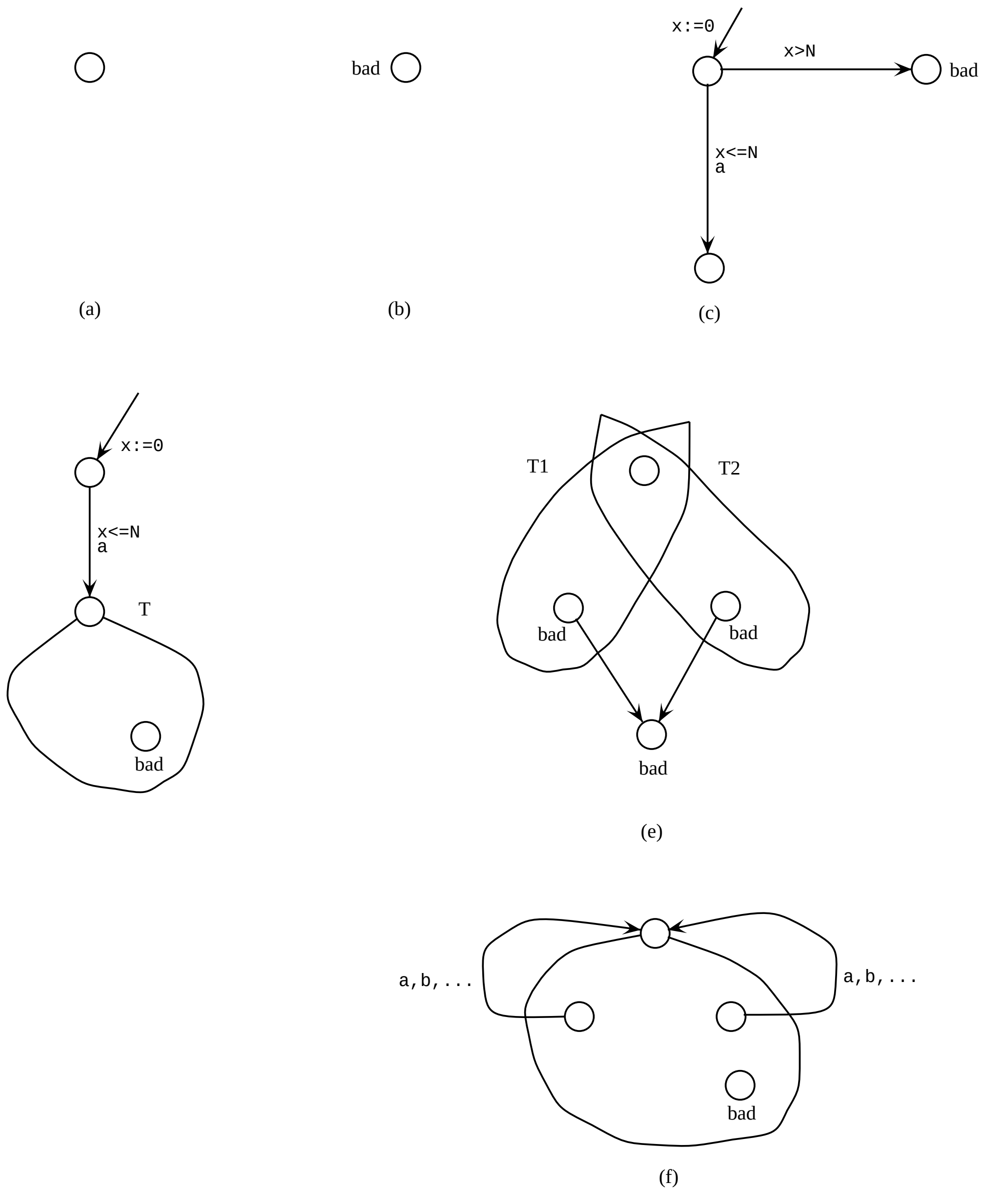

Figure 15: Generation of Testing Timed Automata: The respective test automata implements the following formulas: `(a)` tt, `(b)` ff, `(c)` $\langle a \rangle_{\leq n}$, `(d)` $[a]_{\leq n}\varphi$, `(e)` $\varphi_1 \wedge \varphi_2$ and `(f)` INVφ. In the figure, `T` indicates the testing automaton for φ. `T1` and `T2` indicates the automata for φ_1 and φ_2 respectively.

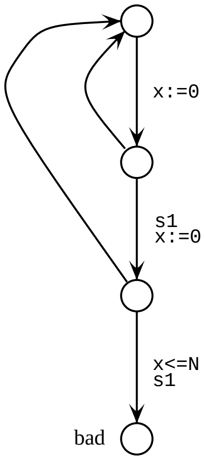

Figure 16: Test Automata for the property of (1)

In the PROMELA design phase we made extensive use of the simulation facilities of SPIN, especially the Message Sequence Charts. Within short time a 'running' prototype was designed and at an early stage faults were detected without having the full design at hand. In contrast UPPAAL does not yet allow for simulations and consequently, the UPPAAL design has to be more fully developed before the verification can be applied which delays the tool support in the design phase.

Considering the design languages, the obvious distinction is the possibility of modelling real–time systems in UPPAAL. In the case study it is shown that interesting bounded liveness properties can be expressed and verified in UPPAAL. Another beneficial feature of UPPAAL is the possibility of committed locations which makes possible a quite natural modelling of the broadcast behaviour needed in the case study. In contrast PROMELA can not apply the atomicity construct on sequences of send- and receive statements as these might be blocking.

Considering the verification phase, the kind of properties expressible in the property language of UPPAAL are restricted to invariance and possibility properties. Other properties as e.g. the bounded liveness properties of our case study needs to be expressed as separate test automata probing the design. In section 3.4 we present ideas on how to extend the property language and automatically generate the test automata. This is already possible in SPIN for transforming LTL properties to never automata.

The committed locations of UPPAAL make it possible to design non realizable systems. In particular systems that may enter completely blocked states (in the sense that neither actions nor time delays are possible) can be described. Obviously, we would like the possibility of checking whether the global design suffers such unrealizable properties or not.

Both SPIN and UPPAAL offers diagnostic information upon negative verification results. SPIN offers the possibility of examine an error scenario using the MSC's and UPPAAL offers a textual sample error trace leading to the unwanted state. By performing breadth first reachability analysis UPPAAL makes available a shortest error trace, whereas this is not guaranteed in SPIN as the reachability is performed depth first.

References

[Abr95] J.-R. Abrial. Steam-boiler control specification problem. *International Seminar on Methods for Semantics and Specification*, June 1995.

[AL93] Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. *Lecture Notes in Computer Science*, 600, 1993.

[BGK+96] 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. Accepted for presentation at the 8*th* Int. Conf. on Computer Aided Verification, 1996.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL— a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of the 4*th *DIMACS Workshop on Verification and Control of Hybrid Systems*, Lecture Notes in Computer Science, October 1995.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of FTRTFT'94*, volume 863 of *Lecture Notes in Computer Science*, 1994.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 66–75, December 1995.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A Users Guide to HYTECH. Technical report, Department of Computer Science, Cornell University, 1995.

[HLY92] U. Holmer, K.G. Larsen, and W. Yi. Decidability of bisimulation equivalence between regular timed processes. In *Proc. of CAV'91*, volume 575 of *Lecture Notes in Computer Science, Springer Verlag*, Berlin, 1992.

[Hol] Gerard J. Holzmann. *Basic Spin Manual.* AT&T Bell Laboratories, Murray Hill, New Jersey.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[JBW+96] Mathai Joseph, Alan Burns, Andy Welling, Krithi Ramamritham, Jozef Hooman, Steve Schneider, Zhiming Liu, and Henk Schepers. *Real-time Systems Specification, Verification and Analysis.* Prentice Hall, 1996.

[KV96] K. Karsisto and A. Valmari. Verification-driven development of a collision avoidance protocol for the ethernet. *FTRTFT96*, 1996.

[LPY95a] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87, December 1995.

[LPY95b] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of the 4*th *DIMACS Workshop on Verification and Control of Hybrid Systems*, Lecture Notes in Computer Science, October 1995.

[OSY94] A. Olivero, J. Sifakis, and S. Yovine. Using Abstractions for the Verification of Linear Hybrids Systems. In *Proc. of CAV'94*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[TC96] Stavros Tripakis and Costas Courcoubetis. Extending promela and spin for real-time. In *Tools and Algorithms for the Construction and Analysis of Systems, Second International Workshop, TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 329–348, 1996.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7*th *International Conference on Formal Description Techniques*, 1994.