Model Checking Hierarchical Communicating Real-Time State Machines

Angelo Furfaro and Libero Nigro
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, I-87036 Rende (CS) – Italy
a.furfaro@deis.unical.it, l.nigro@unical.it

Abstract

Hierarchical Communicating Real-Time State Machines (H-CRSM) is a formal modelling language for the modular development of distributed real-time systems. The formalism is characterized by the use of state transitions with guarded commands and timing constraints, the adoption of a few distilled statecharts constructs, and the modular specification of timing constraints along a state hierarchy. This paper proposes a translation of H-CRSM into UPPAAL *which enables model checking. Translation rests on unfolding a hierarchical model on a flat representation.*

1 Introduction

In the work described in this paper, the Communicating Real-Time State Machines (CRSM) [13] formal language is chosen as part of a research effort aimed at defining a full methodology for the development of real-time systems. CRSM has an operational semantics and permits decomposition of an embedded real-time system into a collection of interacting machines. Each machine hosts a local encapsulated state, a behavior in the form of a timed state machine, and a set of typed point-to-point channels supporting CSP-like communications.

Graphical environments enabling editing, prototyping and code generation of CRSM systems have been implemented [6, 7, 10]. Due to the high-level character of a general system specification, all these tools rely on simulation for checking timing properties.

In [7] a hierarchical extension of CRSM (H-CRSM) with basic statechart constructs was defined, which is similar to existing methods for reactive systems like STATEMATE [8], ROOM [11] and UML-RT [12]. A distinguishing feature of H-CRSM, though, concerns its timing model and the expression of timing constraints which can be modularly specified along a state hierarchy and involve each transition definition.

This paper addresses timing verification of a H-CRSM specification through *model checking* [4]. For experimentation purposes, the adopted approach, similarly to that proposed in [5], centres on a mapping of H-CRSM into UPPAAL Timed Automata (TA) [2] which unfolds a hierarchical model into a flat one. The flattening procedure actually occurs from a restriction of H-CRSM, H-CRSM$^{\mathrm{R}}$, where e.g. only integer with a restricted range of values and boolean values are permitted. Restrictions are necessary to make it possible for the reachability graph to be finite in many practical cases.

2 An Introduction to H-CRSM

At the *architectural* level, a system is modelled as a net of machines connected to one another by means of unidirectional typed channels (see Fig. 8). The *behavioral* level models the behavior of each machine as a state automaton an a set of local data. A H-CRSM system model is *closed*, i.e. machines are introduced to model both the computer system and its controlled environment. Machines share no memory and execute concurrently except when they need to communicate in which case they synchronize their behavior for a message exchange.

The notion of time is involved in every state transition. Each transition is labelled by a *guarded-command* and a timing constraint: $G \rightarrow C[\tau]$. The guard G (true if omitted) is a side-effect free boolean expression that may involve constants and variables of the machine local environment. A guard is evaluated upon entering the source state of the corresponding transition. The command C can be either an *internal command* or an *i/o command*. An internal command models a local computation which takes a duration. An i/o command can be an input `ch(v)?` or an output `ch(expr)!` command referring to a channel `ch`. A communication between machines `M1` and `M2` through channel `ch` may happen if both machines are ready (*rendezvous*), in which case the couple of matched i/o commands is executed causing a simultaneous state change in both machines and the assignment `v:=expr`. The timing constraint is a timing interval $[t_{min}, t_{max}], t_{min} \leq t_{max}$ whose interpretation depends on the type of transition. In the case of an i/o command, the timing constraint expresses the possible rendezvous times. Would the time interval be unspecified, it is assumed to be $[0, \infty]$ which means that the machine is ready for communication at any time the partner is. The time constraint associated with an internal command models its possible finite duration. When omitted it is assumed to be $[0]$.

A global time notion is provided in a system through the function `rt()`. Each machine owns a special hidden input channel called `timer` which can be used to model timeouts. A command like `timer(x)?`$[\tau]$ triggers a timeout to happen

0-7803-9402-X/05/$20.00 © 2005 IEEE

after τ time units and its occurrence time to be stored in the (optional) local variable x.

Timing intervals are relative to the instant in time the current state was entered. Let τ be the enter time of state S of machine M and $[t_x, t_y]$ the time interval of a command C exiting from S. The absolute (effective) time interval for C is $[t_x + \tau, t_y + \tau]$, meaning that C cannot occur before $t_x + \tau$ but should occur before or at $t_y + \tau$ for it to be taken at all.

Let C1 and C2 be two matching i/o commands exiting respectively state S1 of M1 and state S2 of M2, $[t_1, t_2]$ and $[t_3, t_4]$ the time intervals of C1 and C2, τ_a and τ_b respectively the enter time of S1 and S2. The rendezvous can actually occur in the time interval $[t_1 + \tau_a, t_2 + \tau_a] \cap [t_3 + \tau_b, t_4 + \tau_b]$. An impossible rendezvous can deadlock M1 or M2 in the case the i/o command is the only one outgoing current state.

The set of state transitions/commands originating from a state S are conflicting: only one can possibly be chosen and executed. The other commands are discarded.

A timer with effective expire time $t + \tau$, will be automatically disabled if a conflicting command with an occurrence time $\leq t + \tau$ is chosen and executed during the firing process.

An operational semantics for CRSM was proposed in [13] with an abstract executor which relies on the *Earliest Time First* (ETF) strategy for choosing, at each step, the *maximal set* of *next events* (one per machine) which can concurrently execute at the same global earliest time. In the case of multiple and conflicting candidate events in the current state of a machine, one such a candidate event is chosen non deterministically. Basic events are associated with a rendezvous, a timer timeout, and start or finish of an internal command. Before executing the next step, the global time is first advanced to the occurrence time of next events.

2.1 Hierarchical Features

Each state of a hierarchical state machine can recursively be decomposed into a set of substates. A state that is not decomposed is said to be a *leaf*, or *basic*, state. The only state that has no parent is called the *root* or *top* state. As in [11] H-CRSM admits only the *or*-decomposition and thus the machine is the unit of concurrency.

At a given point in time, a machine's behavior can find itself simultaneously in a set of states that constitutes a path leading from one of the leaf states up to the top state. Such a set of states is called *configuration* [8]. A configuration is uniquely characterized by the only leaf state which it contains.

Firing a transition moves a machine from one configuration to another. When a configuration is left each belonging non leaf state keeps memory of its direct substate that is also part of the configuration. This substate is referred to as the *history* of the compound state. The first time a state is entered, its history coincides with its initial state.

As an example, consider the state machine in Fig. 1 where the current configuration is $C_s = \{Top, A, A2, A22\}$, and the history of B consists of state $B1$ and within it $B12$. In case of firing of the transition going from A to the H^*-connector of B, the configuration changes from C_s to $C_d = \{Top, B, B1, B12\}$.

To each compound state is associated an *initialization action* which is executed each time the compound state is entered by a transition ending on its border.

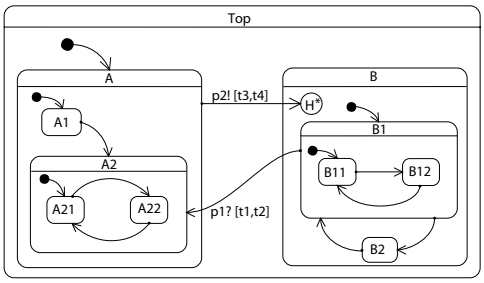
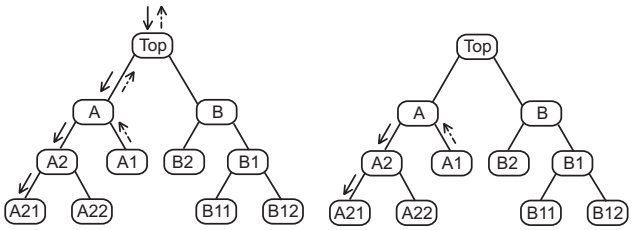


Figure 1. A hierarchical state machine

2.2 Modular vs Flat Semantics

Two possible semantic models for transition firing, called *flat* and *modular*, are considered (see Fig. 2).



(a) Flat (b) Modular

Figure 2. Enter and exit sequences under flat and modular semantics

Flat semantics takes the point of view that all the states which are part of the leaving configuration are exited and then all the states of the new configuration are entered. Flat semantics implies that the enabling of transitions originating from the states of the reached configuration, along with their timing constraints, are always re-evaluated with respect to the current time horizon.

Under modular semantics, only the states that are in the levels crossed by the transition are involved in configuration switching. The scope [8] of a transition tr is defined as the lowest common ancestor in the hierarchy of states that properly contains both source and target states of tr. When tr fires, all the states of the leaving configuration that are proper substates of its scope are exited, and all the states of the entering configuration which are substates of the scope are entered.

Fig. 2 illustrates the states which are entered and exited, under flat (a) and modular (b) semantics, when transition from $A1$ to $A2$ in Fig. 1, fires. It is worth noting that the transition from A and ending on the H^*-connector in B is unaffected in the modular approach by the configuration switch and thus its timing constraint remains relative to the time the A state was last entered.

3 UPPAAL/Timed Automata

A system consists of a network of concurrent processes - Timed Automata (TA) [1]- modelled as finite state machines. TA synchronize to one another by CSP-like channels which do not carry data values. States (*locations*) of an automaton are linked by a set of *edges* (transitions). Time is handled by means of *clock* variables. Clocks have only a *reset* operation of the form `x:=v` where `v` is a non negative integer value. In addition a clock `x` can be compared against a non negative integer constant. Clocks of a system are automatically increased following the rate of advancement of the (hidden and dense) system time. UPPAAL extends basic TA with integer (and boolean) variables and arrays of integers, clocks and channels. TA processes are parameterized *template processes*. Global data variables can be used for process synchronization. State transitions admit three (optional) components: (i) a *guard*, (ii) a *synchronization* operation (`?` for input and `!` for output) on a channel, and (iii) an *action-part* consisting of a set of clock resets and variable assignments. The action part of an output command is executed before that of the matching input command. A guard (`true` if omitted) is a boolean expression built over data variables and clock constraints. It defines the transition enabling condition. For bounded delay in a location s, a clock *invariant* can be attached to s (see Fig. 3) as a *progress* condition. UPPAAL supports also *committed* locations which must be exited immediately (without passage of time), and *urgent channels* whose synchronizations must be fired as soon as possible.

UPPAAL consists of a graphical editor, a simulator and a verifier. For systematic property assessment, the verifier tries to build the reachability graph of the model, where execution states are organized into equivalence classes (zones).

Because guards involving clocks on urgent channels can cause the generation of non-convex zones whose handling may be computationally expensive [3], this construct is not allowed in current version of UPPAAL.

Safety (e.g., absence of deadlock) and bounded liveness (e.g. an end-to-end time constraint) properties can be verified by reachability analysis using a subset of TCTL formula [2] for networks of timed automata. Admitted formulas refer to local state properties, i.e. boolean expressions over predicates on locations and integer variables and clock constraints. $E <> \phi$ means *Possibly* ϕ (a state can be reached in which ϕ holds). $A[]\phi$ means *Invariantly* (in all states) ϕ holds. $E[]\phi$ means *Potentially Always* ϕ (a path exists where ϕ holds in all reached states). $A <> \phi$ means *Always Eventually* ϕ (equivalent to: not $E[]$ not ϕ). $\phi -- > \psi$ means ϕ *always leads to* ψ (equivalent to: $A[](\phi$ imply $A <> \psi)$.

4 Mapping H-CRSM on to Timed Automata

For translation purposes, the following considers the dialect H-CRSM$^{\mathrm{R}}$ of H-CRSM where:

– data variables directly correspond to those admitted in UPPAAL, e.g. integers with a limited range of values;

– global time, returned by `rt()` function or `timer(x)?` commands, cannot be used in a machine except for purposes like that shown in Fig. 7, e.g. to make the timing constraint of a command relative to a fixed time instant whose value was previously stored in a variable;

– computation associated to an internal command is assumed to be expressed by an arbitrary algorithm with a bounded execution time. H-CRSM$^{\mathrm{R}}$ internal commands are restricted to computations that can be modeled by UPPAAL features.

The translation process maps each machine of an H-CRSM specification into a corresponding UPPAAL template process and each channel into a corresponding urgent channel so as to comply with the ETF strategy. For each channel *ch* transmitting data, a corresponding variable `ch_v` is introduced for storing the value of the transmitted data.

More complex is the task of translating machine behaviors because it requires mapping a hierarchical automaton into an equivalent flat one (*unfolding*) and it needs re-interpretation of group transition timing constraints. Timing constraints could be expressed as guard constraints on UPPAAL clocks, however current version of the tool does not allow clock constraints to be used in guards of urgent channels [3]. Therefore, the proposed mapping still relies on urgent channels but uses *virtual clocks* and a *synchronizer*. A virtual clock is an integer variable `clkM`. Its reset is responsibility of the corresponding automaton M but its increase is delegated to an extra TA, the synchronizer, which is charge of orchestrating all the virtual clocks. The resultant approach delivers a discrete time model for H-CRSM. Virtual clocks must have a limited range of values. The synchronizer increases the value of a virtual clock only if it is less than the admitted maximum.

Fig. 3 depicts a synchronizer schema which has one state and uses one UPPAAL clock `y`. The *action* urgent channel is specifically introduced for modelling the execution of an instantaneous action not related to i/o commands (i.e., start/finish of an internal command and firing of a timer).

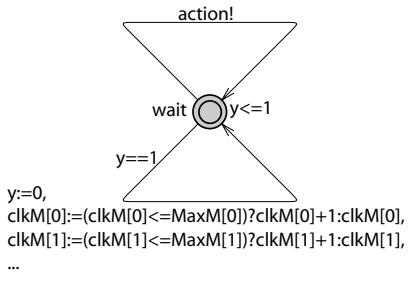


Figure 3. Schema of a synchronizer

The synchronizer ensures that time gets advanced of a unit when no more action-transitions can occur at current time.

The following describes the unfolding algorithm that maps the behavior of a H-CRSM machine into a flat TA under modular semantics of transition firings.

4.1 State Unfolding Rules

Let M be a machine. For each state s of M different from its top state:

1. Introduce in the declarations of the template corresponding to M a distinct constant $path(\mathsf{s})$ whose name is constructed on the basis of the pathname of state s.

2. For each level l of the hierarchical automaton add an integer variable `clkM[`l`]` modelling a virtual clock associated to that level. Let $MaxM[l]$ be the maximum finite value appearing in a timing constraint of a transition originating from a state whose level is l. Associate to variable `clkM[`l`]` the range $[0, MaxM[l] + 1]$.

3. If state s is not a leaf state, introduce a variable `h[`$path(\mathsf{s})$`]` for taking track of the history of s. Initially the value of `h[`$path(\mathsf{s})$`]` is set to -1 to indicate that the history of state s is undefined.

4. Introduce a committed location $e(\mathsf{s})$ and a location $m(\mathsf{s})$ which is committed if s is not a leaf state. Introduce a transition from $e(\mathsf{s})$ to $m(\mathsf{s})$ whose action consists in updating the history of the parent state (if different from the root) of state s, i.e. `h[`$path(parent(\mathsf{s}))$`]:=`$path(\mathsf{s})$, and in resetting the virtual clock associated with its level, i.e. `clkM[`$level(\mathsf{s})$`]:=0`.

5. If s is not a leaf state and there exists at least one transition ending on a history pseudo state of s, introduce a committed location $he(\mathsf{s})$. For each child q of state s add a transition with guard `h[`$path(\mathsf{s})$`]==`$path(\mathsf{q})$ originating from $he(\mathsf{s})$ and ending on $e(\mathsf{q})$, and a transition from $he(\mathsf{s})$ to $m(\mathsf{s})$ with guard `h[`$path(\mathsf{s})$`]==-1`, whose action consists in resetting the virtual clock associated with the level of state s.

6. If s is not a leaf state and there exists at least one transition ending on a deep history pseudo state of s, or of one of its ancestors, introduce a committed location $dhe(\mathsf{s})$. For each child q of state s add a transition with guard `h[`$path(\mathsf{s})$`]==`$path(\mathsf{q})$ originating from $dhe(\mathsf{s})$ and ending on $e(\mathsf{q})$, in the case q is a leaf state, or on $dhe(\mathsf{s})$ otherwise. Add a transition from $he(\mathsf{s})$ to $m(\mathsf{s})$ with guard `h[`$path(\mathsf{s})$`]==-1`. Associate to each of these transitions an action which consists in resetting the virtual clock associated with the level of state s.

7. If state s is not a leaf state, let d be the state entered by the default transition of s. Introduce a transition from $m(\mathsf{s})$ to $e(\mathsf{d})$ and associate to this transition an action corresponding to the setting specified by the default transition of s.

For the top state, add a committed location, mark it as *initial* and add a transition from this location to location $e(\mathsf{d})$ where d is the state entered by the default transition. Associate to this transition an action corresponding to the setting specified by the default transition of the top state.

4.2 Transition Unfolding Rules

For each transition t with an explicit guard, introduce a corresponding boolean variable g_t initialized to `false` in the local environment. Let s and d indicate respectively the source and the destination states of t:

1. Let r refer to location $e(\mathsf{d})$ if state d is not a pseudo state, to location $he(\mathsf{d})$ or to location $dhe(\mathsf{d})$ if it respectively is a shallow or a deep history pseudo state.

2. Let $src(t)$ be the set of all leaf states which are descendant of state s. In the case state s is a leaf, it is the only element in $src(t)$. For each state q in $src(t)$:

   - introduce a transition originating from location $m(\mathsf{q})$ by applying the rules of subsection 4.3 to the specification of transition t
   - if t has an explicit guard, augment the action of the transition from $e(\mathsf{q})$ to $m(\mathsf{q})$ by assigning to g_t the result of the evaluation of t guard.

3. Let $enter(t)$ be the set of all ancestors of state d which are not ancestors of state s. The destination of the transition(s) introduced in the previous step will be location r, if $enter(t)$ is empty, it will be a new committed location $e(t)$, otherwise. If $enter(t)$ is not empty, add a transition from location $e(t)$ to location r whose action consists in:

   - resetting virtual clocks associated with levels of states in $enter(t)$
   - updating the history variables related to states in $enter(t)$.

The introduction of boolean variable g_t for each transition t equipped of an explicit guard is required, under modular semantics, to achieve the right interpretation of the guard when t is a group transition. The value of g_t is computed upon entering location $m(\mathsf{s})$, where s is the source state of t. The value of g_t is then unaffected by the occurrence of internal transitions to macrostate s. As an optimization, when t is not a group transition the introduction of g_t can be avoided.

In the case flat semantics is adopted, there is no need for introducing a virtual clock per level. Therefore, only one virtual clock per machine is used. When a transition fires, all the states of the source configuration are left and all the states of the destination configuration are (re-)entered. As a consequence, the timing constraints of enabled transitions are relative to this instant in time. Similarly, no variable g_t must be introduced because also data constraints (guards) are re-evaluated at each state entering time.

4.3 Command Mapping

Figg. 4 to 7 summarize the basic translation rules for H-CRSM$^{\mathrm{R}}$ commands. All these rules take into account the need of shifting timing constraints with respect to the value of the clock corresponding to the level of the source state.

Figg. 4(a) and 4(b) show the general rules for translating i/o commands. Fig. 5(a) illustrates the general translation rule for timers: the upper transition does not specify synchronization over an urgent channel and thus it may be fired or not when its guard evaluates to `true`. The guard of this edge is augmented by the expression `y==0` to comply with the achieved

discrete time model. Fig. 5(b) shows a simplified version of the Timer translation rule that may be used when `lb==ub==d`. Fig. 6(a) depicts the rule for internal commands. Fig. 6(b) and 6(c) show simplifications of this rule which can respectively be used when `lb==ub==d` and when the command is instantaneous, i.e. `lb==ub==0`.

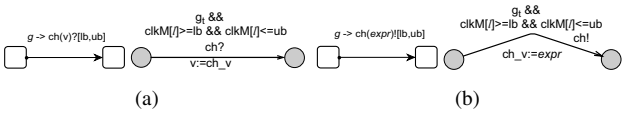


Figure 4. Input and Output command translation rules

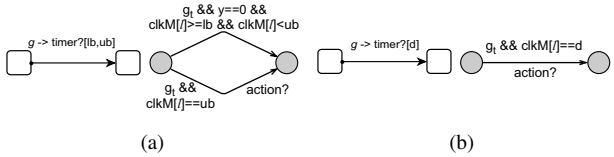


Figure 5. Timer translation rules.

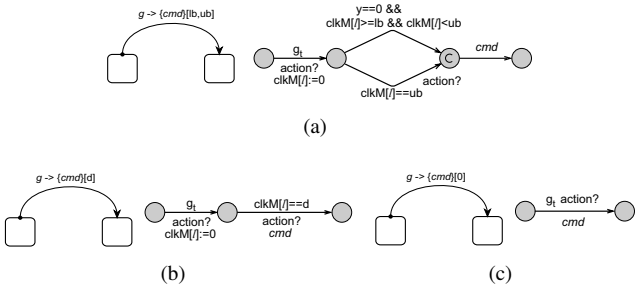


Figure 6. Internal command translation rules

Fig. 7 portrays a rule for translating in TA a H-CRSM$^{\mathrm{R}}$ sub-behavior where a transition is constrained to occur exactly after `p` time units since state s0 was entered. An integer variable `d` is introduced which models the time elapsed since the enter time of s0. The synchronizer is in charge of incrementing `d` together with virtual clocks. The construction in Fig. 7 can easily be adapted to achieve a periodic behavior by making s0 the destination state of the timer transition exiting from the dashed area.

Proposed rules ensure the flat automaton is polynomial in the size of the corresponding hierarchical machine. In particular, the number of locations is bounded by $O(|S| + |T|)$, the number of transitions by $O(|T| \times |S|)$, and the number of variables by $O(|S| + |T| + |V| + L)$, where S, T, V are respectively the state, transition and variable sets of the original machine, and L is the depth of the hierarchical automaton.

5 Modelling and Verification Example

Figure 8 shows a simple H-CRSM system model. Figure 9(a) depicts the behavior of machine Device which models a processing activity made up of four steps. Initially, the

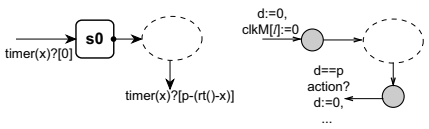


Figure 7. Translation rule for an activity lasting exactly `p` time units

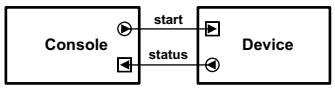


Figure 8. System model

machine stays in state Stop waiting for a synchronization on channel *start*. When this signal arrives, the machine moves to macrostate Proc where the four processing steps are done in sequence. In state Proc, the machine can be interrupted by a signal from channel *status* which asks for displaying the current processing status. Displaying operation takes one unit of time. After that, the machine restarts its processing from the point where it left off.

Timing constraint on the group transition from state Proc to state Display ensures that `t` time units elapse between the handling of two consecutive displaying requests.

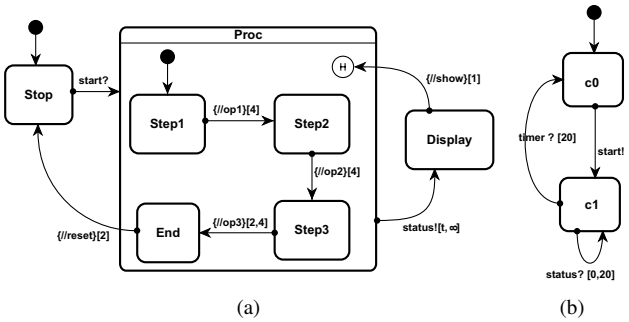


Figure 9. Behaviors of machine Device **and** Console

Figure 9(b) shows the behaviour of machine Console which after sending a signal through channel *start*, continuously try to engage a synchronization on channel *status*. If twenty time units elapse without successful synchronization, the timeout transition fires and initial state c0 is re-entered.

Figure 10 shows the Device machine translated into TA. Machine Console is translated with the same procedure. The achieved UPPAAL system can be used for verifying the model against a set of properties. An important property concerns liveness of machine Device, i.e. once the sequence of four operations is started it will be always completed. This property can be checked by asking the UPPAAL verifier if "*every time state Proc is entered, state Stop will eventually be reached*" which is expressed in UPPAAL syntax as: `Device.Proc --> Device.Stop`. Satisfaction of this property depends upon the value of constant `t`. If the value of `t` is strictly greater than

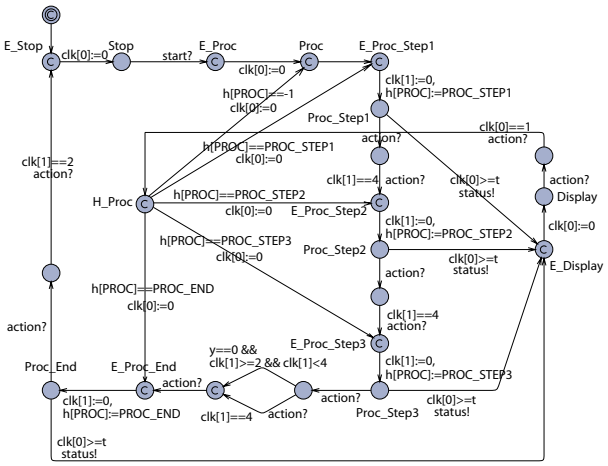


Figure 10. UPPAAL **TA obtained by translating machine of Fig. 9(a)**

zero the property is found satisfied. When `t` is equal to zero it is possible that machine Device engages an unbounded number of synchronizations through channel *status* before executing the next processing step.

Correctness of real-time systems is often specified by means of bounded-liveness properties which express that certain activities must complete before a deadline. Specification of bounded-liveness properties can require model decoration [9] in order to be formulated in UPPAAL syntax. For example, checking that the execution of the four processing steps takes at most `k` time units can be accomplished by introducing a boolean variable `enable` and an UPPAAL clock `x` in the template corresponding to machine Device. The action part of the transition from location E_Proc to location Proc is augmented by setting the value of variable `enable` to `true` and resetting clock `x`. The transition that re-enters location E_Stop is augmented by assigning `false` to variable `enable`. The bounded-liveness property is then expressed as: `A[] Device.enable imply Device.x<=k`.

Let `u` be lowest value of `k` for which the property is satisfied, `u` is the upper bound on the completion of processing activities.

The value of `t` influences the number of times a request coming on channel *status* can be handled during a processing cycle and then the value of `u` depends on it. It emerged that for `t` in $[1, 4]$ `u`=17, for `t`=5 `u`=15 and for `t`=6 `u`=14.

It is interesting to note that when the value of `t` is greater than or equals to 6, communications on channel *status* never happen. This was checked by asking the UPPAAL verifier if state Display is eventually reached, i.e. `E<> Device.Display`. The verifier found this property satisfied only for values of `t` less than 6.

6 Conclusions

Recently, the Violin [7] tool was extended so as to mechanize the translation approach from H-CRSM into UPPAAL/Timed Automata proposed in this paper. The approach relies on virtual clocks and discrete time. Would UPPAAL support guards involving clock constraints in combination with synchronization on urgent channels, it would be possible to use UPPAAL standard clocks and dense time and take full advantage of the state reduction techniques that the tool offers. On-going work is geared at:

- improving the translation approach by integrating a H-CRSM system specification with issues of a runtime scheduling control structure (e.g., EDF on a single processor)
- experimenting with the implementation of run-time infrastructures for H-CRSM in the context of a configurable microkernel.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, LNCS 3185, pages 200–236. Springer, 2004.

[3] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, LNCS 3098. Springer, 2004.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[5] A. David, M. O. Möller, and W. Yi. Verification of UML statechart with real-time extensions. Technical Report TR 2003-009, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2003.

[6] G. Fortino and L. Nigro. A toolset in Java2 for modeling, prototyping and implementing communicating real-time state machines. *Microprocessors and Microsystems*, 23(3):573–586, 2000.

[7] A. Furfaro, L. Nigro, and F. Pupo. Real-time systems development using hierarchical state machines. In L. Böszörményi and P. Schojer, editors, *Modular Programming Languages*, LNCS 2789, pages 110–121. Springer, 2003.

[8] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[9] M. Lindhal, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller: an industrial case stuy using UPPAAL. *Software Tools for Technology Transfer*, 3(3):353–368, 2001.

[10] S. C. V. Raju and A. C. Shaw. A prototyping environment for specifying and checking Communicating Real-time State Machines. *Software–Practice and Experience*, 24(2):175–195, 1994.

[11] B. Selic, G. Gullerkson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley Professional Computing. John Wiley & Sons, Inc., 1994.

[12] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems, 1998.

[13] A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, 1992.