Verification of UML-based real-time system designs by means of cTLA [*]

Günter Graw, Peter Herrmann, and Heiko Krumm
Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund
{graw|herrmann|krumm}@ls4.cs.uni-dortmund.de

Abstract

The Unified Modeling Language UML is well-suited for the design of real-time systems. In particular, the design of dynamic system behaviors is supported by interaction diagrams and statecharts. Real-time aspects of behaviors can be described by time constraints. The semantics of the UML, however, is non-formal. In order to enable formal design verification, we therefore propose to complement the UML based design by additional formal models which refine UML diagrams to precise formal models. We apply the formal specification technique cTLA which is based on L. Lamport's Temporal Logic of Actions TLA. In particular cTLA supports modular definitions of process types and the composition of systems from coupled process instances. Since process composition has superposition character, each process system has all of the relevant properties of its constituting processes. Therefore mostly small subsystems are sufficient for the verification of system properties and it is not necessary to use complete and complex formal system models. We present this approach by means of an example and also exemplify the formal verification of its hard real-time properties.

1. Introduction

Presently, the Unified Modeling Language UML becomes well established and is increasingly used for the specification of object-oriented application systems. It aims to a broad and practice-oriented design support and proposes a set of diagram types, each supporting the graphical description of certain system properties, aspects, and views [2]. Besides of the well-known class diagrams, use case diagrams document the purpose and utilization of systems. Object diagrams present the object instances of typical system configurations. Interactions between objects are described by collaboration diagrams or sequence charts. Statecharts or activity charts represent the behavior of objects. Moreover, package diagrams and component diagrams deal with architecture and deployment issues.

The UML does neither provide for binding formal semantics of single diagrams nor does it formally define the detailed relationships between diagrams and the resulting semantics of diagram combinations. Instead, the diagrams support intuitive interpretations and the meaning of diagram combinations depends to a certain extend on non-formal informations. Therefore, on the one hand, the UML lacks a concise formal modeling basis which would enforce unique interpretations and moreover could provide for formal verification and analysis means. On the other hand, the non-formal and in some sense open character of the UML semantics can be a strength with respect to practical application. The intuitive understanding based semantics is very flexible and thus enables various types of concessions to the special needs of practical projects. In particular, the set of UML diagrams of a project can concentrate on special and only partially interrelated views. It can reflect intermediate design stages and specify the system incompletely. Moreover, the UML is not limited to the description of structure and functionality. Additionally, descriptions of real-time properties are possible and contribute to the broad applicability of the UML.

For the design of critical systems, however, UML is not sufficient since essential system properties and functions should be specified formally in order to reduce the probability of misinterpretations and to enable the application of stringent formal analysis and verification techniques. With respect to this, it is of high interest, not to substitute the UML based design, but to complement it by formal methods. Then, the system as a whole can be developed by the well-established means of the UML whereas the more costly formal techniques can concentrate on critical parts and aspects of the system. Consequently, several approaches were developed already which propose formal semantics for the UML. Mainly these approaches focus on formal models for single UML diagrams. Mostly this is not sufficient for the formal verification of interesting system properties since very often only the combined semantics of a set of diagrams supplies the necessary prerequisites for a

[*]This work was funded by the German research foundation DFG.

proof. Let, e.g., two objects cooperate and perform critical operations. If a reasoning on possible histories of these operations is of interest, then the combined semantics of three diagrams is needed. The two statecharts describing the behavior of the two objects have to be set in context with an interaction diagram which defines the exchange of messages between the objects.

Our approach therefore not only supports a mapping from single UML diagrams to corresponding detailed formal models. Moreover, it supports the formal composition of diagram models to system models. Additionally, it is of importance that formal verifications can be based on submodels of a system since the combination of all formal diagram models mostly would result in a very large system model which would be too complex to admit formal verifications in practice. This calls for the use of a formal specification technique, the process composition operation of which has the character of superposition. Thus, a system as a whole inherits all relevant properties of its constituting processes and subsystems.

We use the specification technique cTLA which is based on L. Lamport's Temporal Logic of Actions TLA [14]. cTLA supports the modular definition of process types and the composition of process instances to systems [7]. A process may as well describe the behavior of a resource at all, as it may concentrate on partial aspects and thus corresponds to a behavior constraint. Basically, cTLA describes safety and liveness properties like TLA. Moreover, following the approach of [13], it has been extended for the description of real-time aspects and continuous system elements [8]. The so-called structured verification — which uses relatively small subsystems for the proof of system properties of cTLA — can also be applied for the verification of hard real-time properties [6]. Moreover, there is preceding work concerning the formal modeling of object-oriented systems. [3] reports on the systematic transformation of UML diagrams into cTLA specifications. [4] shows how this transformation can be applied for formal verifications of system properties and proofs of refinement correctness.

Related work mainly concerns the formal modeling of object-oriented systems and UML diagrams, the object-oriented description of real-time systems, the formal modeling of real-time systems, and the verification of real-time system designs. So, e.g., [15] presents formal models for UML diagrams. In particular, [12] models behavioral aspects of object-oriented systems by means of the formal technique DisCo which, like cTLA, uses state transition models, applies superposition, and connects components via joint actions. The Real-Time Object Oriented Modeling approach ROOM combines object-orientation with real-time aspects like latency and service times [19]. With respect to verification it relies on simulation since it is not based on formal models. Formal models of hard real-time systems have to represent combinations of real-time and functional properties. Mostly, timed state transition systems are used for that purpose. They extend functional state transition models by timing constraints (e.g., timed IO automata [18], hybrid automata [5], and timed statecharts [11]). In principle, these models support tool-assisted formal verifications. Mainly, model checking tools for the automated verification of finite state real-time systems (e.g., Uppaal [16], Hytech [1]) were proposed. In comparison with these, our approach favors creatively designed symbolic proofs which on the one hand need some development effort, but on the other hand can also be applied to systems with very large state spaces.

In the sequel, we firstly discuss the description of real-time systems in the UML. We slightly extend the corresponding timing constraint constructs of the UML for the representation of time intervals and refer to the timed semantics of statecharts as proposed by [11]. Thereafter we introduce an example and outline its UML based specification. After a short introduction into cTLA we report on the cTLA based formal model of the system. Finally, we outline parts of the formal design verification which is enabled by the cTLA specifications.

2. Describing real-time in the UML

The UML in its present form already supports the annotation of timing information [2]. So, timing events (e.g., *after* 2 msec) and change events (e.g., *when* 3:40) can be used to trigger state transitions in behavior descriptions of objects (i.e., in statecharts and activity charts). Moreover, interaction specifications (i.e., collaboration diagrams and sequence charts) can contain declarations of time mark labels which refer to the exchange of specific messages. Time marks can be used in constraints where the qualifiers '*.startTime*', '*.stopTime*', and '*.executionTime*' stand for the corresponding time points respectively duration period (e.g., a.*executionTime* $<$ 10 msec). Thus, more abstract interaction-oriented timing properties can be specified as well as more implementation-related waiting times for state transitions of objects.

In practice, one needs both types of timing information in order to support design by stepwise refinement. Consequently, the example, we introduce later on, will contain both types. It will provide for two system views, one abstract view modeling a simplified system structure, and one detailed view which corresponds to a refined, more implementation-oriented system. The first view serves as an abstract specification stating the requirements to the system. The second view represents a refined design step and shows how the requirements will be tackled by system internal functions. The provision of specifications with different levels of abstractions can document the process of a refining design. Moreover, it is of high interest for formal verification since it enables proofs of correct refinements. In particular, we like to refer to the notion of correct refinement of TLA which ensures that a refined system in fact will have all relevant properties of the preceding more abstract specification [14]. While other verification approaches mainly check one design step against some criteria in a relatively isolated way, a series of refinement proofs can verify the design process as a whole.

In order to support refinement proofs, we had slightly to extend the time notions of the UML. In fact, it is nearly impossible to implement exact non-interval waiting time constraints like 'the transition has to be triggered in exactly 11 msec after its enabling'. Therefore, we will use time intervals instead of exact periods. In interaction diagrams we will restrict the use of timing constraints correspondingly and the constraint type *'every'* will be parametrized by time intervals constraining the time period between each two consecutive events in a sequence. In statecharts we will also use time interval parameters for time events in order to state minimal and maximal waiting times for state transitions.

Unlike the semantics of interaction diagrams, the semantics of statecharts is relatively complicated but nevertheless only non-formally described in the UML. To prepare stringent reasoning we therefore need a more precise understanding of statecharts and timed state transitions. It can be supplied by reference to existing approaches. In particular, the approach of Timed Statecharts [11] is very near to our needs. With respect to the modeling of time consumption it provides a helpful separation of concerns. It makes a distinction between immediate transitions and waiting transitions. Immediate transitions are triggered by inputs, but abstract from time consumption at all. Whenever an immediate transition is enabled, it must be executed before time can proceed. Thus, immediate transitions serve for the purpose of timeless modeling of functional behavior. Time consumption, i.e., progress of time, is possible only when no immediate transition is enabled. Then the object is waiting in its current state and time will proceed till a waiting timed transition will fire. Timed transitions do not depend on inputs. Therefore, they focus on the modeling of time consumption. Their time parameters specify time intervals realistically where a timed transition is labeled by two duration values, a minimal and a maximal waiting time. It has to wait at least for the minimal waiting time, and if no other transition changes the current state, it has to fire before the maximal waiting time is exceeded.

While these elements of Timed Statecharts fit well with our requirements, there are modifications which are necessary to adapt it to the statecharts of the UML. One major difference concerns the handling of inputs. In original Timed Statecharts inputs are stored in a flag-register like way. All pending inputs are visible and each of them can trigger a series of immediate transitions till the flag-register is reset (It is reset, when time proceeds). Other than this mechanism, each UML object provides for a queue which stores incoming messages in a FIFO manner. Here, only the front element of the queue can trigger the next transition and the transition implicitly removes the element. When the front element does not match with a transition of the current state, it can explicitly be deferred. Otherwise, it will also be removed implicitly. Since the FIFO handling of incoming messages is a basic feature of UML objects, we modify Timed Statecharts accordingly. We keep, however, the separation between functionality and time consumption. We also follow up the principle, that time can proceed only, while no immediate transition is enabled.

Other essential differences between UML statecharts and Timed Statecharts concern the action sequences which can occur as labels of UML statechart transitions. With respect to this, we assume, that transitions are at most labeled by one action. When the transition fires, the action is executed atomically without any time consumption. Therefore we resolve action sequences and replace each sequence labeled transition by a series of one-action transitions and additional intermediate states.

3. Example system

Our example system is a part of a distributed multimedia system. It provides the transfer of a video frame stream from a source to a remote display. The used frame transmission service is of varying quality. Therefore hard real-time mechanisms are applied in order to adapt the system to the current quality of the frame transmission. As often proposed in this field of application (e.g., according to the Open Distributed Processing Reference Model ODP [10] and as realized in corresponding multimedia middleware systems [20]), we connect source and sink by explicit binding objects and use cooperating filters for the quality of service adaption. From the ODP model, moreover, we adopt two levels of abstraction. Corresponding to ODP's computational view, we firstly abstract from distribution and transmission. Our refined system then enters into ODP's engineering view and explicitly models the use of a telecommunication network and the adaption to its current quality of service. The interesting quality parameters of the example are the frame rates and jitter values of source and sink. During frame transfer in the network variable delays can occur. Increasing frame delays indicate overload and therefore shall cause a reduction of the sending frame rate. The example and its formal modeling are furthermore related to [17] who also propose a TLA based formal representation of quality of service constraints. Like our approach,

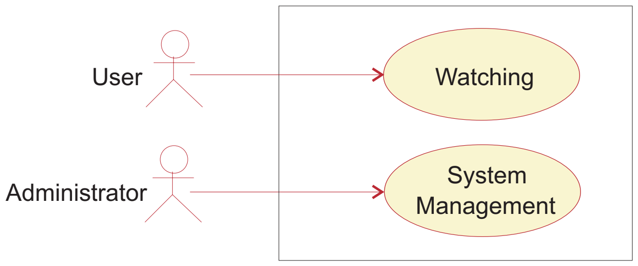

Figure 1. Example system use case diagram

they use formal composition of behavior constraints, however, cannot profit from superposition properties.

Fig. 1 shows the use cases of the example system. A regular user wants to watch the video frame stream produced by the remote camera. Furthermore, an administrator can perform operations in order to manage the system. In the sequel, we will concentrate on the watching use case and do not go into details of management.

Consequently, the class diagram (cf. Fig. 2) concentrates on the attributes and operations which are relevant for the watching functionality. A source object produces a frame stream with a certain frame rate *frameRate* under a certain *jitter*. It periodically calls the operation *consumeFrame* of a connected medium or sink. The environment can influence the frame rate by means of the operation *setFrameRate*. A sink is characterized by its minimal and maximal frame rate and the maximally tolerable jitter. It receives frames by executing the operation *consumeFrame*. Moreover, the operation *changeSinkFrameRate* can be called in order to signal the current frame rate to the sink. A medium is a combination of a source and a sink. In the abstract view, the medium subclass *AbstractBinding* will be used, which later on will be refined to a composition of three medium objects of the classes *SourceFilter*, *NetworkBinding*, and *RenderingFilter*.

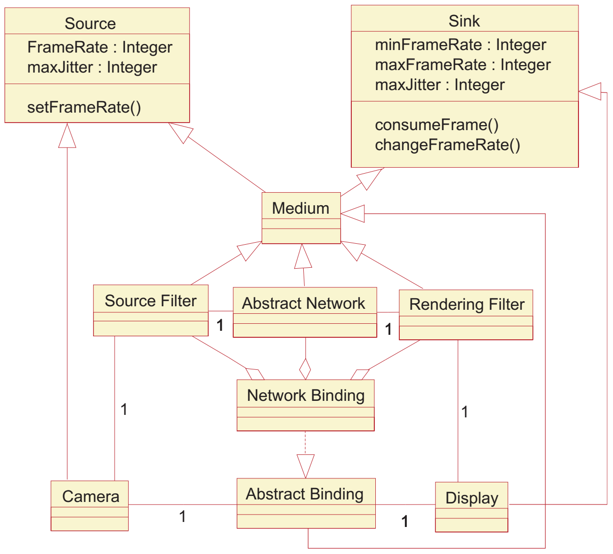

Figure 2. Example system class diagram

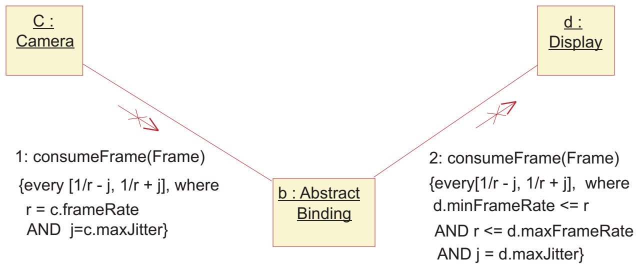

Figure 3. Abstract watching system collaboration diagram

The more abstract model *'Abstract Watching System AWS'* of the example is presented by means of the collaboration diagram in Fig. 3. It shows an association of three object instances. The camera *c* and the display *d* are connected by an abstract binding object *b*. The camera periodically calls the *consumeFrame* operation of the abstract binding object which in turn calls the *consumeFrame* operation of the display. Both operation call sequences underly timing constraints which are documented in the diagram. They constrain the period between each two consecutive operation calls. As introduced in Sec. 2 the constraints refer to corresponding time intervals.

Moreover, we want to model, that the abstract binding transfers the frames reliably without loss, corruption, and reordering. Furthermore, we want to express, that the transfer delay is limited by a constant *maxDelay*. For that purpose, we add a statechart for the behavior of the abstract binding which is shown in Fig. 4. In the main, the statechart model uses a FIFO queue component in order to show that a series of frames may be under reliable transfer at the same time. With respect to the delay time constraint representation, the model queue attaches a time stamp of latest possible delivery to each frame which is computed from the enqueuing current time *time* under addition of the duration constant *timeMaxDelay*. While all other transitions are im-

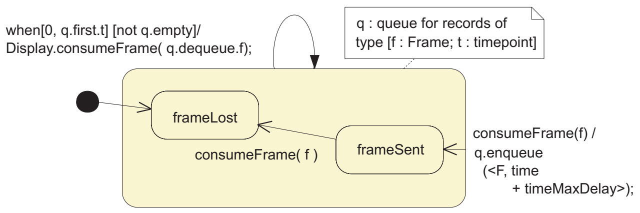

Figure 4. Abstract binding statechart

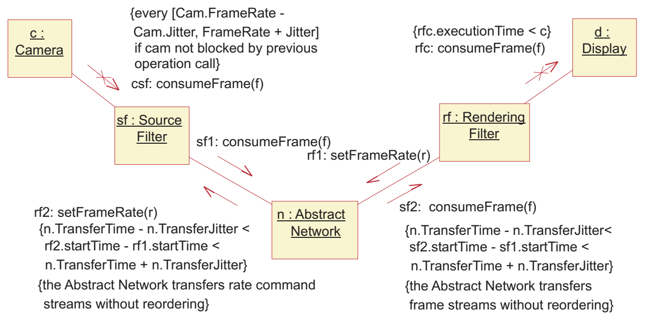
c : Camera
{every [Cam.FrameRate - Cam.Jitter, FrameRate + Jitter] if cam not blocked by previous operation call}
csf: consumeFrame(f)
{rfc.executionTime < c}
rfc: consumeFrame(f)
d : Display
sf : Source Filter
rf : Rendering Filter
sf1: consumeFrame(f)
rf1: setFrameRate(r)
n : Abstract Network
rf2: setFrameRate(r)
{n.TransferTime - n.TransferJitter < rf2.startTime - rf1.startTime < n.TransferTime + n.TransferJitter}
{the Abstract Network transfers rate command streams without reordering}
sf2: consumeFrame(f)
{n.TransferTime - n.TransferJitter< sf2.startTime - sf1.startTime < n.TransferTime + n.TransferJitter}
{the Abstract Network transfers frame streams without reordering}

Figure 5. Detailed watching system collaboration diagram

mediate, the dequeuing and delivering transition depends on a timed change event. The transition has to fire within this time period which starts when a frame becomes the front of the queue and ends when the current time matches the time stamp of the last delivery.

The more detailed model *'Detailed Watching System DWS'* of the example system refines the abstract binding into filter and abstract network objects. The system as a whole is shown by means of its collaboration diagram in Fig. 5. The interactions between the filters and the network are performed by signals. Signal messages of type *consumeFrame* transfer frames from the source filter to the rendering filter. In backward direction *setFrameRate* signals control the source filter's frame rate. The time constraints of the signal transfer use time marks and state that the transfer time in the abstract network is limited. Moreover natural language constraints express that the network does not reorder signals. Furthermore the calls of the *consumeFrame* operation of the source filter by the camera are constrained to occur periodically within given period time limits if the returns of the operation calls occur in time. Finally there is a maximal execution time constraint forcing the return of each *consumeFrame* operation of the display.

For the specification of the relevant properties of DWS the collaboration diagram has to be completed by describing functional and timing properties of the two filters. Since the behavior of the filters is more subtle we use timed statecharts. Fig. 6 shows the statechart of the source filter. The source filter receives frames from the camera. The received frame is buffered and the execution of the *consumeFrame* operation returns immediately. The sending of frames depends on the two control states *'fast'* and *'slow'* which correspond to two sending rate constants *r1* and *r2*. They are used in the time trigger interval of the sending transitions. Moreover, two immediate transitions are triggered by incoming *setFrameRate* signals. They change the control state.

As shown in its statechart (see Fig. 7), the rendering filter has four control states. In its initial phase it only receives frames from the network. The received frames are stored in a queue *q*. When the queue is filled over its middle level constant *ml*, the initial phase ends and the filter will concurrently send and receive frames. In *normal* mode the frames are assumed to be supplied in a fast rate from the network and the rendering filter also forwards them in a fast rate to the display. When the queue tends to become empty (length below lower level constant *ll*), the *sinkslowdown* mode is entered. Then the frames are sent to the display in accordance with the slow sending rate constant. An increase of the queue length over ml will again restore the normal mode. When the queue tends to exceed its capacity (length over upper level constant *ul*), the *sourceslowdown* mode is entered. The rendering filter sends a *setFrameRate* signal to the network and still forwards frames in high sending rate. When thereafter the queue length is normal

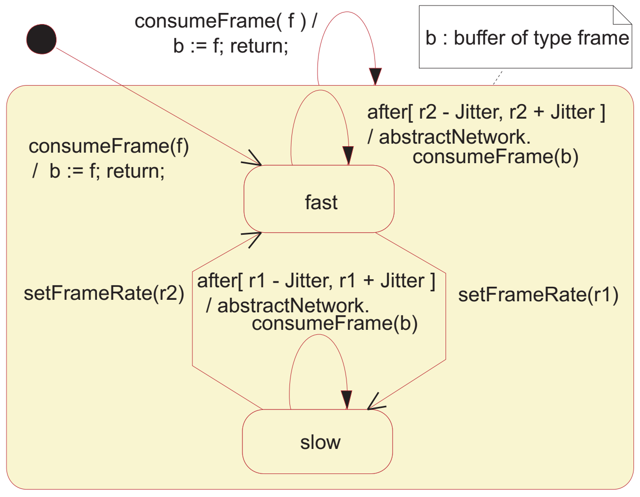
consumeFrame(f) / b := f; return;
b : buffer of type frame
consumeFrame(f) / b := f; return;
after[r2 - Jitter, r2 + Jitter] / abstractNetwork. consumeFrame(b)
fast
setFrameRate(r2)
after[r1 - Jitter, r1 + Jitter] / abstractNetwork. consumeFrame(b)
setFrameRate(r1)
slow

Figure 6. Source filter statechart

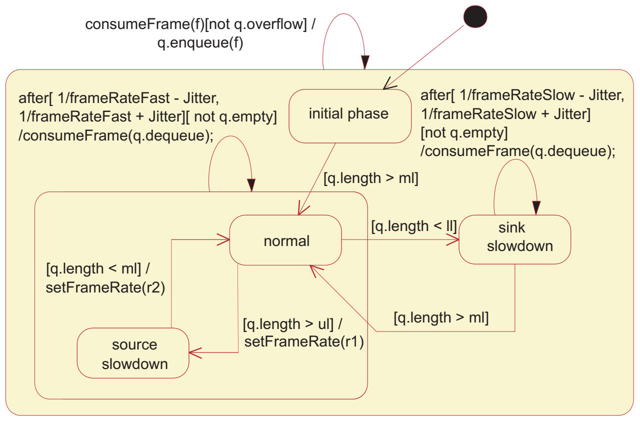
consumeFrame(f)[not q.overflow] / q.enqueue(f)
after[1/frameRateFast - Jitter, 1/frameRateFast + Jitter][not q.empty] /consumeFrame(q.dequeue);
initial phase
after[1/frameRateSlow - Jitter, 1/frameRateSlow + Jitter] [not q.empty] /consumeFrame(q.dequeue);
[q.length > ml]
normal
[q.length < ll]
sink slowdown
[q.length < ml] / setFrameRate(r2)
[q.length > ml]
[q.length > ul] / setFrameRate(r1)
source slowdown

Figure 7. Rendering filter statechart

```
PROCESS Queuing ( Elem : type ) ;
  ! Queuing of elements of type Elem in
  ! a queue-instance.
  VAR    q : queue of Elem ;
  INIT   q = empty ;
  ACTIONS
    enq (e : Elem) ≜ q′ = enqueue(q,e) ;
    deq (e : Elem) ≜ q ≠ empty ∧
                      e = front(q) ∧
                      q′= tail(q) ;
  END ;
```

Figure 8. Process type Queuing

again, the *sourceslowdown* mode is left and a *setFrameRate* signal requests for frame supply in high rate. Besides of the two frame sending transitions all transitions are immediate. The frame sending transitions are triggered by time events in order to occur periodically in accordance with the current rate and jitter constants.

4. cTLA

Like TLA [14], cTLA [7] refers to state transition system models where a state space is defined by a set of state variables, an initial condition defines the set of possible starting states, and a next state relation is given by a disjunction of so-called actions. An action is a condition over action parameters, state variables referring the current state and so-called primed variables referring to the successor state.

cTLA supports modular definitions of process types, instances of which form processes. As an example Fig. 8 shows the simple process type *Queuing*. Processes of this type are state transition systems with a state space of an infinite queue *q* for items of data type *Elem*. Initially, the queue is empty (see initial condition *INIT*). Two actions are defined, *enq* enqueues a new item *e* into the queue, *deq* dequeues the front item of the queue. Besides of simple processes, process types can describe subsystems which consist of a set of coupled processes. Like in the standard specification language Lotos [9], processes can be coupled via joint actions where two or more processes act simultaneously by each process performing one of its actions. Processes which do not participate in a joint action are assumed to perform stuttering steps. Thus, the actions of a system are conjunctions of process actions and process stuttering steps. Action parameters can be used to communicate data values between processes. Fig. 9 shows the subsystem type *Camera* as an example. A camera is a system of three processes *CC*, *CPmin*, and *CPmax*. A camera has two actions which both are three-party rendezvous of the three constituting processes. This example applies the so-called constraint-oriented style of specification where a fine-grained process structure exists and each process can represent properties of a certain single concern.

```
  PROCESSES
    CC :    FlipFlop ;
    ! Call / Return Controlflow
    CPmin : MinCamPeriod (1/rate - jitter);
    ! Time constraint
    CPmax : MaxCamPeriod (1/rate + jitter);
    ! Time constraint
  ACTIONS
  SfConsFCall (t : Real) ≜
  ! call of op consumeFrame of source filter
    CC.switch1 ∧
    CPmin.sfConsFCall(t) ∧
    CPmax.sfConsFCall(t);
  SfConsFReturn (t : Real) ≜
  ! return from op
    CC.switch0 ∧
    CPmin.sfConsFReturn(t) ∧
    CPmax.sfConsFReturn(t);
  END;
```

Figure 9. Process type Camera

Pure state transition systems express safety properties stating what a system can do, but they do not force state transitions. For that purpose liveness properties are of interest. They state that a system must not delay reactions on certain conditions over an infinite period of time. As in TLA, cTLA describes liveness indirectly by fairness statements for actions. If a fair action would be enabled infinitely, it must eventually be executed. Describing liveness indirectly by action fairness has the advantage that it cannot introduce inconsistencies to safety properties.

Moreover, cTLA supports the description of real-time

```
PROCESS MinCamPeriod(minTime : Real) ≜
  VAR    timestamp, nCall : Real;
  INIT   nCall = minTime;
  ACTIONS
    sfConsFCall(t : Real) ≜
      t = nCall ∧ timestamp′ = now;
    sfConsFReturn ≜
      nCall′ = max(0, minTime -
                      (now - timestamp));
  V MIN TIME sfConsCall(t) : t;
  END;
```

Figure 10. Process type MinCamPeriod

properties [8] following the TLA approach of [13]. We assume that a state variable *now* exists which represents the current time and is lively incremented by a clock action *tick*. It is readable by all processes of a system. Additionally, comparable to the annotation of actions with a fairness label, one can attach minimal and maximal waiting times for actions. An action can be executed only, if it is enabled for a time period of its minimal waiting time. It must be executed, before it is enabled for a longer time period than its maximal waiting time. As a short example, Fig. 10 shows the process type *MinCamPeriod* which is the type of the process *CPmin* in Fig. 9. It expresses that the time period between two consecutive *sfConsFCall* actions is at least of length *minTime* if the return of the former operation call occurred before. The return is modeled by the action *sfConsFReturn*. The two state variables store the time of the last call (*timestamp*) and the minimal waiting time for the next call which is computed when the last return occurs (*ncall*). The 'V MIN TIME' construct states the minimal waiting time property of the action *sfConsFCall*. Correspondingly a 'V MAX TIME' construct and an 'IMMEDIATE' construct exist which state maximal waiting time properties. The 'V' denotes volatile and means that disruptions of the corresponding enabling period will restart the time condition.

Unlike TLA, cTLA supports superposition of process properties. If a process instance is part of a system, then all of its relevant safety, liveness, and real-time properties shall be guaranteed by the system, too. While superposition of safety properties is not a problem (since the state variables are private to owning processes), a special solution is necessary for the semantics of forcing liveness and real-time annotations. Since an action of a process can be coupled on system level to participate in joint actions with the system environment and with other processes, an action of a process may be blocked by its environment. Consequently, time periods may exist where a process action is enabled, but cannot execute since one of its joint action peers is not enabled. With respect to this, the semantics of forcing annotations of cTLA actions is conditional and refers to the period of time where the action is as well enabled as its environment does not block it. For this conditional semantics superposition properties hold. For proofs of unconditional forcing properties, however, we have to consider the environment of a process additionally.

5. Transformation to cTLA

All diagrams of the UML based design (cf. Sec. 3) can be transformed systematically to cTLA based formal specifications following the approach of [3]. Here, we use creatively designed transformations instead which work similar to the systematic translation but yield more concise specifications. The transformations concentrate on the logical content of the diagrams. In particular, they do not model internal mechanisms (e.g., the message queues of object instances) separately.

The cTLA specifications define two systems, the *Abstract Watching System AWS* and the *Detailed Watching System DWS* by a series of process type definitions. The simple process types represent separate functional and real-time behavior constraints. The subsystem types define the mutual coupling of the constraints. Examples of the specification are already outlined in Figs. 8 to 10. The complete specification can be found in the WWW under the URL '`http://ls4-www.cs.uni-dortmund.de/RVS/MA/hk/ExaSpec.html`'.

The complete specification of the abstract watching system AWS is given by the system process type *AWS*. The actions of this system describe calls and returns of object operations. So, each execution of the action *abConsFCallLoss* models a call for the operation *consumeFrame* of the abstract binding object which will result in a loss of this frame. The action *abConsFCall* models calls for *consumeFrame* which will transfer their frames to the sink without loss. The action *dConsFCall* represents calls for the *consumeFrame* operation of the display. Thus, the cTLA model directly deals with the interactions between object instances. The functional and real-time properties of the corresponding behavior are described by the components of the system *AWS*:

- *AB*, a subsystem describing the abstract binding,
- *SRmax* and *SRmin*, two timing constraints describing the source frame rate,
- *BRmax* and *BRmin*, two timing constraints describing the rate of the frames delivered to the display.

AB again is a system type and consists of the following constraints:

- *FQ* of type *Queuing* (cf. Fig. 8) describes the functional behavior of the abstract binding,
- *LO* states that the binding will at most lose each second frame,
- *DL* specifies a bounded delay for the frame transfers.

The formal verification to be discussed later on will use *AWS* respectively the constituting constraints listed above as proof goals.

The specification of the detailed watching system DWS also is given by the system type *DWS* which, again, models calls and returns of object operations by cTLA system actions. Moreover, few internal actions (e.g., the switching from initial to normal state of rendering filter) are introduced. *DWS* is a composition of subsystems (the camera *C*, the source filter *SF*, the abstract network *AN*, and the rendering filter *RF*) with one simple process *D* modeling that the display will respond to calls of its operation *consumeFrame* by returns. The camera *C* consists of three constraint processes (see Fig. 9):

- *CC* models the blocking of the camera after an operation call till the corresponding return occurs,
- *CPmin* and *CPmax* specify minimal and maximal waiting times between calls for the operation *consumeFrame* of the source filter.

The source filter *SF* consists of four constraints:

- *B* models the functionality of the one-place buffer for the current frame,
- *CR* models the call / return functionality,
- *RC* describes the control of the sending rate,
- *TC* defines the minimal and maximal waiting times for calls of the *consumeFrame* operation of the network.

The abstract network *AN* contains five constraints:

- *FQ* and *RQ* stand for the network's functional queue-behavior,
- *FDmin* and *FDmax* constrain the delay of frames,
- *RDmax* constrains the delay of rate commands.

Finally, the rendering filter *RF* consists of six constraint processes:

- *FQ* models the internal frame queue of *RF*,
- *IP* describes the switching between initial phase and normal mode,
- *PR* describes the switching between normal mode and slow display,
- *SR* describes the switching between normal mode and slow source,
- *BL* specifies that *RF* is blocked after calling *consumeFrame* of the display until this operation returns,
- *TC* supplies minimal and maximal waiting times for *consumeFrame* calls.

We listed here all constraint processes of *DWS* since they are used as axioms resp. assumptions during the formal verification.

6. Proofs

The formal verification of the design correctness has to prove that the detailed system *DWS* correctly implements the abstract system *AWS*. Both systems are compositions of constraint processes (listed in Sec. 5). Due to the superposition property of the cTLA composition, each composition corresponds to a consistent logical conjunction of its processes, i.e., the properties expressed by a constraint process cannot be in contradiction with the properties of another constraint process and the composition will have all properties of all of its constituting processes. Based on this, cTLA supports the so-called structured verification. If one wants to prove that a system *S* implies a property *P* and if there is a subsystem *R* of *S* which implies *P*, then it is sufficient to prove that the subsystem *R* implies *P*.

We combine structured verification with TLA's notion of refinement proofs (cf. [14]). With respect to this, a proof of the correct implementation of an abstract specification *A* by a refined specification *B* is accomplished by proving the formula '*B implies A*' to be a valid TLA implication. In our example this means we have to prove '*DWS implies AWS*'. By means of structured verification this proof can be splitted into a series of proofs of the form '*some subsystem of DWS implies a constraint process of AWS*'. Thus, each relevant *AWS* property can be verified separately. Furthermore, under appropriate structuring subsystems of *DWS* can be used which are less complex than the system *DWS* as a whole. To structure the necessary proofs of a verification further and to reduce the size of the subsystems used within the proofs, moreover, subgoals, so-called lemmas, can be introduced.

In accordance with these principles, the refinement proof of the example has been accomplished by one person in about two weeks of work. Due to length restrictions it is not possible to document proofs in more detail here. The reader, however, shall gain some impression of the proof process. Therefore we outline below some typical phases of the proof in different levels of abstractions.

Typical helpful lemmas for the verification of our example system design concern the length of the frame queue in the rendering filter *RF*. So we can introduce a lemma *NoUnderflow* which states that the queue will never become empty after initialization. With the help of this lemma, for instance, it is relatively easy to prove, that *DWS* implies the *AWS* constraint *BRmax*. Then *RF*'s *dConsFCall* action occurrences only depend on *RF*'s timing constraint *TC* which nearly directly corresponds to the *AWS* constraints *BRmax* and *BRmin*. *TC* can in fact force the action *dConsFCall*, since the other processes of *RF* and the display *D* tolerate the action within the time period of *TC*: *FQ* tolerates it since its queue is not empty (assumption by lemma *NoUnderflow*). *PR* and *SR* tolerate it since their coupled actions are trivially enabled. Finally *IP*, *BL*, and *D* can only disable their coupled actions for a short time period just when *cConsFCall* has occurred. So, they will tolerate the action *cConsFCall* again and again, each time for a period which is limited only by the next occurrence of the action.

Another helpful queue length lemma to be introduced is *NoOverflow*, which states that the rendering filter's queue never will lose frames due to queue overflow. Since this lemma reflects the distributed cooperation between the most of the system components, a direct proof might to be based on a relatively large subsystem. Therefore we split the proof into a series of steps. We outline the first step in slightly more detail and give a high level description of the others. Firstly, we prove that frames will be removed from the queue in a fast minimal frequency whenever the queue length exceeds the level *ul*. We prove a corresponding invariant by means of a subsystem which consists of the constraint processes *FQ*, *IP*, *PR*, and *BL* of the rendering filter *RF* and the display constraint *D*. In this invariant, the time between two dequeueing actions is represented by the difference of now to the value of an auxiliary variable which stores the time of the last dequeueing action. Thereafter we prove a second sublemma which estimates the frequency of enqueueing actions (source filter and network sending respectively transferring as fast as they can). From the difference between dequeueing frequency and enqueueing frequency combined with the difference between level *ul* and queue capacity, we derive the minimal time period which is available before queue overflow. Then we have to prove that this time period is sufficient for the reaction, which consists of the transfer of a frame rate control command to the source filter, of the source filters reaction, and the transfer of the first slow mode frame to the rendering filter. Now we estimate the fastest possible frame enqueueing frequency for the slow frame sending mode. It has to be less than the fast minimal frequency which was analyzed in the first step in order to prove that the queue length will decrease after the reaction time has passed.

Finally, a refinement proof, i.e., a proof of a constraint of the abstract system, shall be addressed in more detail. We prove that the constraint *FQ* of the subsystem *AB* of the abstract system *AWS* is implied by a subsystem of the detailed system *DWS*. The subsystem consists of the buffering and queueing constraints of *DWS*, i.e., the buffering constraint B of the source filter *SF*, the queueing constraint *FQ* of the abstract network *AN*, and the queueing constraint *FQ* of the rendering filter *RF*. In general, a TLA based refinement proof uses a so-called refinement mapping which has some homomorphism properties and maps states of the refined system to states of the abstract system (cf. [14]). In the example this means, we have to define a mapping which has the abstract queue of the constraint *FQ* of the subsystem *AB* of *AWS* (i.e., *AWS.AB.FQ.q*) as image of the state variables of the *DWS* subsystem. Usually, the mapping corresponds to the idea behind the refinement and therefore is relatively easy to design. In our case here, the idea is, that the abstract queue corresponds to a concatenation of the contents of the buffering and queuing components of the detailed system: *DWS.RF.FQ.q* ∘ *DWS.AN.FQ.q* ∘ *DWS.SF.B.b*. Nevertheless, since items of the buffer of the source filter can be lost, the mapping is more subtle. In accordance with TLA's refinement proof principles we introduce an auxiliary prophecy variable P which equals zero when the content of *DWS.SF.B.b* will not be lost and is unequal zero otherwise. With this, the refinement mapping is:

```
AB.FQ.q ≜ IF (P = 0)
           THEN DWS.RF.FQ.q ∘ DWS.AN.FQ.q
                              ∘ DWS.SF.B.b
           ELSE DWS.RF.FQ.q ∘ DWS.AN.FQ.q
```

Now we prove, that this mapping is a refinement mapping and the corresponding homomorphism properties hold. Since *AWS.AB.FQ* is a pure safety constraint, it suffices to prove the safety conditions:

1. The mapping images of initial states of the detailed system are possible initial states of the abstract system.

2. Each transition of the detailed system is mapped to a stuttering step or a possible transition of the abstract system.

The first condition is trivially true due to the initial conditions of the three *DWS* constraints. The proof of the second condition can be split according to the action structure of *DWS*. For each action of the DWS subsystem we have to prove, that its mapping image implies the disjunction of *AWS.AB.FQ* actions and stuttering steps. With the help of the *NoOverflow* lemma, the proofs are relatively simple.

7. Concluding remarks

For the design of real-time systems we presented an approach which complements the UML based object-oriented design by enabling formal verifications of functional and hard real-time properties. We described an example application, which showed that the formal modeling and verification of interesting system parts can be accomplished with feasible efforts. Since our approach primarily profits from the designer's understanding of a system and is based on his creative proof designs, we did not discuss the application of existing supporting tools. Moreover, meanwhile we study further support, which transfers the design and software pattern approach to formal specification. For special application domains specification frameworks are under development, which supply as well re-usable specification modules as re-usable proof-elements.

References

[1] R. Alur, T.A. Henzinger, and P.-H. Ho, "Automatic Symbolic Verification of Embedded Systems", in *14th Annual Real-time Systems Symposium*, IEEE Computer Society Press, 1993, pp. 2–11.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language — User Guide*, Addison Wesley Longman, 1999.

[3] G. Graw, P. Herrmann, and H. Krumm, "Constraint-Oriented Formal Modelling of OO-Systems", in L. Kutvonen, H. Knig, and M. Tienari (eds.), *2nd Int. Working Conf. on Distributed Applications and Interoperable Systems (DAIS'99)*, Kluwer Academic Publisher, 1999, pp. 345–358.

[4] G. Graw, P. Herrmann, and H. Krumm, "Composing Object-Oriented Specifications and Verifications with cTLA", in *Workshop on Semantics of Objects as Processes (SOAP'99)*, BRICS Notes Series NS-99-2, 1999, pp. 7–22.

[5] Th. Henzinger, "The Theory of Hybrid Automata", in *11th Annual IEEE Symposium on Logic in Computer Science (LICS'1996)*, IEEE Computer Society Press, 1996, pp. 278–292.

[6] P. Herrmann, G. Graw, and H. Krumm, "Compositional Specification and Structured Verification of Hybrid Systems in cTLA", in *1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'98)*, IEEE Computer Society Press, Kyoto, 1998, pp. 335–340.

[7] P. Herrmann and H. Krumm, "Compositional Specification and Verification of High-Speed Transfer Protocols", in S. T. Vuong and S. T. Chanson (eds.), *Protocol Specification, Testing, and Verification XIV*, Chapman & Hall, Vancouver, 1994, pp. 339–346.

[8] P. Herrmann and H. Krumm, "Specification of Hybrid Systems in cTLA+", in *5th International Workshop on Parallel & Distributed Real-Time Systems (WPDRTS'97)*, IEEE Computer Society Press, Geneva, 1997, pp. 212–216.

[9] ISO, "Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour", International Standard ISO 8807, 1989.

[10] ITU-T, "ISO/IEC Recommendation X.902, IS 10746-2, ODP Reference Model: Descriptive Model", ITU, Geneva, 1995.

[11] Y. Kesten and A. Pnueli, "Timed and Hybrid Statecharts and their Textual Representation", in J. Vytopil (ed.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, LNCS 571, 1992, pp. 591-619.

[12] R. Kurki-Suonio, "Fundamentals of object-oriented specification and modeling of collective behaviors", in H. Kilov and W. Harvey (eds.), *Object-Oriented Behavioral Specifications*, Kluwer Academic Publishers, 1996, pp. 101–120.

[13] L. Lamport, "Hybrid Systems in TLA$^+$", in R. L. Grossmann, A. Nerode, A. Ravn, and H. Rischel (eds.), *Hybrid Systems*, Springer Verlag, LNCS 736, 1993, pages 77–102.

[14] L. Lamport, "The Temporal Logic of Actions", *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, 1994, pp. 872–923.

[15] K. Lano and A. Evans, "Rigorous Development in UML", in *FASE Workshop (ETAPS'99)*, Springer Verlag, 1999.

[16] K. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell", *Springer International Journal of Software Tools for Technology Transfer*, vol. 1, no. 1+2, 1997.

[17] L. Leboucher and E. Najm, "A framework for real-time QoS in distributed systems", in *IEEE Workshop on Middleware for Distributed Real-Time Systems and Service*, IEEE Computer Society Press, San Francisco, 1997.

[18] N. Lynch and F. Vaandrager, "Forward and backward simulations for timing-based systems", in J. W. de Bakker, W. P. de Roever, C. Huizing, and G. Rozenberg (eds.), *Real-Time: Theory in Practice (REX'91)*, Springer-Verlag, LNCS 600, Mook, 1992, pp. 397–446.

[19] B. Selic, G. Gullekson, and P. Ward, *Real-time Object-Oriented Modelling*, Wiley & Sons, 1994.

[20] D. Waddington and G. Coulson, "A Multimedia Component Architecture", in *1st IEEE International Workshop Enterprise Distributed Object Computing (EDOC'97)*, IEEE Computer Society Press, Surfers Paradise, 1997.