@inproceedings{FHDGG1999,
 author = {Thomas Firley and
Michaela Huhn and Karsten Diethers
and Thomas Gehrke and
Ursula Goltz},
 title = {Timed Sequence
Diagrams and Tool-Based Analysis
-- A Case Study},
 booktitle = {The Second
International Conference on
The Unified Modeling Language,
Beyond the Standard (UML'99)},
 series = LNCS,
 volume = 1723,
 pages = {645--660},
 ISBN = {3-540-66712-1},
 url =
{http://www.cs.tu-bs.de/ips/firley/docs
/FHDGG1999-abstract.shtml},
 publisher = {Springer},
 year = 1999,
 month = oct,
 keywords = {UML, Sequence
Diagrams, Real-Time, Model
Checking,
Timed Automata,
Observer, Uppaal},
 language = {english},
 bibdate = {Wed Dec 1 12:04:17
MET 1999}
}

Timed Sequence Diagrams and Tool-Based Analysis – A Case Study

Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke*, and Ursula Goltz

Institut für Software, Abteilung Programmierung,
Technische Universität Braunschweig
{firley,huhn,diethers,gehrke,goltz}@ips.cs.tu-bs.de

Abstract. We use UML timed Sequence Diagrams to specify the real-time behaviour of a communication protocol of audio/video components. The Sequence Diagrams build the requirements specification against which an implementation of the protocol developed by the Bang & Olufsen company is proven correct.
To obtain a complete requirements specification, we have to mark the UML Sequence Diagrams as *optional* or *mandatory* behaviour. Then the Sequence Diagram interactions with their timing constraints and periods are transferred to a setting of timed automata. We use the UPPAAL tool for verification. In particular, we show that the implementation of the protocol conforms to the Sequence Diagram specification concerning the correct data transfer on the bus.

1 Introduction

Approaches like Realtime UML [Dou98], the ROOM method [SGW94], or the ACCORD approach [LGT98] aim to introduce the advantages of object-orientation, in particular reusability and evolutivity of the designs, into the development of real-time systems. Several of these real-time extensions are already supported by commercial tools like Rhapsody, ObjecTime (ROOM), and ObjectGeode (UML-RT) [Enc96].

However, the development of real-time systems remains a complex and error-prone task even if object-oriented concepts are used. Therefore one is interested in validation techniques which allow to analyse the behaviours of a design already in early phases. As a first step, nearly all tools provide some basic consistency checks for object-oriented designs: For instance, warnings indicate the states which cannot be entered or left.

Recently, also formal verification techniques have been considered for the validation of more complex requirements on the designs. In [LH99], a graphical interface for (a variant of) the ROOM model to the widely accepted SPIN model checker [HP96] is presented. In [AHP96] the timing consistency of Message Sequence Charts is investigated. A similar analysis for Sequence Diagrams is mentioned in [SvG98].

* This author was supported by the DFG project EREAS

Here we consider formal analysis on the basis of timed Sequence Diagrams, but take the timing consistency within *one* Sequence Diagram for granted. Our starting point is the common use of Sequence Diagrams as attachments to use cases, where they describe the dynamic aspects of certain scenarios [BjRJ99]. If in later design phases the behaviour of the system is modelled in more detail, one may ask whether the design is able to perform a sequence of interactions according to the Sequence Diagram specification. In that sense our approach can be regarded as a check on the timing consistency between *different* dynamic models used in the object-oriented design of real-time systems. From a slightly different viewpoint, we consider timed Sequence Diagrams as a requirements specification which should be satisfied by other dynamic models of the system that are closer to implementation.

To check if scenarios described in timed Sequence Diagrams can be performed by a system, we have to indicate which Sequence Diagrams describe *mandatory* or *optional* behaviour, and in which order the Sequence Diagrams shall occur. Harel and Damm extended Message Sequence Charts by a similar classification in [DH99].

To perform the formal analysis we transfer timed Sequence Diagrams as defined in [SvG98] to a timed automata setting [AD94]. Timed automata are well-suited for our purposes since the real-time constraints of the diagrams can be translated directly to clock conditions of timed automata. The transformation has to be directed by the designer who has to provide the connection between the different dynamic models: The designer has to associate the messages from the diagrams to transitions in the state-based models of the real-time design. Then the transformation to the timed automata setting can be done automatically.

We show the feasibility of our approach by investigating a medium size case study taken from a real-time protocol of the Bang & Olufsen company [HSLL97]. The interactions of the audio/video components with the single bus are modelled as timed Sequence Diagrams. Using the UPPAAL tool [LPW95,LPW97] we prove that an implementation of the protocol performs the data transfers correctly.

The paper is structured as follows: In Section 2 we briefly describe timed Sequence Diagrams. Section 3 is concerned with the transformation of Sequence Diagrams to timed automata. In Section 4 the verification of the case study is described. Section 5 concludes.

2 Real-Time extensions to Sequence Diagrams

2.1 Sequence Diagrams

UML contains several diagram types to model dynamic behaviour. The UML Interaction Diagrams can be attached to use cases to show the interactions of the system and some actors, or different subsystems or classes by indicating a sequence of messages which is exchanged between the participating instances. There are two different forms of Interaction Diagrams: The focus of Collaboration Diagrams is on the relationship between the instances participating in

a communication. To accentuate the temporal aspects of the communication process, Sequence Diagrams are used. They can either express one possible run (*instance form*) or they define all possible sequences by using loops and branches (*generic form*). In a Sequence Diagram the instances are horizontally arranged and represented by named rectangles. Below the rectangle of an instance a lifeline is attached. The flow of time is displayed in vertical direction. Arrows between the lifelines of two instances represent messages which are sent in the direction of the arrow. The stick arrow is deployed to specify a flat flow of control and concurrent objects. In the case study, messages are synchronous. The position of the instances in horizontal direction has no semantic relevance.

2.2 Syntax of Real-Time Extensions

To define timing constraints in standard UML, labels can be attached at the beginning and the end of a message arrow. The labels are interpreted as time stamps and can be used in timing constraints, e.g., to specify the minimum or maximum time gap between two marked points in the diagram or to define the duration of a periodic sequence. We will only deal with specific constraints (see Section 3.1). We adopt the notation of [SvG98] which extends the Sequence Diagrams of UML to express loops. In this notation, a sequence of messages which is repeated several times is surrounded by a rectangle. The loop condition can be placed at the top or at the bottom of the rectangle. We use the notation LOOP N TIMES $\{expr\}$. The constraint at the right side of LOOP defines the set of possible values for N. If the constraint is missing, N is an arbitrary natural number. To deal with different occurrences of a labelled event in loops, we introduce the following convention: Before a loop, a_{first} can be used in constraints to refer to the first occurrence time of an event with time stamp a in the loop. After a loop, a_{last} refers to the last occurrence of the tagged event in the loop. Within a loop, a_{next} denotes the time of the event occurrence in the following iteration. Figure 1 (a) shows examples of this notation.

3 Tool-based Verification

We aim to formally check whether a real-time design behaves according to a collection of timed Sequence Diagram scenarios. Therefore, we have to transfer the problem to an appropriate formalism which is able to handle real-time constraints. A further restriction is imposed by the fact that verification of practical examples is only feasible if tool support is available. Thus we selected timed automata as our design representation [AD94] and the UPPAAL tool [LPW97].

3.1 Timed Automata

Timed automata were introduced by Alur and Dill [AD94] for formal reasoning on real-time systems. They extend finite automata by real-time clocks and acceptance conditions. The transitions of a timed automaton may be labelled with actions, reset operations for clocks and guards containing timing constraints

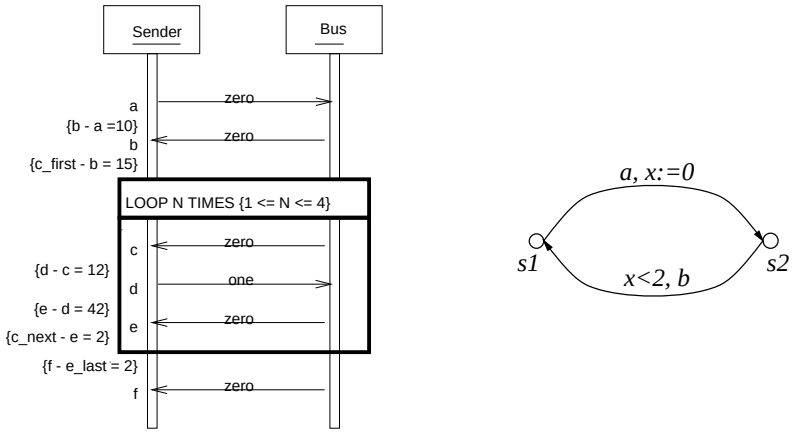

Fig. 1. (a) Loops and timing constraints (b) Example of a timed automaton

on clocks. However, to keep the verification problems decidable, the timing constraints and acceptance conditions are restricted to simple arithmetic expressions and comparisons: Only constraints of the form $c_1 \approx x \approx c_2$ and $c_1 \approx x - y \approx c_4$ are permitted where the c_is are non-negative constants or ∞ (infinity), x, y denote clocks, and $\approx$ denotes a comparison, i.e., $\approx \in \{=, \leq, <\}$.

Figure 1 (b) shows an example of a timed automaton. The automaton has two states s_1 and s_2 and it may perform the actions a and b toggling between the states. Additionally, the automaton has a clock x. A run of the automaton is an infinite sequence of timed transitions which obey to the timing constraints. Every time a is performed the clock is reset. The automaton may stay in both of the states and let time elapse. But it must leave s_2 within 2 time units after entering, because the transition back to s_1 may only be taken while the value of the clock is less than 2. If staying longer in s_2 the execution gets stuck, which is not a valid behaviour.

In this paper, we work with networks of the extended version of timed automata which are used in the UPPAAL-tool [LPW95,LPW97]. In addition to clocks, integer variables are available[1]. Communication is possible on shared memory (i.e., via global variables) or by output and input actions via channels. Input and output actions are synchronized by a synchronization function. Furthermore, a special kind of states is introduced. While a process may stay in a regular state while time is elapsing, in so-called *committed* states time must not elapse. Moreover, a process must leave a committed state in the next step of the system.

[1] The admissible constraints on integer variables are restricted in an analogous manner as those on clocks.

3.2 The Verification Process

In this paper, we consider timed Sequence Diagrams as requirements specification. We assume that the real-time design under consideration is already represented as a collection of UPPAAL timed automata.

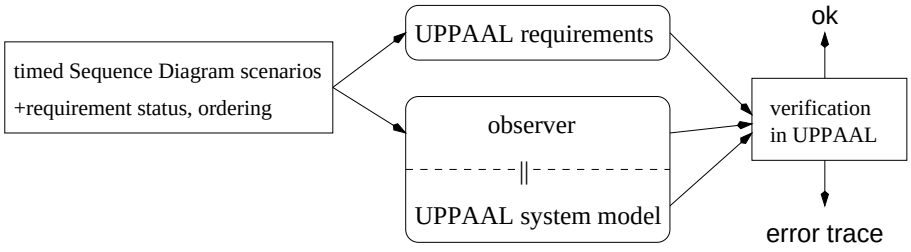

Fig. 2. The verification process

To be able to verify the requirements specified within the timed Sequence Diagrams the following steps are necessary.

- The instances addressed in the Sequence Diagrams have to be associated with components of the real-time design under development. In our setting, the instances will be mapped to sets of automata.
- The specification language offered by the UPPAAL-tool for the requirements only allows to express reachability properties like *in all reachable states some predicate is satisfied* or dually *a state satisfying some predicate is reachable*. Thus, to express complex behaviour like communication sequences as they occur in timed Sequence Diagrams, we have to construct an observer automaton. The observer monitors the system and is changing its state according to the Sequence Diagram scenarios. It will eventually reach a final state if the system conforms to the Sequence Diagram scenario. In case that the system violates the requirements, the observer will enter an error state.
  For the construction of the observer, the designer has to map the messages occurring in the Sequence Diagrams to actions the observer shall perform. Additionally, the designer has to declare the requirement status of a Sequence Diagram. Thus, it has to be indicated whether a family of Sequence Diagrams describes *mandatory* or *optional* behaviour. Also the order in which Sequence Diagrams shall be observed has to be given. Afterwards the observer and a proof obligation can be constructed automatically from the collection of timed Sequence Diagrams. The proof obligation will be that no error state can be reached and that it is possible to reach a final state.
- The system under development has to be slightly prepared to become observable by an observer. First the designer has to select a set of actions which shall correspond to messages which occur in the Sequence Diagrams, i.e., these actions are candidates for observation. Then the automata modelling the real-time design can be semi-automatically extended by our techniques to become observable.

3.3 Translation to UPPAAL timed automata

Instances. The real-time system consists of different components communicating among each other. These components do not necessarily reflect the instances whose communication we want to consider. Therefore we choose a partition $\mathcal{I}_1, \ldots, \mathcal{I}_m$ of the set of components, where each instance $\mathcal{I}_i$ corresponds to an instance in the Sequence Diagram. The internal communication within an instance will not be considered.

Observability of the system. In the general case of timed automata two communication actions a and b synchronize via a synchronization function that maps two actions to a new action. For instance a synchronization function f may map a and b to $c = f(a, b)$. The resulting action may be observed by an observer automaton. However, the verification tool UPPAAL imposes a restriction to this communication model. The restricted model uses *input* and *output actions* over certain *channels* to realize synchronous communication. For instance, $a!$ is an output action over the channel a and $a?$ is the complementary input action. Only complementary actions may communicate and the result is not visible for other automata. Hence we have to introduce additional observable actions which must not change the behaviour of the modelled system. The observer has to know which communication actions take place in the system and which are the participating instances $\mathcal{I}_i$.

We use the following approach to connect the observer to the system: After a reception on channel a the receiving instance B sends a new action $a\_receiver\_B!$. The observer is synchronized to this action by $a\_receiver\_B?$. Moreover, the sending instance A sets a fresh global variable $a\_sender := A$ to enable the observer to identify A as the source of the communication by the condition $a\_sender == A$. Figure 3 shows the modification of instance B to make the reception of a communication action a in the system model observable. The

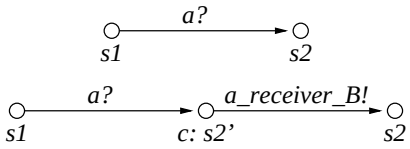

Fig. 3. Original and observable transition

new intermediate state $s2'$ is marked as committed. These committed states are an UPPAAL-specific extension which ensures that neither time elapses nor any other automaton performs actions while control is in such a state. Note that our construction only works if the target state of a communication action is not a committed state since otherwise the observer cannot do some necessary observation steps. However, in practical examples this does not pose severe problems. In case this situation occurs, more sophisticated preparation techniques have

to be applied, like pushing communications a transition further or doubling a state. We did not investigate automatization for these steps, because these cases occur rarely and cannot be handled uniformly. One has to make sure that the communication order and timings are not changed.

Formally we introduce the notion ACT for the set of communication actions of the system model which have to be observed and we derive the set ACT′ of actions and assignments which are added to the system to make the communication observable and the set $\overline{\text{ACT}}'$ containing the complementary actions and assignments which are used in the observer to identify a communication action of ACT in the system model and the corresponding sending and receiving instances.

$$\begin{aligned} \text{ACT}' := & \{\, a\_receiver\_\mathcal{I}! \mid \text{for all instances } \mathcal{I} \text{ and all actions } a? \in \text{ACT} \,\} \quad (1) \\ & \cup \{\, a\_sender := \mathcal{I} \mid \text{for all instances } \mathcal{I} \text{ and all actions } a! \in \text{ACT} \,\} \\ \overline{\text{ACT}}' := & \{\, (a\_sender == \mathcal{I}_1, a\_receiver\_\mathcal{I}_2?) \quad (2) \\ & \mid a\_sender := \mathcal{I}_1 \in \text{ACT}' \text{ and } a\_receiver\_\mathcal{I}_2! \in \text{ACT}' \,\} \end{aligned}$$

Now we have to define a connection between the observable communication pairs and the messages of the Sequence Diagram. Let M be the set of considered messages of the Sequence Diagram, consisting of triples $(n, \mathcal{I}_1, \mathcal{I}_2)$, where n is the name of a message, $\mathcal{I}_1$ is the sending instance and $\mathcal{I}_2$ is the receiving instance. Then the injective function m from M to $\overline{\text{ACT}}'$ is defined as

$$m(n, \mathcal{I}_1, \mathcal{I}_2) \;:=\; (a\_sender == \mathcal{I}_1, a\_receiver\_\mathcal{I}_2?) \quad \text{and} \quad n \sim a \qquad (3)$$

The designer has to associate message names n from the Sequence Diagrams to communication channels a of the system via the relation $\sim$.

An example of the relationship between messages and observer actions is shown in Figure 4. In the left part of the figure we see the message (a, A, B). The

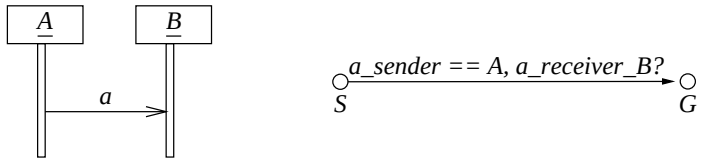

Fig. 4. Sequence Diagram and simple observer

message name a corresponds to the communication on channel a. Therefore ACT equals $\{a!, a?\}$. If we consider the two instances A and B, we get the set $\text{ACT}' = \{a\_sender := A, a\_sender := B, a\_receiver\_A!, a\_receiver\_B!\}$. The system is modified accordingly. The set $\overline{\text{ACT}}'$ which is important for the

observer construction is defined as

$$\{(a\_sender == A, a\_receiver\_A?),\ \ (a\_sender == A, a\_receiver\_B?),$$
$$(a\_sender == B, a\_receiver\_A?),\ \ (a\_sender == B, a\_receiver\_B?)\}$$

As the set M contains only (a, A, B) the basic part of the observer only reads the communication $(a\_sender == A, a\_receiver\_B?)$. The basic automaton is shown as the right part of Figure 4.

Construction of the Observer. The observer automaton is constructed according to the visual order of messages in the Sequence Diagram [AHP96]. For every instance $\mathcal{I}$ there is a local total order $\leq_{\mathcal{I}}$ over the input and output events which corresponds to the order in which the events are displayed. The relation

$$\leq_v := (\bigcup \leq_{\mathcal{I}}) \cup \{(e_o, e_i) |\ e_o \text{ and } e_i \text{ are output and input events of the same message}\} \tag{4}$$

is called the *visual order*, which is a partial order. For simplicity we will consider only Sequence Diagrams in which the messages are totally ordered. The general case can be treated by adding all interleavings accordingly.

First we construct an automaton which observes the desired behaviour but ignores the timing contraints. Suppose that there are N_0 messages $msg_i \in M, (1 \leq i \leq N_0)$ in the Sequence Diagram. Then the observer consists of a starting state $S = I_0$ and a successful final state $G = I_{N_0}$, the goal, and intermediate states I_i with $i \in \{1, \ldots, N_0 - 1\}$ between them. Transitions labelled with $m(msg_j)$ lead from I_{j-1} to I_j.

Then we have to add transitions to the automaton, which are used when observing a behaviour not according to the Sequence Diagram.

We introduce an error state F, which is entered if a behaviour is recognized that is not according to the Sequence Diagram specification. For each state I_j with $j \geq 1$ we calculate a set of transitions

$$T_j := \{\, t \in \overline{\text{ACT}}' \mid t \neq m(msg_j) \,\} \tag{5}$$

All transtions of T_j lead from I_j to F. Additionally we introduce transitions for any element of $\overline{\text{ACT}}'$ from F to itself. Once we observe a faulty behaviour the observer gets stuck in F.

The construction so far yields an automaton which enters G if the system has the same behaviour as described in the Sequence Diagram. The state F is entered if a wrong behaviour is observed. If the observer is in one of the intermediate states $I_1, \ldots, I_{n-1}$, an incomplete behaviour has been observed, which may be extended to a correct run.

Timing Constraints. We now add the timing constraints to the basic observer automaton. For each timing constraint we introduce a realtime clock $x_i, 1 \leq i \leq k$, where k is the number of timing constraints. Each timing constraint compares

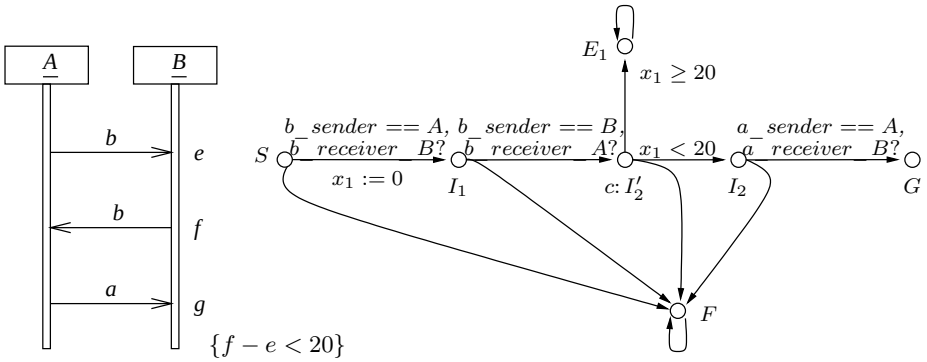

Fig. 5. Observer with timing constraints

the duration of a set of actions with a constant. In the construction we reset the clock to 0 if the first of these actions occurs. Directly after the final action concerned with the timing constraint, we introduce a new state in the observer. If the original transition of the basic observer, which should observe the final action, leads to I_j, then the inserted state is called $c\colon I'_j$. Thus it is a committed state where no time delay is allowed. We add two transitions with source $c\colon I'_j$. The first is labelled with the timing constraint and leads to the original state I_j. The second is labelled with the negation of the timing constraint and leads to the new error state E_i. This construction allows to state, that whenever the state E_i is entered, the timing constraint i is violated. If E_i cannot be entered the constraint is always fulfilled.

Figure 5 shows a simple example with three messages and the constraint that the delay between the first two messages must not exceed 20 time units. In the timed automaton we introduce the clock x_1, which is reset when the first message is observed. Immediately after the second message, we decide if the contraint is fulfilled and change the state accordingly. Note that we omitted all labels of transitions to F in Figure 5 for clarity reasons.

Loops. Observing loops is slightly more difficult, because we want to allow constraints concerning more than one cycle. The technique is exemplified in Figure 6. Suppose that four messages are in the Sequence Diagram. The message m_1 precedes the loop, the messages m_2 and m_3 are cycled within the loop and m_4 follows the loop.

In Figure 6, action s may set a counter for the loop. The loop condition l is checked after leaving the loop. Note that this is only possible if the actions m_2 and m_4 are different. If they are not different, the automaton is nondeterministic which poses a restriction to the requirements which may be checked, because the observer might choose the wrong transition and end up in an error state.

In order to distinguish between the action m_2 in the first interation and m_2 in any of the following iterations, we double this action. Constraints that refer to the

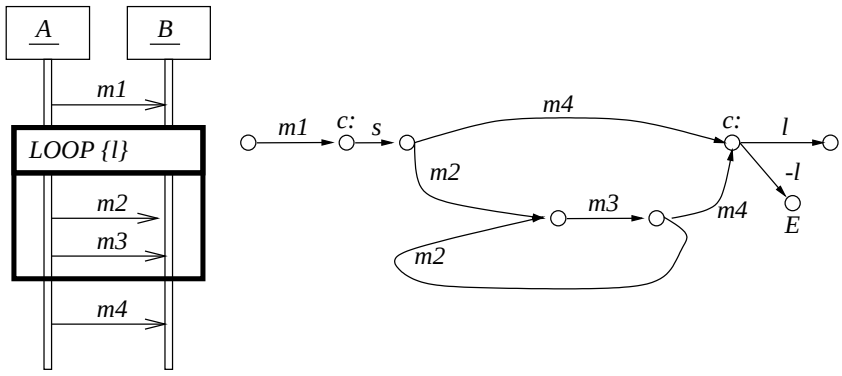

Fig. 6. Observer for a loop

first iteration cycle may be added to the upper m_2-transition, while constraints referring to one of the following iterations and the counter incrementation has to be added to the lower m_2-transition. Constraints that only refer to actions within one iteration cycle do not pose any problem. If a time stamp in a loop is indexed with *first* or *next* which is not the first message in the loop, the automaton has to split the first cycle and the following ones up to this transition as it is done here for the message m_2.

3.4 Verification

In addition to the observer we need proof obligations, expressed in a simple branching time temporal logic, which are verified automatically by the tool.

Besides the regular logical operators the UPPAAL temporal logic has got temporal operators and path quantifiers. The temporal operators, $\Box$ *(always)* and $\Diamond$ *(eventually)*, allow to refer to particular moments in an execution. The path quantifiers $\exists$ *(there exists a run)* and $\forall$ *(for all runs)* allow to refer to a specific run or to regard all possible executions at once. Only two combinations of these operators are allowed. $\forall\Box P$ means *in all reachable states predicate P holds* and $\exists\Diamond P$ means *a state satisfying predicate P is reachable.*

Sequence Diagrams show either an instancious view of a system or a generic one. We formulate three different kinds of interpretations for Sequence Diagrams. For each of them requirements and in two cases additional transitions in the observer are necessary.

Mandatory Behaviour. When the behaviour of a Sequence Diagram is regarded as *mandatory*, no other behaviour of the system is allowed than the behaviour given in the diagram. The following obligations describe this condition:

$$\forall\Box\neg F \tag{6}$$

$$\forall\Box\neg E_i \text{ for every timing constraint } i \tag{7}$$

$$\exists\Diamond G \tag{8}$$

These requirements consider the state F as an error state. Besides the timing constraints an exact correspondence to the communication structure is demanded.

Optional Behaviour. The behaviour described in a Sequence Diagram is *optional*, if it may happen at an arbitrary point in the run of the system. In this case, we have to ignore the behaviour not described in the Sequence Diagram. Therefore we add transitions from the starting state S to itself for all actions of $\overline{\text{ACT}}'$. Thus we get a nondeterministic automaton. We cannot guarantee that we will not enter an error state. The only requirement that we can formulate is:

$$\exists \Diamond G \tag{9}$$

If-Then Behaviour. If we require that whenever the behaviour of a first Sequence Diagram α has been observed, the system will immediately behave as described in a second diagram β, then the observers for both have to be connected by introducing a new transition from G_α to the start of β, S_β. The state G_α has to be marked as committed. The state S_α needs transitions to itself for all elements of $\overline{\text{ACT}}'$. The requirement α is optional, but whenever its behaviour has been observed, the requirement β gets mandatory. The obligations for this interpretation are

$$\forall \Box \neg F_\beta \tag{10}$$

$$\forall \Box \neg E_{i\beta} \text{ for every timing constraint } i \tag{11}$$

For the requirement α *is eventually observed followed by* β, we add the proof obligation

$$\exists \Diamond G_\beta. \tag{12}$$

4 Case Study: Protocol for Audio/Video Components

To illustrate our approach, we specify the audio/video protocol described in [HSLL97] by a set of Sequence Diagrams. The company Bang & Olufsen uses this protocol to define in which way the components of a stereo system may access a common bus to communicate. In particular, the protocol distinguishes the phases of initialization of the communication process, the transmitting of a data frame and the detection and handling of collisions. Due to the lack of space we only consider the transmission of messages here.

4.1 The Protocol

Frames. Communication between the components of a stereo system takes place by the transmission of data frames. Each frame consists of at least 17 so called T-messages. The protocol considers 5 different T-messages: A T_5-message marks the beginning of a frame and a T_4-message signals the end of a frame. Inbetween, a sequence of at least 15 messages (T_1-T_3) occurs which encode the information to transfer. After sending a frame, a component waits $50000\mu s$ before initializing the next communication.

T-messages. T-messages are subdivided in protocol periods. Two T-messages are separated by a *zero*-signal of one period. A message T_i, $1 \leq i \leq 5$, is encoded by a *one*-signal of $2 * i$ periods. Figure 7 (a) shows the Sequence Diagram of the transmission of a data frame. The diagram shows only the messages which change the state of the bus. All constraints refer to microseconds.

Periods. A protocol period has a duration of $1562\mu s$. Additional sampling of the bus is required at the beginning (S_1) and at the middle of a period (S_2) to detect collisions. For physical reasons, the sender has to put its message on the bus at W ($0\mu s < (W - S_1) < 600\mu s$) (see Figure 7 (b)). In the implementation, W is arbitrarily set to $40\mu s$ after S_1. The message $m2$ has the value *zero* in the first and the last iteration, otherwise the value *one*. If no collision occurs the values of $m1$ and $m3$ result from the previous setting of the bus at W. The Sequence Diagram in Figure 8 contains all messages which are exchanged during the transmission of a frame.

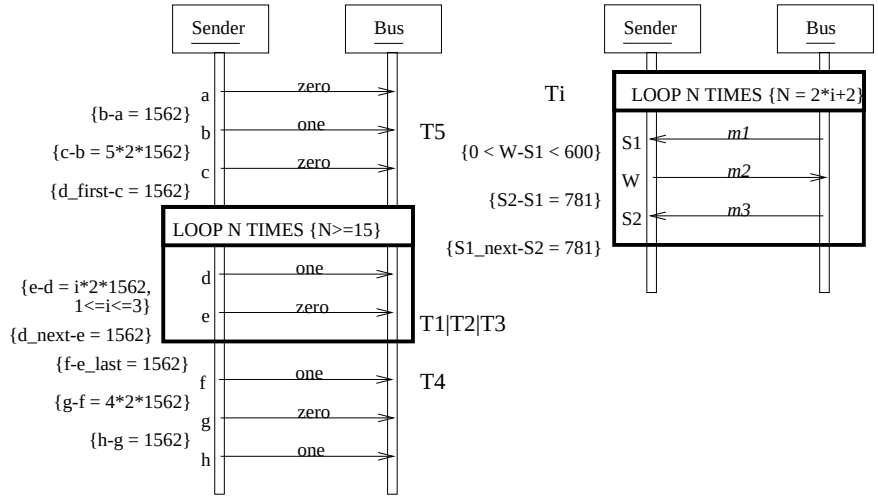

Fig. 7. (a) Transmission of a data frame (b) Bus sampling for collision detection

4.2 The Implementation

Our case study was already modelled as a set of UPPAAL timed automata in [HSLL97]. The system consists of several automata, three for each sending unit. In addition one automaton, modelling the global bus, is present. Only the sending is verified since the listening involves the same techniques. As the sending units are symmetric, it is sufficient to consider one of them. For a complete verification, however, a second sender for the modelling of collisions would be necessary.

Each sender consists of three automata, which communicate internally. The interesting communication is that with the bus. Reading is done through the

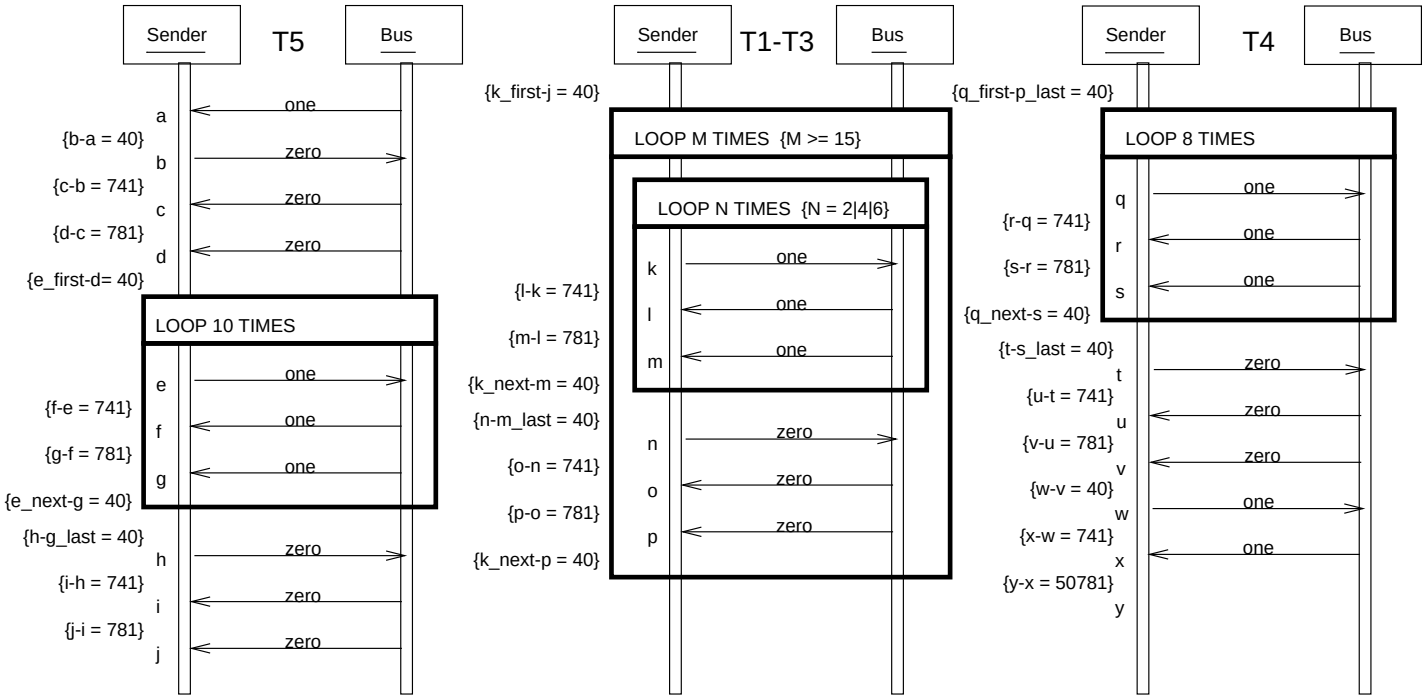

Fig. 8. Communication within one frame

channels *zero* and *one* and writing on the bus is done via variables which are in the scope of one sender and the bus. Every sender has got its own variables to communicate, e.g.,. instance A uses the variable Pn_A. The actions in ACT are thus $\{zero?, one?, Pn\_A := 0, Pn\_A := 1\}$. Since we only consider instances, one sender and the bus, we need not introduce variables to identify the participating instances of a communication. The following table shows the mapping from ACT to ACT′:

ACT	ACT′
$zero?$	$zero\_rec\_A!$
$one?$	$one\_rec\_A!$

ACT	ACT′
$Pn\_A := 0$	$zero\_send\_A!$
$Pn\_A := 1$	$one\_send\_A!$

Using the techniques described in Section 3.3, the system can be semi-automatically made observable.

4.3 The Observer Automaton

We build an observer which verifies that a whole frame according to the Sequence Diagram in Figure 8 can be transmitted. We interpret the behaviour as mandatory, since with one sender there must not be a collision on the bus.

The observer is built straightforward following the construction rules of section 3.3 and is shown in Figure 9. We had to introduce 29 clocks for the timing constraints. Each of them got an own error state. We also introduced 4 integer variables as counters for the loop iterations. Although it is possible to reuse clocks, we decided not to do that with regard to an automatic generation of the observer automaton.

We proved all the timing properties of the Sequence Diagram by verifying $\forall\Box\neg Observer.E_i$ for all i. We also proved that the first loop cycles exactly 8 times, and that the second loop (the inner one of the nesting) cycles either 2, 4, or 6 times. The requirement for this is $\forall\Box\neg Observer.EL_i$. However, we could not prove the number of cycles of the other two loops. The reason is that the observer has to decide nondeterministically how to proceed after the third loop. Choosing the wrong alternative yields a wrong number of cycles and a thus a deadlock in the corresponding error state. Furthermore we proved that the state G is reachable $\exists\Diamond Observer.G$ and that it is also reachable when the sender is in its final state $\exists\Diamond(Observer.G \wedge Sender\_A.stop)$.

The UPPAAL-tool is additionally looking for deadlocks. The only deadlocks found were those, in which the observer is either in state EL_3 or in state EL_4.

5 Conclusions

We investigated verification based on UML timed Sequence Diagrams. The Sequence Diagram scenarios were transferred to timed automata to allow for an automated check whether an implementation conforms to the real-time constraints specified in the diagrams. We validated the approach in a case study on a real-time protocol using the UPPAAL tool.

We have started to implement the transformation from Sequence Diagrams to UPPAAL timed automata to provide tool support for verification of timed

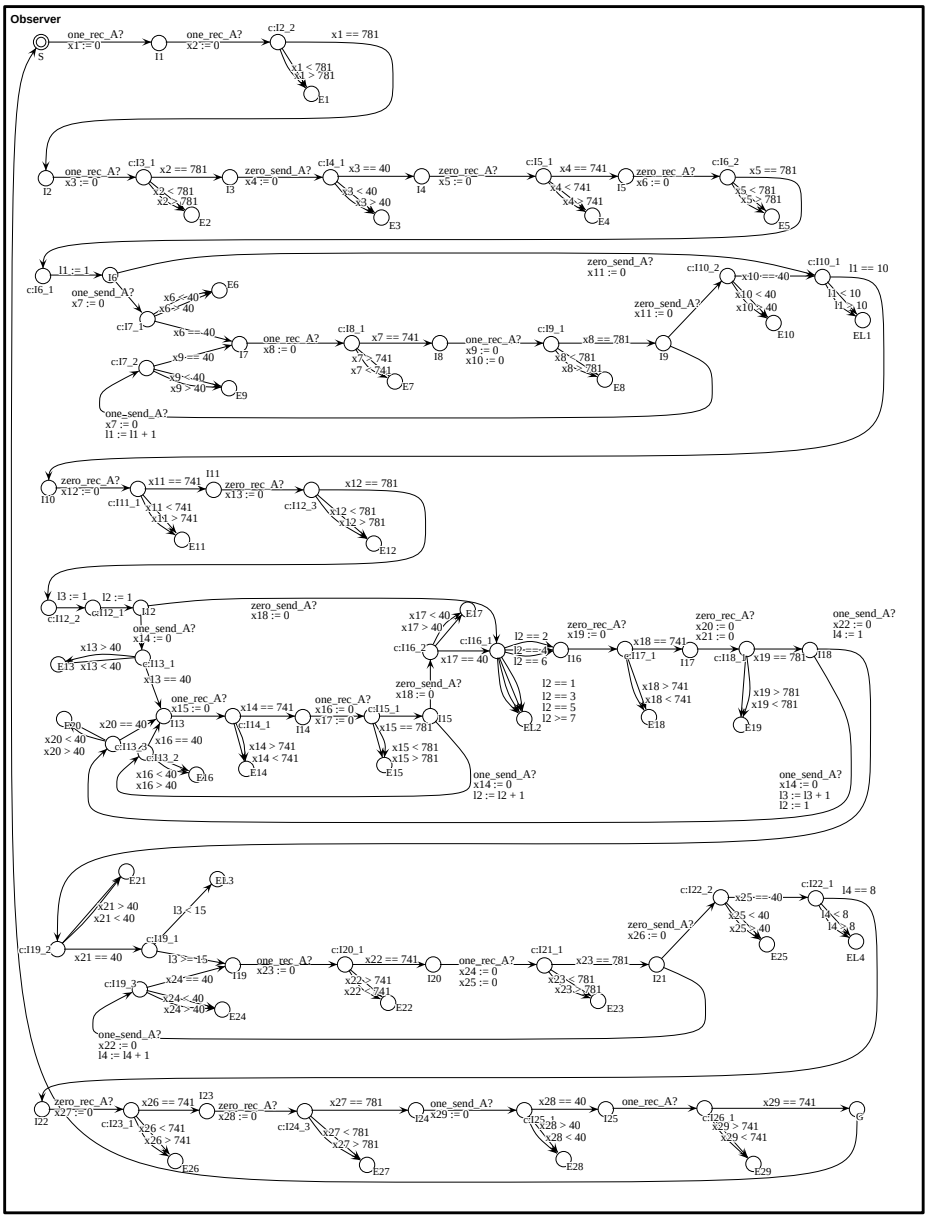

Fig. 9. Observer automaton

Sequence Diagrams requirements. In future work, we want to investigate the transformation of UML behavioural system models, in particular Statechart Diagrams containing timing constraints to timed automata. The aim is a formal analysis of the timing consistency between *different* dynamic models giving different views on a real-time design.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for Message Sequence Charts. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 35–48. Springer-Verlag, 1996.

[BjRJ99] Grady Booch, james Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[DH99] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. In *3rd IFIP Int. Conference on Formal Methods for Open Object-Based Distributed Systems, (FMOODS'99)*, pages 293–312. Kluwer Academic Publishers, 1999.

[Dou98] Bruce P. Douglass. *Real-Time UML.* Addison-Wesley, 1998.

[Enc96] Vincent Encontre. Modeling and implementing correct, scalable and efficient real-time applications with ObjectGEODE. *1rst Quarter Edition of Real-Time Magazine*, 1996.

[HP96] Gerard J. Holzmann and Doron Peled. The state of SPIN. In *8th International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 385–389, New Brunswick, NJ, USA, 1996. Springer Verlag.

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 2–13, 1997.

[LGT98] Agnès Lanusse, Sébastian Gérard, and Francois Terrier. Real-time modelling with UML: The ACCORD approach. In *UML '98*, volume 1618 of *LNCS*, pages 287–296. Springer-Verlag, 1998.

[LH99] Stefan Leue and Gerard Holzmann. v-Promela: A visual, object-oriented language for SPIN. In *Proc. of the 2nd IEEE Intern. Symp. on Object-Oriented Real-Time Distributed Computing.* IEEE Computer Society Press, 1999.

[LPW95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic model-checking for real-time systems. In *Proc. of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *LNCS*, pages 575–586. Springer-Verlag, 1995.

[LPW97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Intern. Journal on Software Tools for Technology Transfer*, 1(1+2), 1997.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[SvG98] J. Seemann and J. Wolff von Gudenberg. Extension of UML Sequence Diagrams for real-time systems. In *UML '98*, volume 1618 of *LNCS*, pages 225–233. Springer-Verlag, 1998.