Verification of COMDES-II Systems Using UPPAAL with Model Transformation

Xu Ke[1], Paul Pettersson[2], Krzysztof Sierszecki[1], Christo Angelov[1]

[1] Mads Clausen Institute for Product Innovation, University of Southern Denmark
Alsion, Alsion 2, 6400 Soenderborg, Denmark
{xuke, ksi, angelov}@mci.sdu.dk
[2]Department of Computer Science and Electronics, Mälardalen University
Högskoleplan 1, Box 883, 72123, Västerås, Sweden
paul.pettersson@mdh.se

Abstract. COMDES-II is a component-based software framework intended for Model Integrated Computing (MIC) of embedded control systems with hard real-time constraints. We present a transformational approach to formally verifying both timing and functional behavior of COMDES-II systems using UPPAAL. The proposed approach adopts timed automata in UPPAAL as the semantic units to which the behavioral semantics of COMDES-II are anchored, such that a COMDES-II system can be equivalently transformed into the timed automata models in UPPAAL, and verified with precise preservation of system operational semantics. In the paper a concrete discussion of semantic transformation from COMDES-II to UPPAAL is given, and a turntable case study is developed to show how to apply the presented approach in practice.

Table of Contents

1. Introduction ... 4
2. Modeling in COMDES-II ... 6
3. Timed Automata in UPPAAL ... 10
4. Transformation from COMDES-II to UPPAAL ... 11
   4.1 Transformation of Concurrency and Time ... 11
      4.1.1 Modeling Actors in UPPAAL ... 12
      4.1.2 Modeling Actor Interaction in UPPAAL ... 13
      4.1.3 Modeling Actor Concurrency in UPPAAL ... 14
      4.1.4 Modeling Discrete-Time Scheduler in UPPAAL ... 16
   4.2 Transformation of Functionality ... 17
5. Turntable Case Study ... 19
   5.1 COMDES-II Design of Turntable system ... 20
   5.2 Model Transformation of Turntable Case Study Design ... 31
      5.2.1 Instantiation of Task Control Blocks ... 32
      5.2.2 Actor Interaction in UPPAAL ... 33
      5.2.3 Actor Functional Behavior in Timed Automata ... 34
      5.2.4 Formulation of System Properties ... 38
6. Conclusion ... 41
References ... 42
Appendix A: UPPAAL Model of Turntable Case Study ... 43
   Global Declarations ... 43
   System Declarations ... 55
   Timed Automata Models ... 56

List of Figures:

Fig. 1. Semantic anchoring process of CODMES-II 4
Fig. 2. Hierarchical architecture model of a COMDES-II system 6
Fig. 3. Split-phase execution of actors under timed multitasking 7
Fig. 4. Interaction between a state machine FB and a modal FB 8
Fig. 5. An example of state machine FB in COMDES-II 9
Fig. 6. Status transition graph of actor tasks 13
Fig. 7. Actor concurrency over discrete-time in UPPAAL 14
Fig. 8. Discrete-time scheduler automaton in UPPAAL 16
Fig. 9. UPPAAL automaton equivalent to SMFB_1 17
Fig. 10. The turntable system setup 19
Fig. 11. System design illustrated by an actor diagram 20
Fig. 12. *TurntableSupervisor* actor 23
Fig. 13. *TurntableSupervisor* actor task 24
Fig. 14. *TurntableController* actor... 25
Fig. 15. *DrillClampSupervisor* actor.. 26
Fig. 16. *DrillClampSupervisor* actor task 27
Fig. 17. *DrillController* actor.. 28
Fig. 18. *TesterSupervisor* actor 29
Fig. 19. *TesterSupervisor* actor task... 29
Fig. 20. *TesterController* actor 30
Fig. 21. Behavior of *Input* environment process: product loading 30
Fig. 22. Behavior of *Output* environment process: product unloading 31
Fig. 23. Timed automata for *TurntableSupervisor* actor functional behavior 35
Fig. 24. Timed automata for *TurntableController* actor functional behavior 38
Fig. 25. Timed automata for discrete-time scheduler.. 56
Fig. 26. Timed automata for *TurntableController* actor functional behavior 56
Fig. 27. Timed automata for *DrillController* actor functional behavior 56
Fig. 28. Timed automata for *TesterController* actor functional behavior 56
Fig. 29. Timed automata for *TurntableSupervisor* actor functional behavior 57
Fig. 30. Timed automata for *DrillClampeSupervisor* actor functional behavior ... 58
Fig. 31. Timed automata for *Input* system 58
Fig. 32. Timed automata for *Output* system 58
Fig. 33. Timed automata for *TesterSupervisor* actor functional behavior.............. 59

List of Tables:

Table 1. Behavioral differences between COMDES-II and UPPAAL 11
Table 2. Sensors and actuators of the system 19
Table 3. Execution information of actors 21
Table 4. Messages exchanged among actors 21
Table 5. Actor output functions 34
Table 6. System properties and verification results 39

1. Introduction

Recently emerging concepts and techniques, such as *model-integrated computing* (MIC) and *component-based design* (CBD) are considered as appropriate methods for efficient development of reliable embedded software systems [1]. On one hand, MIC advocates a domain-specific model-driven approach for the embedded software development, by equipping developers with a domain-specific modeling language (DSML) that captures the modeling concepts, constraints and assumptions of the application domains. On the other hand, CBD can be regarded as one of the most suitable design paradigms for MIC, due to the considerable benefits brought by reusability of components and higher-level of abstraction. Moreover, from a software engineering point of view, CBD is also an effective way to bridge the gap between conceptual system design models and concrete system implementations [2], provided that an automatic code generation technique is developed.

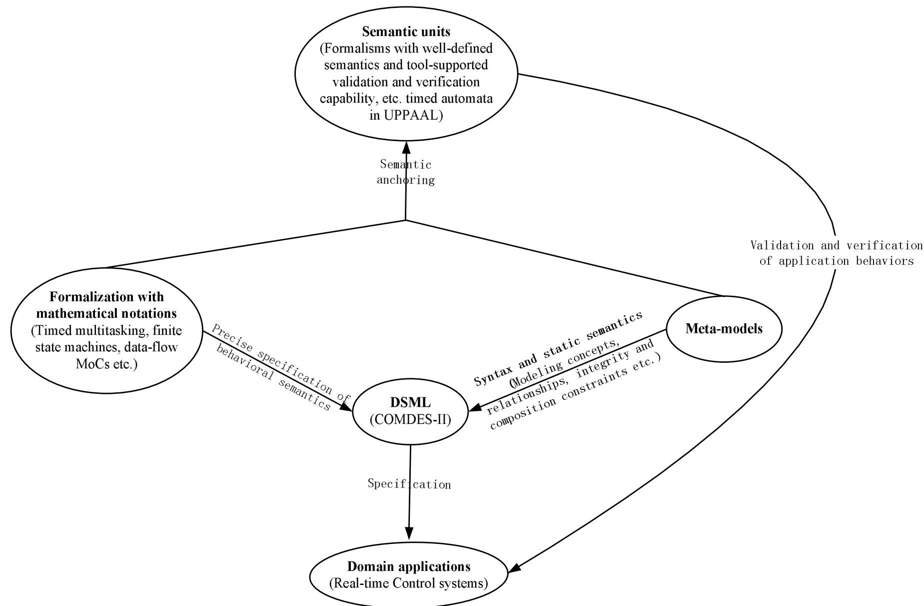

Fig. 1. Semantic anchoring process of CODMES-II

COMDES-II is a component-based software framework adopting MIC as a methodological basis for the development of distributed control systems with hard real-time constraints. In order to achieve this objective, COMDES-II provides various kinds of component models to address the critical domain-specific issues, such as system concurrency, real-time operation, sequential behavior with continuous computation etc. using a *separation-of-concerns* approach [3]. A meta-modeling process formally defines the syntax and static semantics of the framework component

models [1], however, specification and verification of the composed component behaviors still remain a challenging problem.

Semantic anchoring [1,4] is a promising approach to transformational specification and verification of system behavioral semantics, by relying on semantic units (such as finite state machines, timed automata etc.) with well-defined operational semantics and tool support. Briefly, the elements and their relationships in a DSML can be equivalently transformed onto the counterparts in an executable semantic unit with well-defined behavior, which can subsequently be validated and verified – by preserving the original system operational semantics – using the supportive toolsets. This transformation process from the original DSML to the corresponding semantic unit is referred to as semantic anchoring, as shown in Fig. 1.

We choose timed automata in UPPAAL as the semantic unit, and this paper presents the concrete process of developing such a transformational approach to specify and verify the behavior of COMDES-II systems via semantic anchoring. The structure of paper follows a logical sequence: Section 2 and 3 provide an overview about COMDES-II component models and timed automata in UPPAAL respectively, which would give a general perspective on the semantic gaps between two kinds of systems. Section 4 subsequently describes in details how the behavioral semantic gaps are bridged. Section 5 presents a turntable case study as an example to demonstrate the application of the proposed approach. Finally, the concluding section summarizes the features of our work and their implications.

2. Modeling in COMDES-II

As a component-based design framework intended for the real-time control systems, COMDES-II takes into account both the architectural and behavioral characteristics of the targeted domain during a system development process.

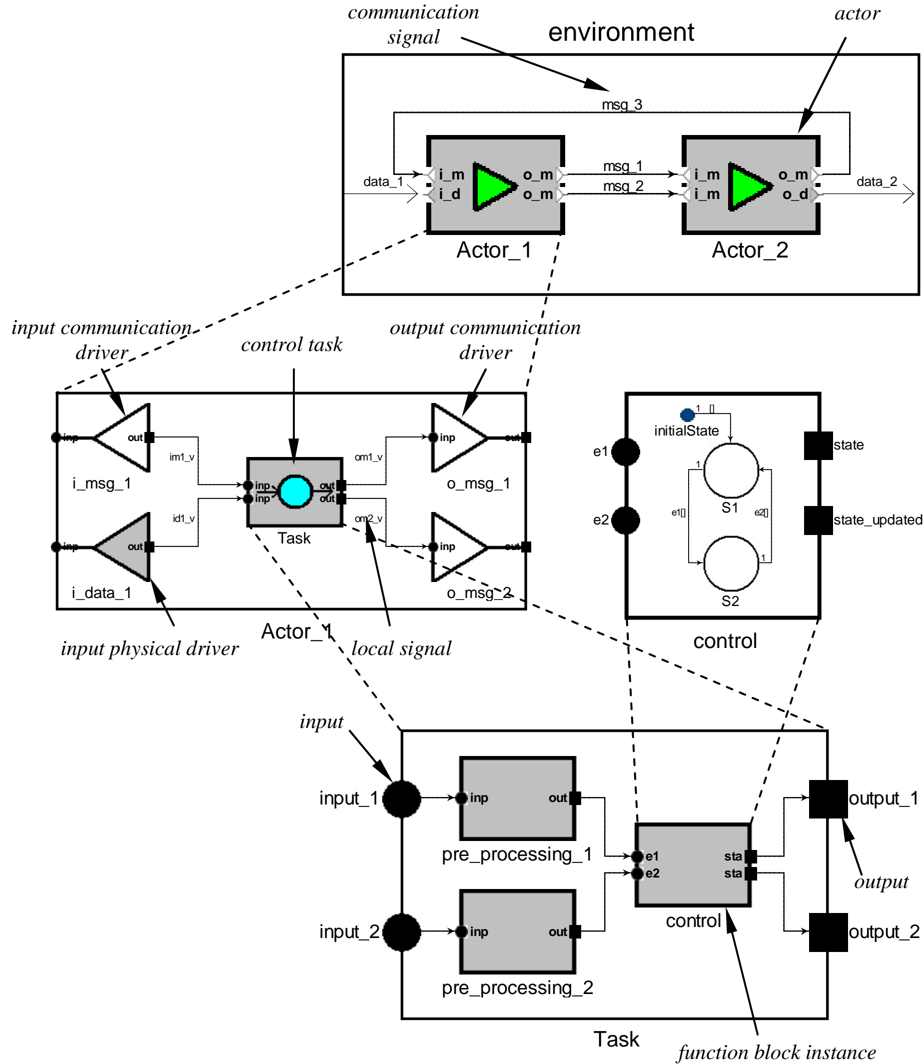

Fig. 2. Hierarchical architecture model of a COMDES-II system

COMDES-II employs a hierarchical model to specify systems architecture as illustrated in Fig. 2: at the system level a control application is conceived as a network of communicating *actors* (active components), which interact transparently with each

other by exchanging labeled messages (signals), following an asynchronous producer-consumer protocol.

At the actor level, an actor is specified as a software artifact containing multiple *I/O drivers* and a single *actor task* (execution thread). The I/O drivers are responsible for sensing or actuating signals from/to network or physical units, while the actor task processes the acquired signals to fulfill the required functionality which is specified by a composition of different *function block instances.* Function block instances are instantiations of reusable and reconfigurable function block *types*, which can be categorized into four function block *kinds* (meta-types): *basic*, *composite*, *modal* as well as *state machine* function blocks. A detailed description of the CODEMS-II systems architecture and function block models is out of the scope of this paper and we refer the interested readers to [3].

As to the systems behavior modeling, a separation-of-concerns approach is extensively applied. In COMDES-II, concurrency and time are separated from the functionality, in the sense that scheduling and real-time issues are specified with respect to actors, while the functional behavior can be represented by the composition of different kinds of function block instances contained within the actor tasks.

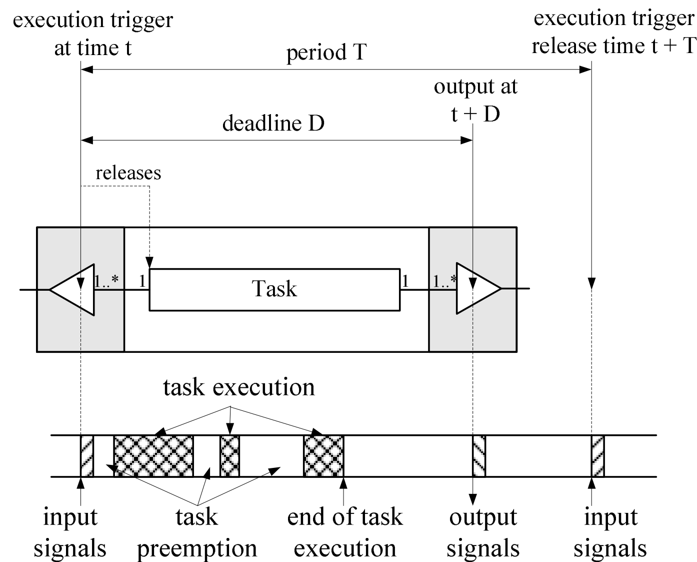

Fig. 3. Split-phase execution of actors under timed multitasking

Scheduling of actors follows a *fixed-priority timed multitasking* (TM) strategy [5], in which actors can be activated by either a periodic or an aperiodic event, and execute preemptively according to the assigned priorities with *non-blocking read-do-write* semantics. The core element of TM in COMDES-II is a time-triggered scheduler which controls I/O activities and execution status of actors over discrete-time, such that the timestamps of actor behavior are represented as the multiples of a basic timing unit (i.e. the period of scheduler activation).

Upon activation, input drivers of the activated actor will be invoked (*read*) in logically zero time to acquire all input signals which are latched throughout the whole actor execution. The activated actor task will process (*do*) exactly once the input data stepwise along the time axis, as long as it becomes the highest priority task among all released/preempted tasks in the processor. The processed data will then be buffered into output drivers that can be atomically executed to generate (*write*) output signals when the corresponding actor *deadline* expires. If the deadline of an actor is not be specified (i.e. deadline = 0), the actor output drivers will be immediately executed when the actor task finishes its computation. This *split-phase* execution pattern of COMDES-II actors is illustrated as in Fig. 3.

The four kinds of function blocks (FBs) defined in COMDES-II are pure functional components implementing concrete computation or control algorithms to specify different kinds of system functional behavior. Specifically, basic and composite FBs are used to model the data-flow computation process executing in a single mode of operation, while state machine FBs and modal FBs are jointly used to specify the system reactive behavior (control-flow) combined with multi-mode (modal) data-flow computations. Among these two kinds of behaviors, data-flow computation is not of our analysis interest, but rather, the system reactive behavior is.

A COMDES-II state machine FB consists of a number of binary *event/guard* inputs, an *event-driven* state machine model, and exactly two outputs: *state* and *state_updated*. When a state machine FB is executed, the integrated state machine parses the binary event/guard input signals, determines the current state and updates two outputs: *state* and *state_updated*. Here, *state* represents the currently active state, and *state_updated* will be set *true* if a state transition has happened, otherwise it is not blocked, but set as *false*.

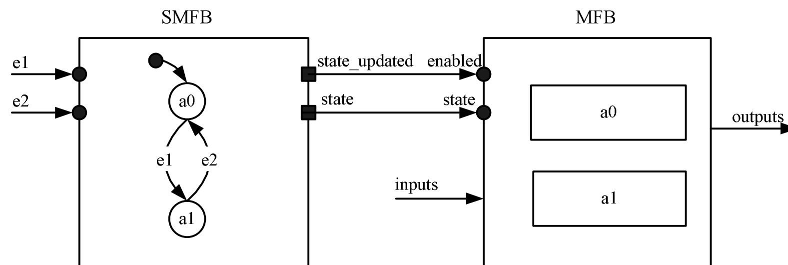

Fig. 4. Interaction between a state machine FB and a modal FB

The two output signals from a state machine FB will be used by the corresponding modal FBs to execute the control actions associated with the specific state, as shown in Fig. 4. A modal FB has a number of operation modes (states) and each mode contains a function block diagram representing the control action to be performed. The selection of executing state is decided by the currently active *state* information provided from the supervisory state machine FB, whereas the enabledness of executing state is determined by the *state_updated* value, i.e. the control action should only be performed when a state transition occurs, since the state machine model is event-driven.

In this computation model, a modal FB merely acts as a component containing multi-mode (modal) data-flow execution actions, whereas the system control logic (reactive behavior) is actually specified by the corresponding state machine FB. As a result we will introduce state machine FBs in more details, and refer the interested readers to [] for more information about the other kinds of function blocks.

An example of state machine FB called *SMFB_1* is illustrated as in Fig. 5. The *SMFB_1* contains three binary event inputs *e1*, *e2* and *e3*, an event-driven state machine model, and exactly two outputs: *state* and *state_updated*. The internal state machine includes a dummy *initialState* pointing to the actual initial state *s1*, two states *s1* and *s2*, three state transitions that are labeled by events and transition orders. Transition events are manipulated as input signals of *SMFB_1* acquired from input drivers or preprocessing FBs, and transition orders are numbers starting from 1 to indicate the importance of transitions (e.g. two outgoing transitions from *s1*). Based on transition orders, the choice of transition to be fired is deterministic when multiple transition triggers associated with the current state are true at the same time, as required in safety-critical control systems.

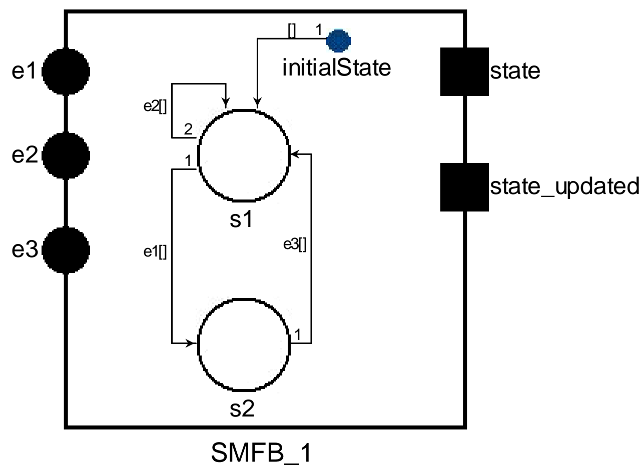

Fig. 5. An example of state machine FB in COMDES-II

When the host actor is activated and then *SMFB_1* is executed, its internal state machine parses binary input event signals, determines current state and updates two outputs: *state* and *state_updated*. The two output signals will be used by the associated modal FBs to perform the corresponding control actions, as introduced above.

3. Timed Automata in UPPAAL

The theory of timed automata has proven to be useful for specification and verification of real-time systems. In this section we briefly review the basic definition needed in this paper. We refer the reader to [6] for a more thorough description for the timed automata used in the UPPAAL tool [7].

Assume a finite set of real-valued variables C standing for clocks, and a finite set of actions Act. Let $B(C)$ denote the set of Boolean combination of clock constraints of the form $x \sim n$ or $x - y \sim n$, where $x, y \in C$ and n is a natural number.

Syntactically a timed automaton A is a tuple $<N, l_0, E, I>$ where: N is a finite set of locations, $l_0 \in N$ is the initial location, $E \in N \times B(C) \times Act \times 2^C \times N$ is the set of edges, and $I : N \rightarrow B(C)$ assigns invariants to locations.

The semantics of a timed automaton is a timed transition system with states of the form $<l, u>$, where $l \in N$ and u is a clock assignment assigning all clocks in C to a non-negative real-number. Transitions are defined by the two rules:

- (discrete transitions) $<l, u> \xrightarrow{a} <l', u'>$
  if $<l, g, a, r, l'> \in E$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$
- (delay transitions) $<l, u> \xrightarrow{d} <l, u \oplus d>$
  if $u \in I(l)$ and $(u \oplus d) \in I(l)$ for a non-negative real number d

where $u \oplus d$ denotes the clock assignment which maps each clock x in C to the value $u(x)+d$, and $[r \mapsto 0]u$ is the clock assignment u with each clock in r to be zero.

A network of automata is a finite set of *automata processes* composed in parallel with a CCS-like parallel composition operator [8]. For a network with the timed automata $A_1, \ldots, A_n$ the intuitive meaning is similar to the CCS parallel composition of $A_1, \ldots, A_n$ with *all* actions being restricted, that is, $(A_1 \mid \ldots \mid A_n) \setminus Act$. Thus an edge labeled with action a must synchronize with an edge labeled with an action complementary to a, and edges with the silent τ action are internal, so they do not synchronize. In UPPAAL '?' and ' !' are used to represent complementary actions, so *a?* and *a!* are considered complementary and can synchronize. The silent τ action is represented in UPPAAL by no synchronization action (i.e., an edge with an empty synchronization action).

Finally, we note that the flavor of timed automata used in the UPPAAL tool is extended with data variables with finite domains, including Booleans and finite domain Integers, as well as records and (multidimensional) arrays of data variables, action channels, and clocks. In UPPAAL it is also possible to declare functions defined a C-like programming language that can be sequentially composed with the resets r of the edges. The programming language allows for branching with *if/then/else* statements, *for*, *while* and *do/while* loops, and a *return* statement. We refer the reader to the online help available on the UPPAAL homepage [1] for more information about this feature.

---

[1] The UPPAAL home page is located at www.uppaal.com.

4. Transformation from COMDES-II to UPPAAL

The introduction in the previous two sections has shown that *processes* and *timed automata* in UPPAAL may act as the basic architectural elements to which *actors* and *state machine FBs* in COMDES-II can be anchored. However, the scheduling policy of actors and the operational semantics of state machine FBs differ from their counterparts in UPPAAL in all aspects listed in Table 1, which requires an extensive model transformation be performed at the meta-level to bridge the semantic gaps between two languages. In this section we will show where these gaps are located and how we bridge them.

Table 1. Behavioral differences between COMDES-II and UPPAAL

Behavioral aspects	COMDES-II	UPPAAL
Concurrency	Fixed-priority preemptive scheduling of actors with non-blocking read-do-write semantics	Interleaving parallelism of timed automata processes
Time	Execution of timed I/O activities over discrete-time	Continuous real-time
Reactive behavior	State machine FB as introduced in Section 2	Timed automata as introduced in Section 3

4.1 Transformation of Concurrency and Time

The preemptive timed multitasking (TM) scheduling policy of actors in COMDES-II is principally different from the interleaving parallelism of UPPAAL processes as defined in CCS. The key factor to overcome concurrency differences between two kinds of systems is to identify how to model the discrete-time scheduler that controls the actor execution status in UPPAAL. In order to achieve this objective, the following four modeling procedures should be accomplished successively, and our solutions will be presented step-by-step in the posterior subsections.

- Finding out a way to model actors and represent their execution information
- Modeling actor interaction following an asynchronous producer-consumer communication protocol
- Establishing a method to manage the non-blocking read-do-write concurrency of actors with preemption
- Modeling the discrete-time scheduler based on the previous three steps

4.1.1 Modeling Actors in UPPAAL

The actor model was briefly introduced in Section 2, from which we can see that the *read* (input drivers), *do* (actor task) and *write* (output drivers) actions of a specific actor will be performed in an ordered sequence within non-successive timing phases (see Fig. 3). Hence from a temporal point of view, it is natural to separately model these three kinds of actor behaviors using different software artifacts so that they can be easily controlled by the time-triggered scheduler.

In UPPAAL, the actor tasks are specified by the corresponding task control blocks containing all the information needed for scheduling tasks execution. The task control block is a data structure defined as following:

```
typedef struct{

          int[0,4] status;
          meta int period;
          meta int executionTime;
          meta int deadline;
          meta int mode;
          bool modeUpdated;
          int timeSinceReleased;
          int computationTimer;

}TTask;
```

Where `status` is a bounded-value integer ([0, 4]) denoting the execution status of a given task. A task could be in `READY` (0), `ACTIVE` (1), `COMPUTED` (2), `FINISHED` (3) or `ERROR` (4) status which are determined and updated by the system scheduler. For a better understanding, Fig. 6 conceptually illustrates the status transition graph of a specific task over discrete-time, whose concrete meaning and determination strategy will subsequently be explained in Section 4.1.3. The three integers `period`, `executionTime` and `deadline` represent the execution period, worst-case execution time and deadline of a specific task. These three parameters remain unchanged during system execution, as a result they are declared as the *meta* integers whose values will be used in the task execution and scheduling, but will not be recorded in the verification state space. Another meta integer `mode` indicates the currently active control state of a task (e.g. product_ready, pre_processing, etc.), and the Boolean variable `modeUpdated` is used to denote if a state transition happens or not in the corresponding task state machine in the current cycle of execution. These two variables control the generation of task outputs associated with the current state at the deadline if a state transition has taken place. The integer `timeSinceReleased` represents the discrete-time which has elapsed since a specific task is released: upon release of the given task, this variable is reset (see Fig. 6), and will be incremented each time the scheduler is executed. The integer variable `computationTimer` is a timer used to record the time left for a task computation. This variable will be set to the value of `executionTime` when a task is released (see Fig. 6), and count down if the task is the highest priority `ACTIVE` task each time the scheduler is executed. The value of that timer will be used to determine

the task status as well as for schedulability analysis. A detailed explanation is given in Section 4.1.3.

The task control block `TTask` can be instantiated into an array of tasks to specify their execution and scheduling information, and the array index (starting from 1) corresponds to the priority of each task: the higher index, the higher priority.

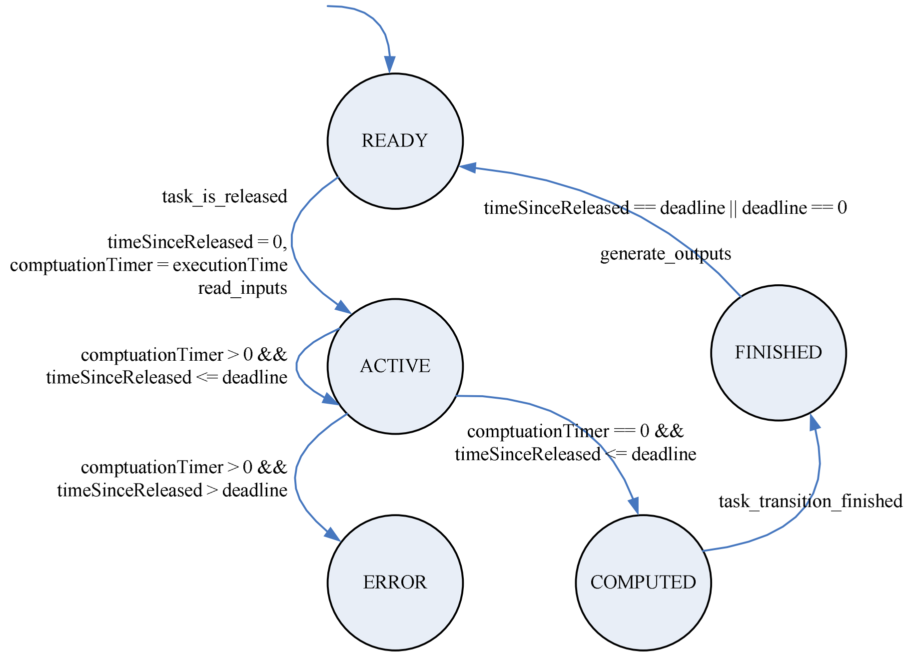

Fig. 6. Status transition graph of actor tasks

Actor input/output drivers will be implemented within two functions in UPPAAL: `taskInputDrivers(int taskID)` and `taskOutputDrivers(int taskID)`. These two functions are application-dependent, and when a specific task *i* is released or its deadline expires, they (`taskInputDrivers(i)` and `taskOutputDrivers(i)`) will be invoked and executed atomically to exchange the information with other tasks, as described in the next section.

4.1.2 Modeling Actor Interaction in UPPAAL

Communication between actors in COMDES-II follows an asynchronous producer-consumer protocol with signal-based non-blocking semantics. Signals are labeled messages containing process data, while in UPPAAL a hand-shaking interaction mechanism is adopted to primarily synchronize the actions between automata processes as defined in CCS parallelism, and no data is exchanged between processes.

A natural way for solving this problem is to model CODEMS-II communication mechanism in UPPAAL through shared variables and data structures. Information

between processes is exchanged by updating and reading from these global resources, where the data race problem is settled by following the COMDES-II semantics:

- When multiple tasks are released, or their deadlines expire simultaneously, the corresponding I/O functions will be invoked and executed sequentially according to the order of task priorities.
- If the deadline of task *i* expires at the same instant as task *j* is released ($i \neq j$), then the output action `taskOutputDrivers(i)`of task *i* will be performed by preceding the input action `taskInputDrivers(j)`of task *j*, regardless the order of *i* and *j*. This rule guarantees that the task *j* can always use the latest data as computed by task *i*, if the interaction ($i \rightarrow j$) happens.

4.1.3 Modeling Actor Concurrency in UPPAAL

Unlike in UPPAAL where the time point of state transitions can be precisely captured by real-time clocks, in TM model of computation it is hard to predict when the actor state transitions will actually happen, but fortunately we do not need to know that either. This is because the determinism of temporal behavior of an actor is enabled by the *read*/*write* actions performed at precisely specified time instants, such that the state transitions of an actor (*do*) may logically happen at any instant confined by its activation instant and deadline, without any side effect to the interaction between actors. Based on this semantics, we model the COMDES-II actor concurrency in UPPAAL by adopting the following abstractions and assumptions.

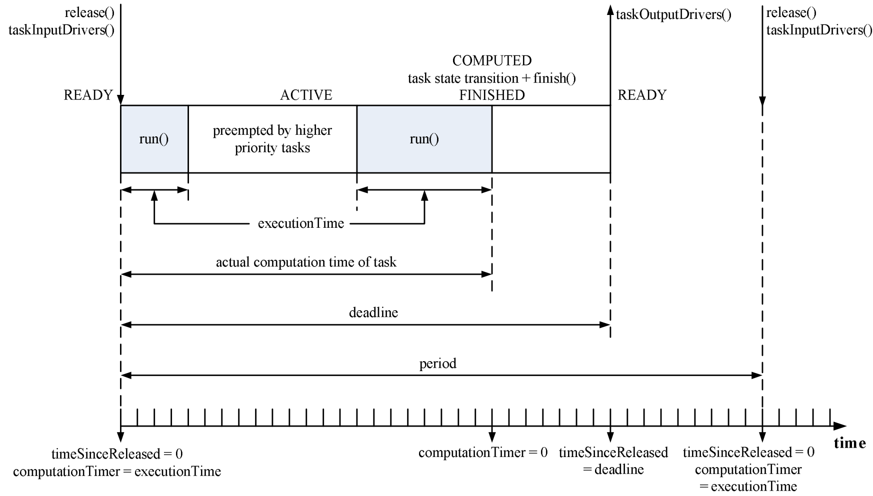

Fig. 7. Actor concurrency over discrete-time in UPPAAL

An actor task may be conceptually in `READY`, `ACTIVE`, `COMPUTED` and `FINISHED` status if all tasks are schedulable, as illustrated in Fig. 7, otherwise the actor task will be in `ERROR` status. In which `READY` means that a task is ready for activation. Status `ACTIVE` denotes that a task has been released, but not completed its

computation yet. In a system it is always the highest-priority `ACTIVE` task to be running, i.e. the corresponding task `computationTimer` decrements with the invocation period of the scheduler. When the `computationTimer` of a specific `ACTIVE` task reduces to zero before its deadline, the task status will be set as `COMPUTED`, meaning that the computation effort has completed and the actor state transition may take place instantaneously, which is followed by the `FINISHED` status. The `FINISHED` status indicates that a particular task computation and control activities have already finished such that the output signals are available for generation, when the associated deadline expires. If the `computationTimer` of an `ACTIVE` task is greater than zero (the task has not finished its computation) when its deadline comes, the task will be scheduled into the `ERROR` status.

Manipulation of the task execution status can be accomplished by invoking a number of scheduling primitives implemented in UPPAAL, including `release()`, `run()`, `finish()`, `outputAction()` and `inputAction()`. These primitives mimic their counterparts in COMDES-II with the following design philosophy:

– `release(int taskID)` takes an integer `taskID` as its argument and will be invoked when the activation condition of a specific `READY` task *i* becomes true (`release(i)`). This primitive will set the released task status as `ACTIVE`, reset the `timeSinceReleased` entry in the corresponding task control block and initialize the `computationTimer` with the value of `executionTime` (see Fig. 7).
– `run()` is invoked in every cycle of scheduler execution. This primitive polls the status of all tasks ordered by their priorities from high to low. Once the highest priority `ACTIVE` task is detected, its `computationTimer` is decremented by the value of scheduler execution period. If the `computationTimer` reduces to be zero (i.e. the task completes its computation effort), the task status will be set as `COMPUTED` meaning that the task state transition can now take place, and then the primitive is exited.
– `finish(int taskID, int mode, bool modeUpdated)` will be invoked immediately when a specific `COMPUTED` task *i* finishes its state transition activity. The primitive records the current operation state as well as the state updated information of task *i* into the corresponding entries of the task control block, through its three arguments, and then set the task status as `FINISHED` (see Fig. 7). In case that the deadline of a specific task *j* is not specified (i.e. deadline = 0), the task output drivers (`taskOutputDrivers(j)`) will be immediately executed within the `finish()` primitive to generate the output signals, afterwards the status of task *j* is set as `READY` (see e.g. Fig. 6).
– `outputAction()` is a primitive used to detect the deadline instant of each task and perform the associated output actions. This primitive is invoked in each cycle of scheduler execution and compares the `timeSinceReleased` of each task with its `deadline` parameter if it is greater than zero (i.e. the task deadline is specified). Once the `timeSinceReleased` of a given task *i* equals to its `deadline`, the task `computationTimer` will be checked to see whether the computation time violates the deadline or not. In case that task is schedulable (i.e. `computationTimer` is zero) and task has `FINISHED` its control activities, the task output drivers (`taskOutputDrivers(i)`) will be executed to generate the

output signals, followed by a change of task status to be READY. If a task is non-schedulable, the task status will be set as ERROR.

– inputAction() principally takes care of the releasing and input actions for periodic tasks (i.e. period > 0). This primitive will be invoked by the time-triggered scheduler to check if timeSinceReleased of a given periodic task *i* is equal to its period or not. If the activation instant has not been reached, the timeSinceReleased value will be incremented. Otherwise two conditions should be considered: 1) If the deadline of task *i* is zero, its computationTimer will be firstly checked to determine the schedulability of task *i*, which is true *iff* the task computation has completed before the activation instant. In case that task *i* is schedulable, it will be released by invoking the release(i) primitive and its input signals sampled via the execution of taskInputDrivers(i); otherwise the task status will be set as ERROR. 2) If the deadline of task *i* is greater than zero, it will directly be released its input signals acquired via the associated primitives.

4.1.4 Modeling Discrete-Time Scheduler in UPPAAL

The approach to modeling COMDES-II actor concurrency as C-like programs in UPPAAL largely eases the design effort of discrete-time scheduler, which is modeled as a timed automaton as shown below:

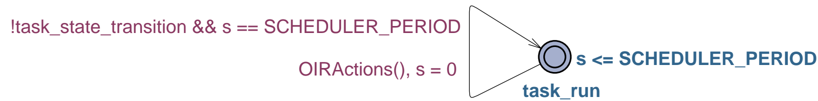

Fig. 8. Discrete-time scheduler automaton in UPPAAL

The scheduler contains only one location called task_run, and one edge guarded by !task_state_transition && s == SCHEDULER_PERIOD. In the edge guard, s is a clock variable that evolves autonomously in real time, and the value of s is confined by SCHEDULER_PERIOD as an invariant condition specified in task_run location. SCHEDULER_PERIOD is an integer constant denoting the execution period of scheduler, which is calculated as the *greatest common divisor* (GCD) of the non-zero period, executionTime and deadline of all tasks. A global Boolean variable task_state_transition is used to guarantee the system behavioral determinism if the scheduler state transition and a task state transition are both fired simultaneously: when an ACTIVE task completes its computation and is then in the COMPUTED status, the task_state_transition will be set as true to indicate that at the moment a task state transition should happen in the system. If in the meanwhile s is equal to SCHEDULER_PERIOD, then the scheduler behavior will be preceded by the task state transition via guarding condition !task_state_transition. When the COMPUTED task finishes its state transition, the finish() primitive will be invoked to reset the task_state_transition variable instantaneously such that the disabled scheduler state transition is enabled. In this approach, non-deterministic concurrent state transitions are equivalently converted into deterministic sequential steps which

execute logically in *zero* time, resulting in considerably reduced state space as only one deterministic state transition pattern is possible during verification.

Two actions will be performed when the execution period of scheduler has come and no task state transition in the system happens: `OIRActions()` and `s = 0`. In which `OIRActions()` is a function simply encapsulating the `outputAction()`, `intputAction()` and `run()` primitives to fulfill the corresponding scheduling functionality in a sequential order. And `s = 0` resets the value of real-time clock so that it is ready to count the time for next cycle of scheduler execution. As to the `task_run` location, it means that when scheduler functionality is not executed, the tasks are allowed to be running to perform computation and control activities.

4.2 Transformation of Functionality

In COMDES-II the functional behavior of a system is described as a composition of different kinds of function blocks (FBs) which are intrinsically independent of the scheduling and timing issues specified at the actor level, hence system functionality can be directly transformed via FBs, regardless of the actor concurrency and time.

We model the COMDES-II basic, composite and modal FBs as functions handling integer variables or data structures in the transformed UPPAAL models. In particular, the basic and/or composite FBs preprocessing the event/guard signals for a state machine FB can be implemented as UPPAAL functions, and invoked in `taskInputDrivers(int taskID)` primitive when the host actor is activated. The modal FBs may be treated in a similar way but will be executed in `taskOutputDrivers(int taskID)` primitive when the host actor deadline expires. To this end, the *state* and *state_updated* information as required by a modal FB can be obtained from the `mode` and `mode_updated` entries of the task control block instance indexed by `taskID`.

On the other hand, system reactive behavior specified by the state machine FBs can be transformed into equivalent timed automata in UPPAAL *without timing annotations*, since time is not involved in the transformation of functionality.

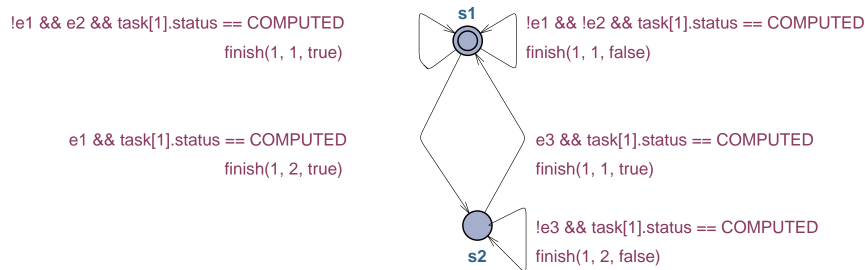

Fig. 9. UPPAAL automaton equivalent to SMFB_1

An UPPAAL automaton with the equivalent semantics to *SMFB_1* illustrated in Fig. 5 is established as in Fig. 9, based on the assumptions and design philosophy

described in previous sections. The model also contains two locations *s1* and *s2*, and a number of transition edges tagged with transition guards and update actions.

A transition guard basically consists of two parts: one is the application-specific event, e.g. *e1* in the transition from *s1* to *s2*. The other one is a condition expression justifying if the status of host actor task is `COMPUTED` or not, as explained in Section 4.1.3. The condition expression `task[1].status == COMPUTED` is labeled in all transitions, in which `task[]` is an array of instances of the task control block, and `1` indicates the task priority.

In case that a transition edge is enabled, the `finish()` primitive is invoked as the update action to perform some finishing activities. For instance in the transition from *s1* to *s2*, `finish(1, 2, true)` will register current state index (`2`) and state updating information (`true`) into the corresponding entries (say, `mode` and `modeUpdated`) of task control block `task[1]`, and then change the task status to be `FINISHED`. All the recorded information will be used to help generate correct output signals at the task deadline instant.

The transition order can be determined by explicitly complementing Boolean event values in the transition guards, as exemplified by two of three outgoing transitions associated with *s1*: from *s1* to *s2*, *e1* guards the transition, while *!e1 && e2* in the self-loop transition of *s1* explicitly specifies that it can only be true after *e1* has firstly been evaluated as *false*, and *e2* is true. An additional self-loop transition of *s1* guarded by *!e1 && !e2* is used to overcome the blocking semantics of UPPAAL automata: assuming that the current location is *s1* and both event signals (*e1* and *e2*) are *false* such that no actual state transitions can happen, the automaton behavior would not be blocked, but rather this transition will be fired to notify the task control block that actor state remains unchanged via the primitive `finish(1, 1, false)`. The techniques enabling ordered and non-blocking transitions are applied to all control states in an automaton, as illustrated in Fig. 9.

Based on the model transformation principles described in this section, the semantics in concurrency, time and functionality aspects of a COMDES-II system can now be safely anchored onto an UPPAAL model, and then verified against desired timing and functional requirements. In next Section we will present a turntable case study to show how we apply this transformational analysis approach in practice.

5. Turntable Case Study

The verification technique is illustrated by a case study, which is based on the turntable system – an example of a real-life manufacturing system that is used for (real-time) control research [9, 10].

The system consists of a turntable, drill, clamp and a tester, as shown in Fig. 10. The turntable transports products in four slots from the input position (0) to the drill/clamp (1), tester (2), and finally to the output position (3). Each slot can hold at most one product, which enters the process in position 0 and leaves in position 3. In position 1 the product is locked by the clamp, a hole is made by the drill, and clamp unlocks the product. The drilling process is checked by the tester in position 2, which tests the depth of the hole, since it is possible that the drilling went wrong.

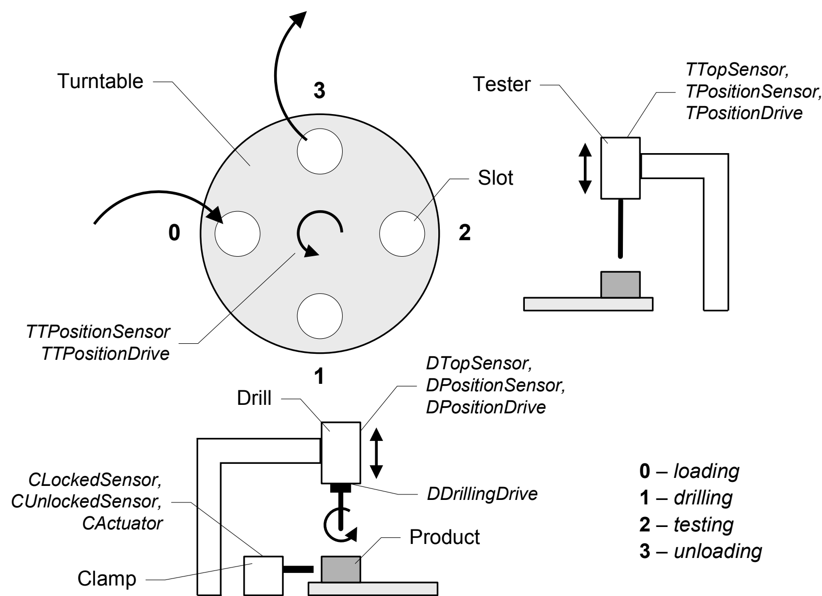

Fig. 10. The turntable system setup

In order to perform the operation of the system, a number of sensors and actuators are used, as summarized in Table 2 and shown in Fig. 10.

Table 2. Sensors and actuators of the system

Sensors	
TTPositionSensor	indicates position (rotation) of the TurnTable
CLockedSensor	shows if Clamp is locked
CUnlockedSensor	shows if Clamp is unlocked

DTopSensor	true if Drill is in top position
DPositionSensor	indicates position of the Drill
TTopSensor	true if Tester is in top position
TPositionSensor	indicates position of the Tester
Actuators	
TTPositionDrive	rotates TurnTable in counter-clockwise direction
CActuator	lockes/unlocks product during drilling
DPositionDrive	moves Drill up/down
DDrillingDrive	makes actual hole
TPositionDrive	moves Tester up/down

5.1 COMDES-II Design of Turntable System

For the purpose of the case study, the turntable system has been designed in COMDES-II. The top-level system design is presented by an actor diagram shown as in Fig. 11.

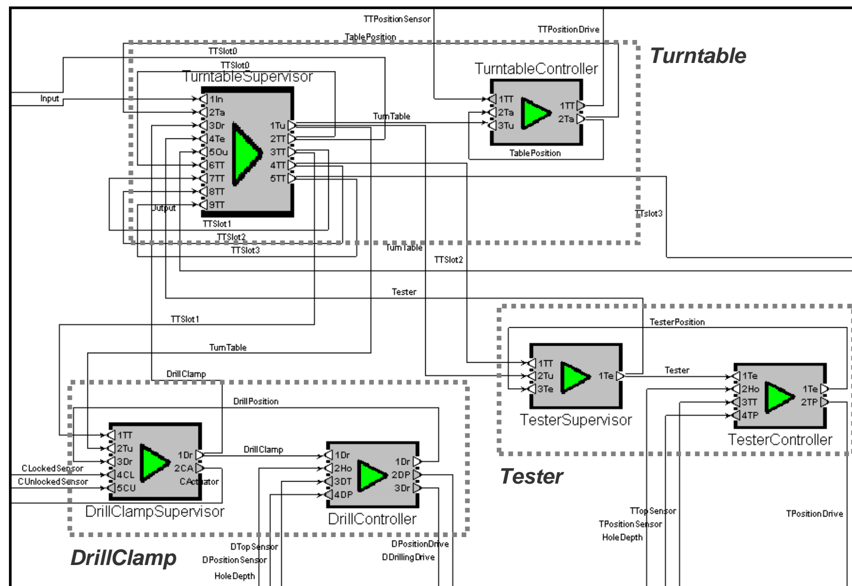

Fig. 11. System design illustrated by an actor diagram

The designed system consists of 6 actors grouped in 3 subsystems: *Turntable*, *DrillClamp* and *Tester* subsystems. Each subsystem is built up from a supervisor and a controller actor to delegate functionality of a subsystem with different dynamics to appropriate body. Therefore, real-time operations are performed by controllers executed periodically, whereas overall tasks coordination is achieved by supervisors

that could be invoked for execution either by timing or external events. However, in this particular design, all actors are executed periodically, where controllers have relatively small periods of the order of few milliseconds, whereas supervisors – relatively big – around 100 ms. The period, worst case execution time (WCET) and deadline of each actor are described as in Table 3 according to priority orders.

Table 3. Execution information of actors

Actor	priority	Period	WCET	Deadline
TesterSupervisor	1	100	2	0
DrillClampSupervisor	2	100	2	0
TurntableSupervisor	3	100	2	0
TurntableController	4	10	1	10
DrillController	5	10	1	10
TesterController	6	10	1	10

Actors interact by exchanging physical signals with an environment via sensors and actuators (see Table 2), as well as by exchanging messages (communication signals), which are presented in Table 4.

Table 4. Messages exchanged among actors

Message name.field	Type	Values	Description
TurnTable.state	enum	ready, loading, loaded, drilling, drilled, testing, testOK, testBad, unloading, unloaded, rotating, rotated	Denotes states of the turntable
TurnTable.updated	bool	true/false	Indicates an update of turntable state
TTSlot0.state	enum	empty, loaded, drilled, testedOK, testedBad	Reports state of product in position 0
TTSlot1.state	enum	empty, loaded, drilled, testedOK, testedBad	Reports state of product in position 1
TTSlot2.state	enum	empty, loaded, drilled, testedOK, testedBad	Reports state of product in position 2
TTSlot3.state	enum	empty, loaded, drilled, testedOK, testedBad	Reports state of product in position 3
Input.state	enum	ready, loading, loaded	Denotes phases of product loading
DrillClamp.state	enum	ready, lock, startDrill, drilling, moveUp, stopDrill, unlock, drilled	Denotes phases of product drilling
DrillClamp.updated	bool	true/false	Indicates an update of Drill state
Tester.state	enum	ready, testing, testOK, testBad, tested	Denotes phases of product testing
Tester.updated	bool	true/false	Indicates an update of Tester state

Output.state	enum	ready, markOK, markBad, unloading, unloaded	Denotes phases of product unloading
TablePosition .rotated	bool	true/false	Becames 1 if turntable completed 90° turn
TablePosition .setpoint	int	0-360	Position of turntable in next rotation (degrees °)
TablePosition .rotation	int	0-360	Currrent position of turntable (degrees °)
DrillPosition.down	bool	true/false	True if Drill reaches bottom
DrillPosition.top	bool	true/false	True if Drill reaches top
TesterPosition.down	bool	true/false	True if Tester is at bottom
TesterPosition.top	bool	true/false	True if Tester reaches top
TesterPosition.BAD	bool	true/false	Indicates bad drilling
HoleDepth.setpoint	int	0-100	Setpoint for Drill and Tester indicating drilling and testing depth (mm)

Each actor accepts a set of messages, as well produces some of them to communicate with other actors. For example, the *TurntableSupervisor* actor informs interested actors about slots status in various positions by sending *TTSlotX* messages (where *X* replaces 0-3).

The *Turntable* subsystem consists of two actors: *TurntableSupervisor* and *TurntableController*. The former actor is responsible for supervising of the latter one, as well it coordinates operation of the whole system with other subsystems supervisors.

Each actor has a task encapsulating a function block diagram: in case of the *TurntableSupervisor*, the task consists of set of comparators, state machine and modal function blocks, see Fig. 12 and Fig. 13 for details.

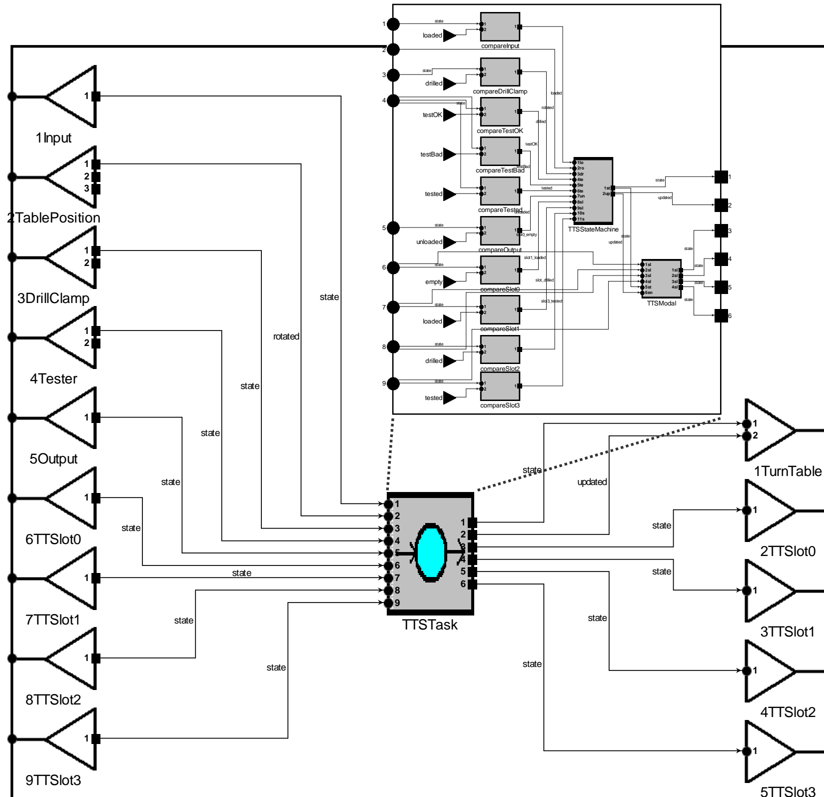

Fig. 12. *TurntableSupervisor* actor

The particular design given here, presents a tutorial/simplified version of the *TurntableSupervisor* with a sequential behavior, however, the real implementation will most likely involve a greater number of state machines such that various processes (loading, drilling, testing, unloading) will be performed concurrently.

The *TTSModal* modal function block (see Fig. 13) changes slot status in its modes according to the *state* signal received from *TTSStateMachine*: *loaded* sets slot 0 to loaded, *drilled* sets slot 1 to drilled, *testOK* sets slot 2 to testedOK, *testBad* sets slot 2 to testedBad, *unloaded* sets slot 3 to empty, *rotated* mode shifts the slot information: from *0* to *1*, *1→2*, *2→3*, *3→0*.

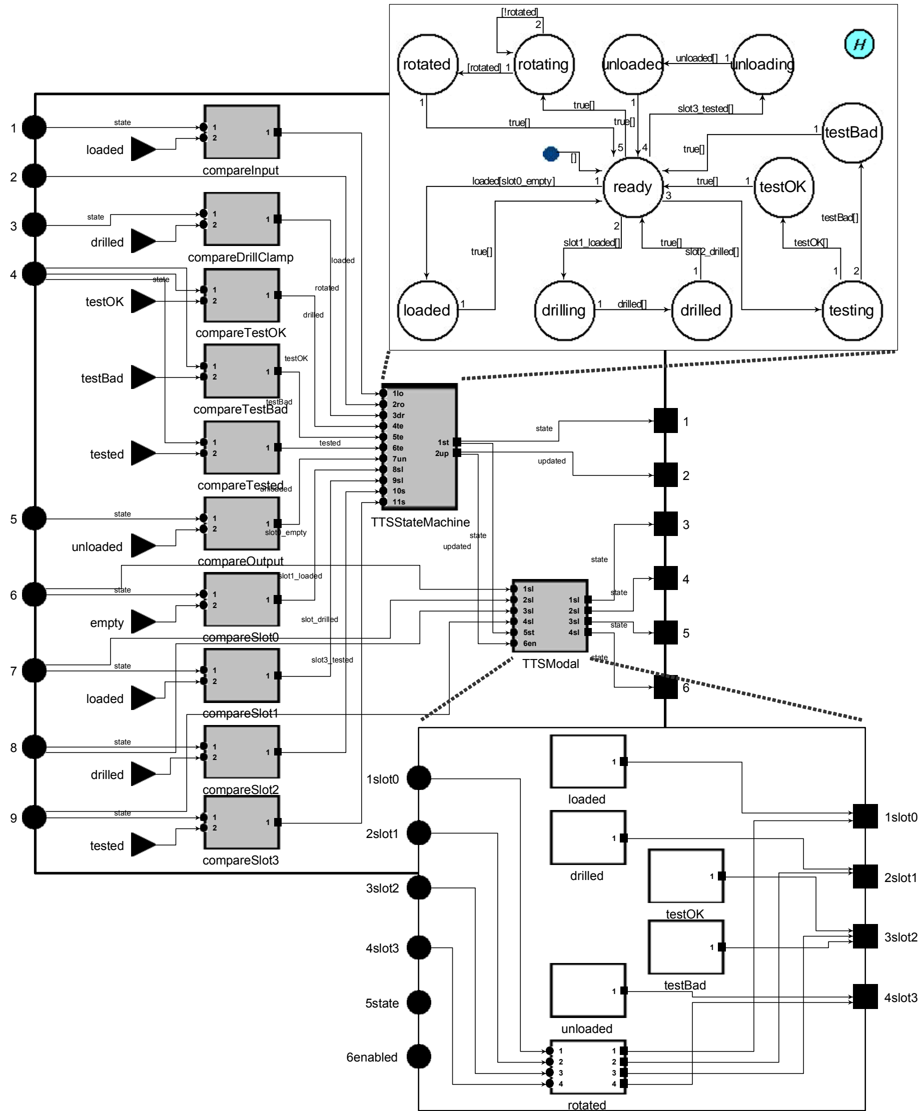

Fig. 13. *TurntableSupervisor* actor task

The *TurntableController* actor task (*TTCTask*) comprises only a modal function block – the *TTCModal*, which is directed by the supervisory state machine of the *TurntableSupervisor*, see Fig. 14. The *TTCModal* has two modes: *rotating* responsible for rotation of the turntable, and *rotated* setting next position to go.

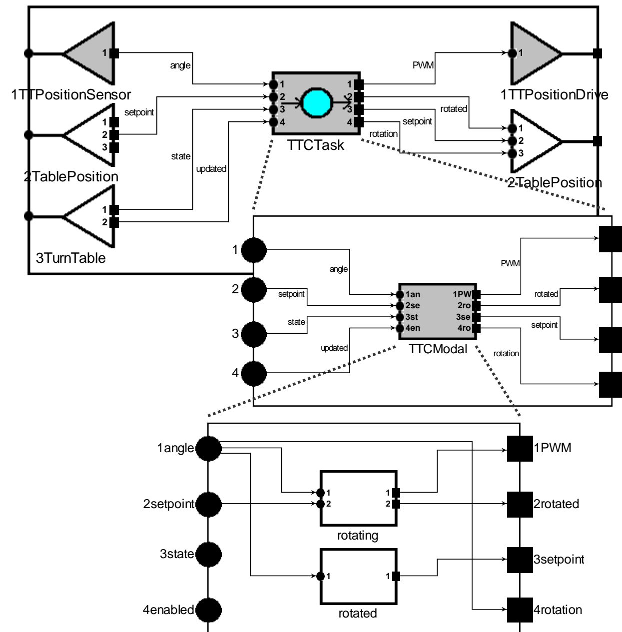

Fig. 14. *TurntableController* actor

Similarly to the *Turntable* subsystem, *DrillClamp* subsystem consists of two actors: *DrillClampSupervisor* and *DrillController*. The supervisor, besides coordinating the subsystem operation, controls also the Clamp, whereas the controller manages drilling process: turning the Drill on/off, moving it up/down.

The *DCSModal* function block of the supervisor generates signal controlling Clamp actuator, more precisely: *lock* tightens the product, *startDrill, drilled* stops the Clamp actuator, *unlock* releases the product.

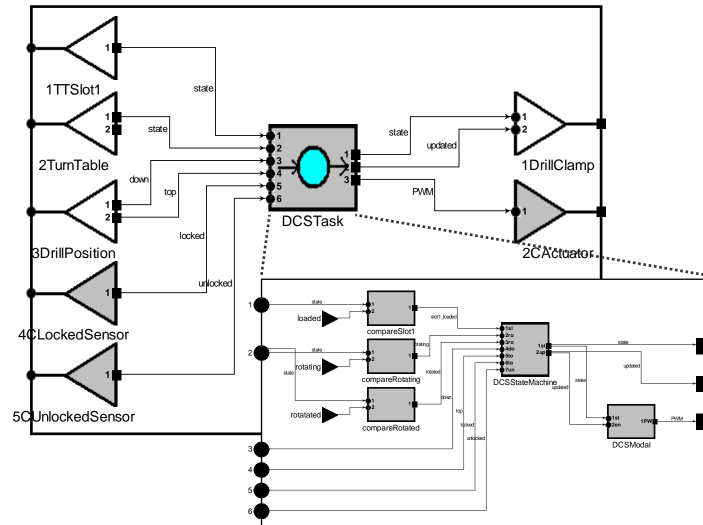

Fig. 15. *DrillClampSupervisor* actor

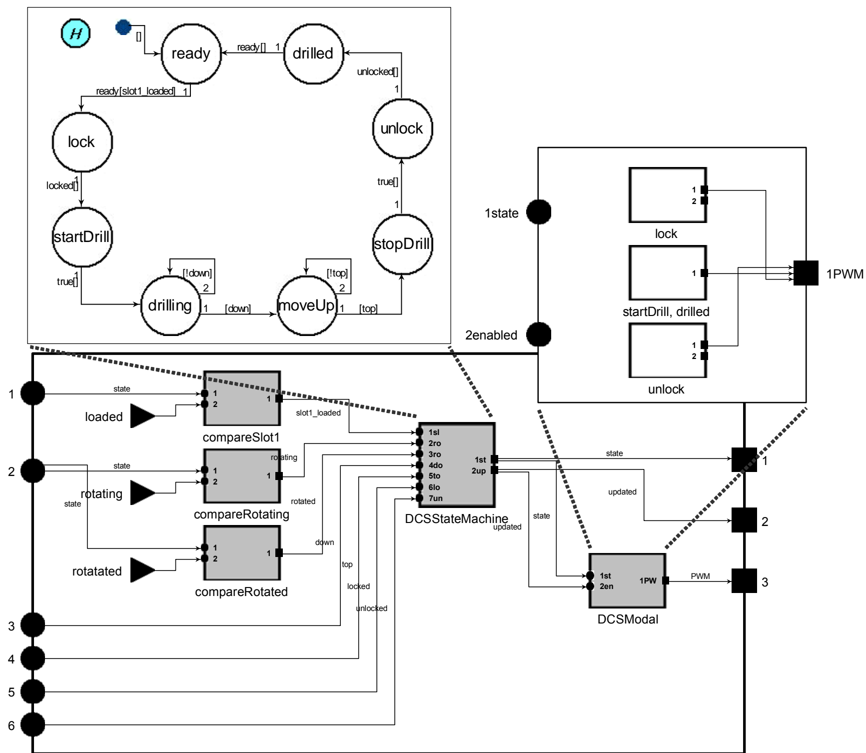

Fig. 16. *DrillClampSupervisor* actor task

The *DrillController* actor task (*DCTask*) comprises one modal function block – the *DCModal*, which is directed by the supervisory state machine of the *DrillClampSupervisor*, see Fig. 17. The *DCModal* has four modes: *drilling* responsible for moving the Drill down, *moveUp* moves the Drill up, *startDrill* turns on the Drill, and finally *stopDrill* turn off the Drill.

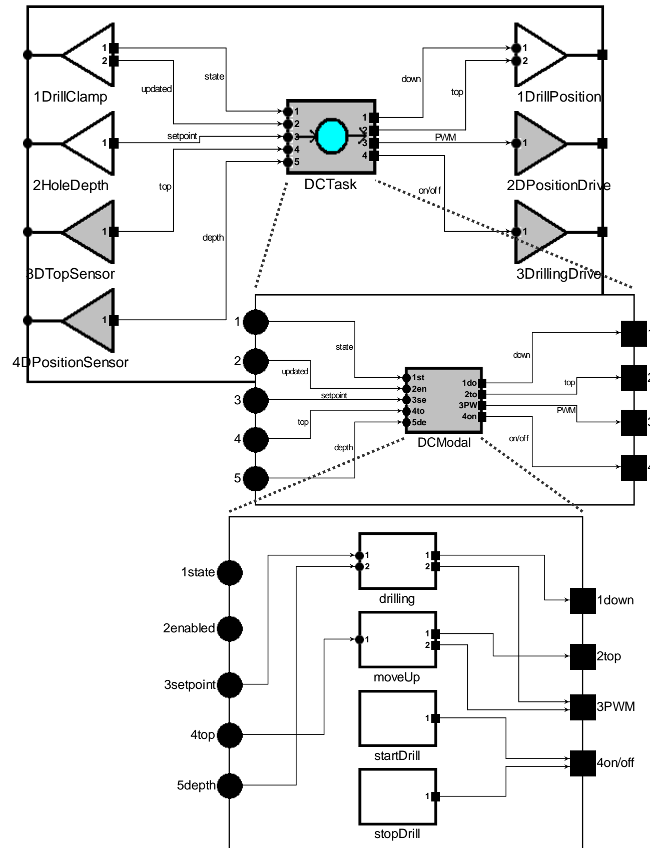

Fig. 17. *DrillController* actor

Tester subsystem consists of two actors: *TesterSupervisor* and *TesterController*, and as previously, the supervisor takes care of the subsystem cooperation with the rest of the system and it directs the controller responsible for controlling the continuous part the subsystem: the Tester drive.

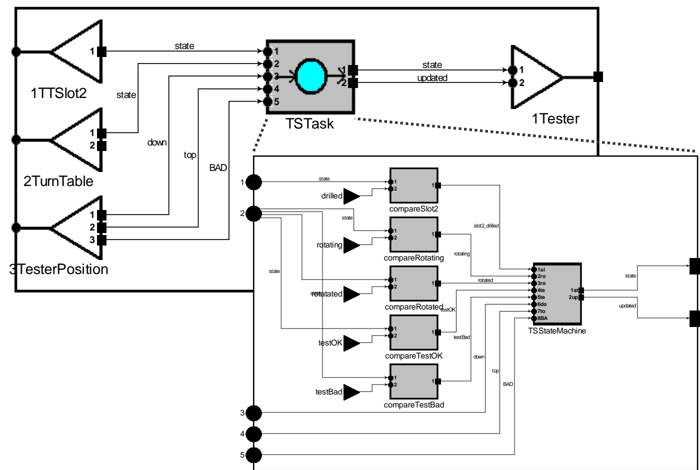

Fig. 18. *TesterSupervisor* actor

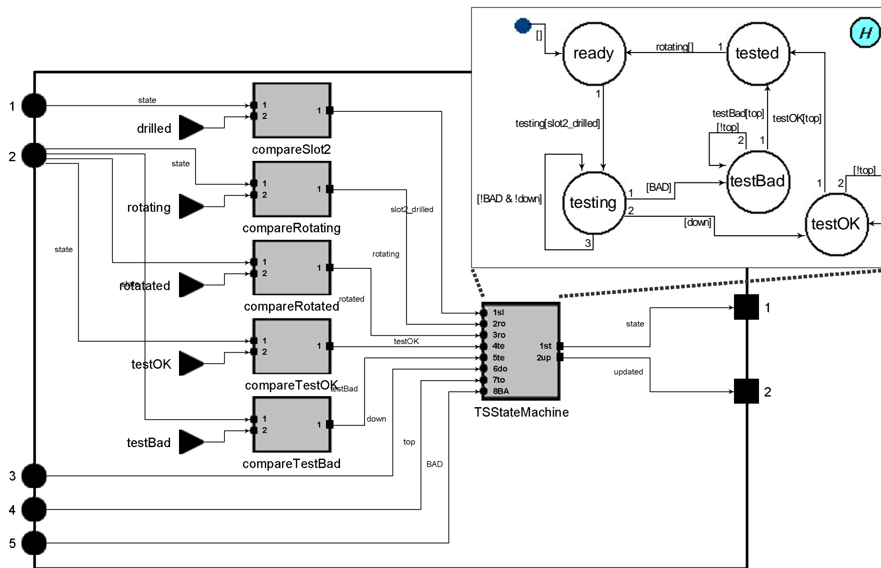

Fig. 19. *TesterSupervisor* actor task

The *TesterController* actor task (*TCTask*) comprises one modal function block – the *TCModal*, which is directed by the supervisory state machine of the

TesterSupervisor, see Figure 20. The *TCModal* has two modes: *testing* responsible for moving the Drill down, and *testOK, testBad* moving the Drill up.

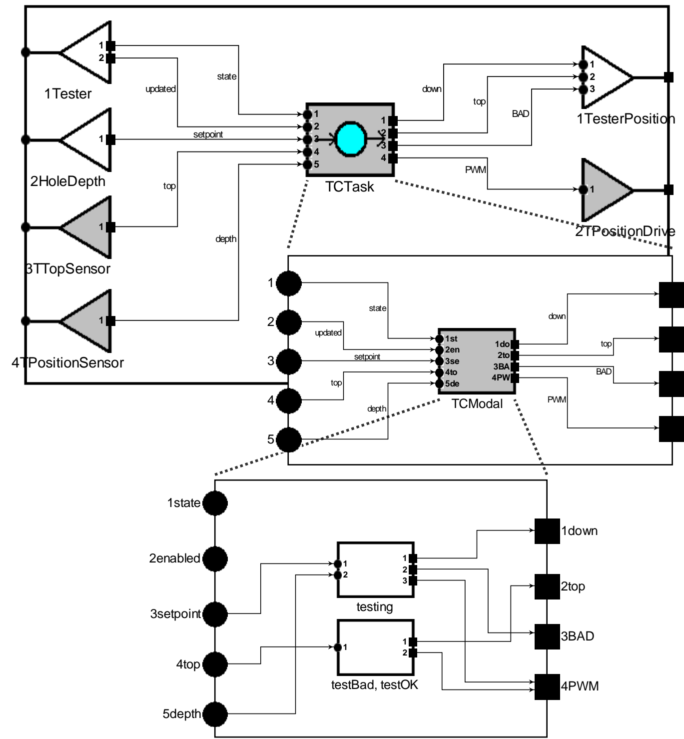

Fig. 20. *TesterController* actor

The design is completed by presenting of *Input* and *Output* environment processes responsible for loading and unloading of products to/from the turntable.

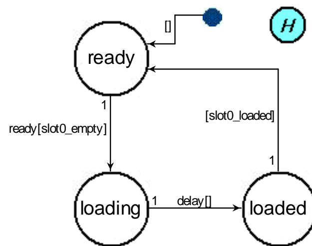

Fig. 21. Behavior of *Input* environment process: product loading

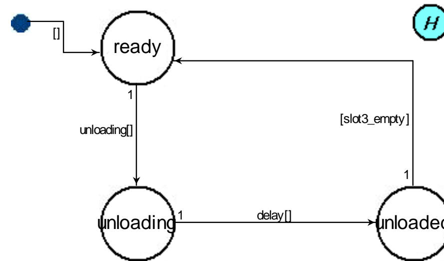

Fig. 22. Behavior of *Output* environment process: product unloading

Given the above design, the system should meet the following properties:

1. *The system has to be schedulable, none of the deadlines is missed (safety)*
2. *The system cannot block, i.e. it does not contain deadlock (safety)*
3. *Every product is drilled (liveness)*
4. *Every product is tested (liveness)*
5. *Product is drilled and tested with the same hole depth (safety)*
6. *Every product leaves the turntable eventually (liveness)*
7. *The turntable does not rotate if any process (loading, drilling, testing, unloading) is in operation (safety)*
8. *No drilling, testing and unloading takes place if there is no product in the slot and no loading is performed if there is a product in the slot (safety)*
9. *The system should not perform redundant operations (safety)*

In order to find answers to the formulated properties a transformation to an analysis model must be conducted – the UPPAAL model, along the path presented in the previous sections of the paper.

5.2 Model Transformation of Turntable Case Study Design

As a real example for illustrating the practical application of aforementioned model transformation techniques, the establishment of an UPPAAL analysis model for Turntable case study in equivalence with its design in COMDES-II is presented in this section, which primarily consists of three parts regarding the separated design concerns:

– *Actor concurrency and timeliness*: instantiation of task control blocks to regulate timed execution and scheduling behavior of the Turntable system actors (see Section 4.1.1).
– *Actor interaction*: 1) definition and instantiation of a number of message data structures in UPPAAL used to exchange information between actors, and 2) implementation of COMDES-II modal function blocks contained in the host Turntable actors as a set of UPPAAL functions, which will be invoked in the

`taskOutputDrivers(int taskID)` primitive to update the message data values (see Section 4.1.2).
– *Actor functionality*: modeling COMDES-II state machine function blocks in the corresponding Turntable actors as different timed automata in UPPAAL, following the transformation approaches described in Section 4.2.

5.2.1 Instantiation of Task Control Blocks

In order to schedule the timed execution of prioritized actors, the task control block defined in Section 4.1.1 is instantiated as below according to the actor execution meta-data summarized in Table 3.

```
// Defining task control block
typedef struct{
    int[0,4] status;
    meta int period;
    meta int executionTime;
    meta int deadline;
    meta int mode;
    bool modeUpdated;
    int timeSinceReleased;
    int computationTimer;
}TTask;

/*****************************************************
    Initializing application-specific tasks
 =====================================================
*****************************************************/
const int TASKS_NUM = 6;
  typedef int[1,TASKS_NUM] t_num;
TTask task[t_num] = {
                      // TesterSupervisor
                {READY, 100, 2, 0, 0, false, 0, 0},
                      //DrillClampSupervisor
                {READY, 100, 2, 0, 0, false, 0, 0},
                      //TurntableSupervisor
                {READY, 100, 2, 0, 0, false, 0, 0},
                      //TurntableController
                {READY, 10, 1, 10, 0, false, 0, 0},
                      //DrillController
                {READY, 10, 1, 10, 0, false, 0, 0},
                      //TesterController
                {READY, 10, 1, 10, 0, false, 0, 0}
};
```

The task control block is instantiated as an array in the length of actor numbers with starting index 1 (e.g. `const int TASKS_NUM = 6; typedef`

int[1,TASKS_NUM] t_num;), and the array elements are ordered according to actor priorities from low to high. In this case study all the tasks are periodic tasks specified by the corresponding non-zero period, executionTime and deadline information. Initially every task is in the READY status, with zero timeSinceReleased and computationTimer.

The given task control block instances provide the central information for discrete-time scheduler (see Section 4.1.4) to control the I/O activities and execution status of actors. For example, the *TesterSupervisor* actor (priority = 1) will be released in every 100 basic timing units, and upon releasing its corresponding input signals can be acquired via taskInputDrivers(int taskID) primitive. During execution its status is controlled by the discrete-time scheduler according to the scheduling principles described in Section 4.1.3. When the corresponding executionTime (2) has been consumed, a control state transition in the timed automata specifying the functional behavior of this actor may happen, and subsequently the output signals will be immediately generated via taskOutputDrivers(int taskID) primitive since its deadline is not specified (deadline = 0), otherwise the outputs will be generated when the deadline expires (e.g. *TesterContrller* actor whose deadline is 10).

5.2.2 Actor Interaction in UPPAAL

The asynchronous producer-consumer communication scheme for COMDES-II actors is achieved in UPPAAL by a number of globally declared message data structures and the related functions responsible for updating message data values. The message data structures are defined in consistency with the information provided from Table 4, while the functions dedicated to updating message content status emulate the functionality of those modal function blocks contained within the system actors, and will be invoked in the taskOutputDrivers(int taskID) primitive when the corresponding actor comes to the time point to generate outputs. An example of TurnTable message is given as below:

```
//Turn table state values
const     int READY          =     0;
const     int LOADING =      1;
const     int LOADED  =      2;
const     int DRILLING       =     3;
const     int DRILLED =      4;
const     int TESTING =      5;
const     int TESTOK  =      6;
const     int TESTBAD =      7;
const     int TESTED  =      8;
const     int UNLOADING      =     9;
const     int UNLOADED       =     10;
const     int ROTATING       =     11;
const     int ROTATED =      12;

//message definition of TurnTable
```

```
typedef struct{
    meta int[READY, ROTATED] status;
    meta bool updated;
}TTurnTable;
TTurnTable TurnTable = {READY, false};
```

The `TurnTable` message instantiates the type of `TTurnTable` which indicates turntable `status` and status `updated` information. Apparently the `TurnTable` is initially in the `READY` status that has not been updated, whose values can be determined by the `TurnTableSupervisorOutput(int taskID)` function that will be executed in the `taskOutputDrivers(int taskID)` primitive when *TurntableSupervisor* actor (`taskID = 3`) has finished its control activity (i.e. `task[3].status == FINISHED`).

This message provides coordinating information for *TurntableController* actor, *DrillClampSupervisor* actor and *TesterSupervisor* actor to control their functional behavior specified as timed automata (see Section 5.2.3), and Table 5 lists the implemented output functions of each system actor.

Table 5. Actor output functions

Actor	Output Function
TesterSupervisor	`void TesterSupervisorOutput(int taskID)`
DrillClampSupervisor	`void DrillClampSupervisorOutput(int taskID)`
TurntableSupervisor	`void TurnTableSupervisorOutput(int taskID)`
TurntableController	`void TurnTableControllerOutput(int taskID)`
DrillController	`void DrillControllerOutput(int taskID)`
TesterController	`void TesterControllerOutput(int taskID)`

For complete details of communication messages and associated functions we refer interested readers to Appendix A.

5.2.3 Actor Functional Behavior in Timed Automata

Among the three most significant model transformation procedures, a correct specification of the actor sequential functionality in UPPAAL timed automata is a key step as it is directly related to the behavior we are interested to analysis – the system sequential behavior. In Section 4.2 a guideline for bridging the semantic gap between COMDES-II state machine function blocks and UPPAAL timed automata has been given, based on which we will exemplify in this section how to equivalently model the actor sequential control behaviors, by using *TurntableSupervisor* actor and *TurntableController* actor as examples.

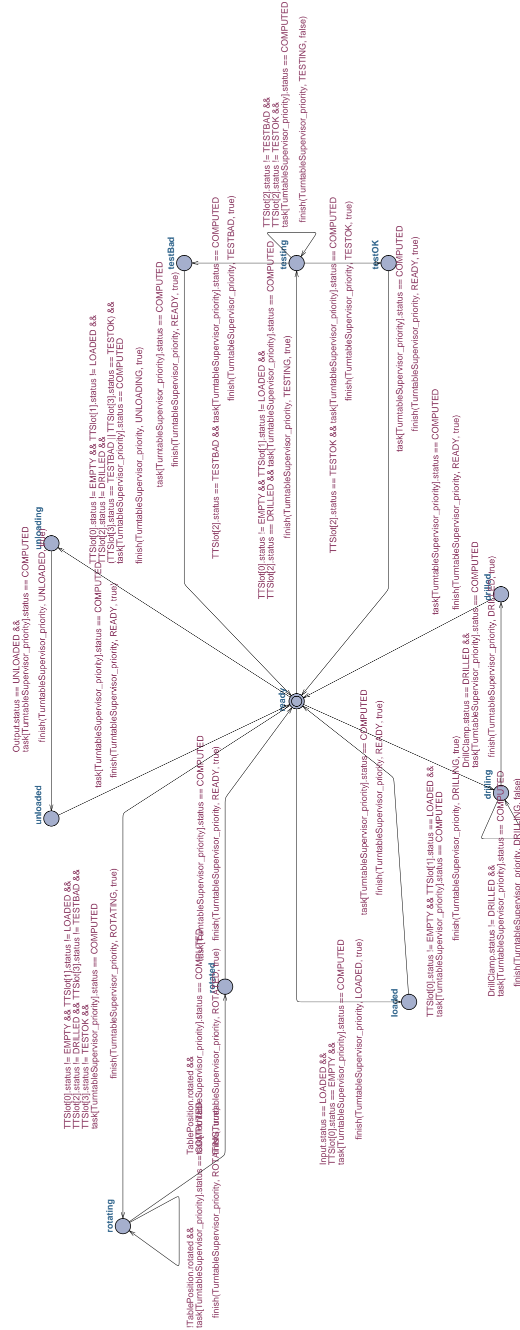

Fig. 23. Timed automata for *TurntableSupervisor* actor functional behavior

For *TTSStateMachine* function block in *TurntableSupervisor* actor (see Fig. 13), its equivalent timed automaton model is presented as in Fig. 23. In this automaton all the transition edges are ordered and only one of them may be fired when the actor execution status is `COMPUTED`, according to the techniques introduced in Section 4.2.

Initially the automaton is in `ready` location, and when the host actor completes its computation (`task[3].status == COMPUTED`), the guards (`Input.status == LOADED && TTSlot[0].status == EMPTY`) associated with the highest order transition (`ready -> loaded`) will be evaluated. In case they are satisfied (i.e. a product is `LOADED` into the input system and slot0 is still `EMPTY`) then this transition is fired to lead the automaton to `loaded` location, where the slot0 status will be changed as `LOADED` (i.e. `TTSlot[0].status = LOADED`). Otherwise the transition guards of `ready -> drilling` will be checked, i.e. if a product has been `LOADED` into slot1 which is ready for drilling (`TTSlot[1].status == LOADED`) `AND` slot0 has finished its loading process (`TTSlot[0].status != EMPTY`), then the automaton will be in the drilling location such that turntable system starts drilling the product in slot1, and so forth (`ready -> testing`, `ready -> unloading`, `ready -> rotating`).

The overall operational behavior of *TurntableSupervisor* actor could be represented by the following repeatedly occurred state transition trace in which each state transition can only take place when the host actor completes its computation (`task[3].status == COMPUTED`), and once a time:

```
ready (TTSlot[0].status == EMPTY, TTSlot[1].status ==
EMPTY, TTSlot[2].status == EMPTY, TTSlot[3].status ==
EMPTY) ->

loaded (TTSlot[0].status == LOADED, TTSlot[1].status ==
EMPTY, TTSlot[2].status == EMPTY, TTSlot[3].status ==
EMPTY) ->

ready -> rotating ->

rotated (TTSlot[0].status == EMPTY, TTSlot[1].status ==
LOADED, TTSlot[2].status == EMPTY, TTSlot[3].status ==
EMPTY) ->

ready ->

loaded (TTSlot[0].status == LOADED, TTSlot[1].status ==
LOADED, TTSlot[2].status == EMPTY, TTSlot[3].status ==
EMPTY) ->

ready -> drilling ->

drilled (TTSlot[0].status == LOADED, TTSlot[1].status
== DRILLED, TTSlot[2].status == EMPTY, TTSlot[3].status
== EMPTY) ->
```

```
ready -> rotating ->

rotated (TTSlot[0].status == EMPTY, TTSlot[1].status ==
LOADED, TTSlot[2].status == DRILLED, TTSlot[3].status
== EMPTY) ->

ready ->

loaded (TTSlot[0].status == LOADED, TTSlot[1].status ==
LOADED, TTSlot[2].status == DRILLED, TTSlot[3].status
== EMPTY) ->

ready -> drilling ->

drilled (TTSlot[0].status == LOADED, TTSlot[1].status
== DRILLED, TTSlot[2].status == DRILELD,
TTSlot[3].status == EMPTY) ->
ready -> testing ->

(testOK || testBad) (TTSlot[0].status == LOADED,
TTSlot[1].status == DRILLED, TTSlot[2].status ==
(TESTOK || TESTBAD), TTSlot[3].status == EMPTY) ->

ready -> rotating ->

rotated (TTSlot[0].status == EMPTY, TTSlot[1].status ==
LOADED, TTSlot[2].status == DRILLED, TTSlot[3].status
== (TESTOK || TESTBAD)) ->

ready ->

loaded (TTSlot[0].status == LOADED, TTSlot[1].status ==
LOADED, TTSlot[2].status == DRILLED, TTSlot[3].status
== (TESTOK || TESTBAD)) ->

ready -> drilling ->

drilled (TTSlot[0].status == LOADED, TTSlot[1].status
== DRILLED, TTSlot[2].status == DRILLED,
TTSlot[3].status == (TESTOK || TESTBAD)) ->

ready -> testing ->

(testOK || testBad) (TTSlot[0].status == LOADED,
TTSlot[1].status == DRILLED, TTSlot[2].status ==
(TESTOK || TESTBAD), TTSlot[3].status == (TESTOK ||
TESTBAD)) ->
```

```
ready -> unloading ->

unloaded (TTSlot[0].status == LOADED, TTSlot[1].status
== DRILLED, TTSlot[2].status == (TESTOK || TESTBAD),
TTSlot[3].status == EMPTY) ->

ready -> rotating ->

rotated (TTSlot[0].status == EMPTY, TTSlot[1].status ==
LOADED, TTSlot[2].status == DRILLED, TTSlot[3].status
== (TESTOK || TESTBAD)) -> ready -> …
```

For *TurntableController* actor (priority = 4), it does not contain any state machine function block, instead only a modal function block called *TTCModal* (see Fig. 14) will be executed periodically with the activation of host actor to control the physical units of turntable, according the operation `state` and `state_updated` information provided from *TurntableSupervisor* actor (see Fig. 11 and Fig. 14). As a result the function behavior of this actor is modeled as in Fig 24:

Fig. 24. Timed automata for *TurntableController* actor functional behavior

This automaton simply consists of one location (`ready`) and one transition edge, which is fired every time the *TurntableController* actor completes its computation (`task[4].status == COMPUTED`), such that the dedicated control behavior can be performed by the `TurnTableControllerOutput(int taskID)` function (see Table 5) that implements the functionality of *TTCModal*, when the corresponding deadline (`deadline = 10`) expires.

The above mentioned modeling mechanisms can be applied to the other Supervisor and Controller actors, and more details are referred to Appendix A.

5.2.4 Formulation of System Properties

Having the complete analysis model of Turntable case study, the last step to final verification of system design is to formulate the desired system requirements as a set of temporal logic properties which can be accepted by the UPPAAL verifier. In Table 6, temporal logic expressions corresponding to the system requirements given in Section 5.1 are listed, together with the verification results as well as memory footprint. All the priorities can be verified within 30s on a computer with Duo CPUs of 1.66 GHz each and 1 GBytes RAM.

Table 6. System properties and verification results

System Requirement	Temporal Logic Formula	Verification Result	Memory Footprint
If slot0 is available, a product will be loaded	`Input.status == READY && TTSlot[0].status == EMPTY --> TTSlot[0].status == LOADED`	Satisfied	18884 Kbytes
Every product is drilled	`TTSlot[0].status == LOADED --> TTSlot[1].status == DRILLED`	Satisfied	23396 Kbytes
Every product is tested	`TTSlot[1].status == DRILLED --> (TTSlot[2].status == TESTOK || TTSlot[2].status == TESTBAD)`	Satisfied	25672 Kbytes
Products are drilled and tested with the same hole depth	`A[] TTSlot[2].status != TESTBAD`	Satisfied	22324 Kbytes
Every product leaves the turntable eventually	`TurnTable.status == LOADED --> TurnTable.status == UNLOADED`	Satisfied	30236 Kbytes
The turntable does not rotate if any process (loading, drilling, testing, unloading) is in operation	`A[] not (TurnTable.status == ROTATING && (Input.status == LOADING || DrillClamp.status == DRILLING || Tester.status == TESTING || Output.status == LOADING))`	Satisfied	28888 Kbytes
No drilling, testing and unloading takes place if there is no product in the corresponding slots and no loading is performed if there is a product in slot0	`A[] not ((Input.status == LOADING && TTSlot[0].status != EMPTY) || (DrillClamp.status == DRILLING && TTSlot[1].status == EMPTY) ||`	Satisfied	23364 Kbytes

	(Tester.status == TESTING && TTSlot[2].status == EMPTY) || (Output.status == UNLOADING && TTSlot[3].status == EMPTY))		
The system should not perform redundant operations	A[] not ((Input.status == LOADING && TTSlot[0].status == LOADED) || (DrillClamp.status == DRILLING && TTSlot[1].status == DRILLED) || (Tester.status == TESTING && (TTSlot[2].status == TESTOK || TTSlot[2].status == TESTBAD)) || (Output.status == UNLOADING && TTSlot[3].status == EMPTY))	Satisfied	23400 Kbytes
The system is deadlock free	A[] not deadlock	Satisfied	23240 Kbytes
Schedulability Analysis	A[] forall(i : int[1,TASKS_NUM]) task[i].status != ERROR	Satisfied	23232 Kbytes

6. Conclusion

The paper has investigated a transformational approach to formal specification and verification of *dynamic behavior* for COMDES-II systems by using UPPAAL, on both theoretical and practical levels. The adopted methodology – semantic anchoring – provides a theoretical foundation for the model transformation that equivalently anchors the behavioral semantics of COMDES-II onto UPPAAL timed automata at the meta-level, which is subsequently instantiated to steer the verification effort of a practical case study – Turntable Case Study designed in COMDES-II.

As a component-based framework intended for model-driven development of real-time embedded software, COMDES-II applies an extensive *separation-of-concerns* approach to model different behavioral concerns, such as concurrency, real-time operation, sequential control behavior combined with continuous computation etc. Specifically, system actors are prioritized and scheduled with a preemptive timed multitasking approach, with I/O activities performed at precisely specified activation and deadline instants. As to functional behavior, the state machine function blocks and modal function blocks are jointly used to specify the system reactive control functionality. However, these behavioral characteristics are completely different from their counterparts in UPPAAL, as summarized in Table 1, Section 4.

In order to bridge the semantic gap, a concrete model transformation procedure is described as in Section 4 by taking into account all the behavioral aspects that would influence the overall system operational semantics, including:

- A task control block is defined to encompass the execution information of actors, such as period, WCET, deadline etc.
- The communication primitives are defined to enable the asynchronous producer-consumer interaction pattern between actors.
- A number of scheduling primitives are implemented such that the discrete-time scheduler is able to manage the preemptive execution of actors with timed I/O activities, based on the execution information specified in the task control block instances.
- A method allowing an equivalent transformation from state machine function blocks to the corresponding timed automata is developed, as a result the actor reactive control behavior can be precisely modeled in UPPAAL.

The above model transformation techniques covering different behavioral concerns are finally applied to verify a practical case study designed in COMDES-II – the turntable control system – against a list of desired system requirements, as described in Section5. The verification results illustrate in a positive way that the developed methods can be used to precisely analyze the schedulability and functional behavior of COMDES-II applications using UPPAAL, with a complete preservation of the original system operational semantics.

References

1. Kai Chen, Janos Sztipanovits and Sandeep Neema: Toward a semantic anchoring infrastructure for domain-specific modeling languages. Proceedings of the 5th ACM international conference on Embedded software, Jersey City, NJ, USA, 2005
2. J. Reekie and E. A. Lee: Lightweight Component Models for Embedded Systems. *Technical Memorandum UCB/ERL M02/30*, University of California, Berkeley, CA 94720, USA, October 30, 2002
3. Xu Ke, Krzysztof Sierszecki, Christo Angelov: COMDES-II: A Component-Based Framework towards Generative Development of Distributed Real-Time Control Systems. Proceedings of the 13 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Daegu, S.Korea, 2007
4. Anantha Narayanan and Gabor Karsai: Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations. Proceedings of the Second International Workshop on Graph and Model Transformation, Brighton, United Kingdom, 2006
5. C. Angelov and J. Berthing: Distributed Timed Multitasking: a Model of Computation for Hard Real-Time Distributed Systems. Proc. of the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems DIPES'06, Braga, Portugal, Oct. 2006
6. Gerd Behrmann, Alexandre David, and Kim G. Larsen: A Tutorial on UPPAAL. In proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04). LNCS 3185
7. Kim G. Larsen, Paul Pettersson and Wang Yi: Uppaal in a Nutshell. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997
8. Robin Milner: Communication and Concurrency. Prentice Hall, 1989. ISBN 0-13-114984-9
9. Bos, V., Kleijn, J.J.T.: Automatic verification of a manufacturing system. Robotics and Computer Integrated Manufacturing, vol. 17, (2001), 185-198
10. Bortnik, E., Trčka, N., Wijsc, A.J., Luttik, B., van de Mortel-Fronczak, J.M., Baeten, J.C.M., Fokkink, W.J., Rooda, J.E.: Analyzing a χ Model of a Turntable System using Spin, CADP and Uppaal. Journal of Logic and Algebraic Programming, vol. 65, (2005), 51-104

Appendix A: UPPAAL Model of Turntable Case Study

Global Declarations:

```
// Place global declarations here.

clock s;

/*******************************************************
    Defining task-related types and macros
*******************************************************/

// Defining task status
const     int READY   = 0;
const     int ACTIVE  = 1;
const     int COMPUTED      = 2;
const     int FINISHED      = 3;
const     int ERROR   = 4;

// Defining maximum task numbers and macro of tasks
const     int MAX_TASKS     = 16;

const     int TASK1   = 1;
const     int TASK2   = 2;
const     int TASK3   = 3;
const     int TASK4   = 4;
const     int TASK5   = 5;
const     int TASK6   = 6;
const     int TASK7   = 7;
const     int TASK8   = 8;
const     int TASK9   = 9;
const     int TASK10  = 10;
const     int TASK11  = 11;
const     int TASK12  = 12;
const     int TASK13  = 13;
const     int TASK14  = 14;
const     int TASK15  = 15;
const     int TASK16  = 16;


// Defining task control block
typedef struct{
    int[0,4] status;
    meta int period;
    meta int executionTime;
```

```
    meta int deadline;
    meta int mode;
    bool modeUpdated;
    int timeSinceReleased;
    int computationTimer;
}TTask;

//broadcast chan not_sch;


/*******************************************************
    Initializing application-specific tasks
=======================================================
******************************************************/
const int TASKS_NUM = 6;
typedef int[1,TASKS_NUM] t_num;

TTask task[t_num] = {
                {READY, 100, 2, 0, 0, false, 0, 0},
                {READY, 100, 2, 0, 0, false, 0, 0},
                {READY, 100, 2, 0, 0, false, 0, 0},
                {READY, 10, 1, 10, 0, false, 0, 0},
                {READY, 10, 1, 10, 0, false, 0, 0},
                {READY, 10, 1, 10, 0, false, 0, 0}
};


/****************************************
    Actors priority definition
****************************************/

const     int TurntableSupervisor_priority   =     3;
const     int TurntableController_priority   =     4;
const     int DrillClampSupervisor_priority  =     2;
const     int DrillController_priority  =     5;
const     int TesterController_priority =     6;
const     int TesterSupervisor_priority =     1;

const int SCHEDULER_PERIOD = 1;
bool task_state_transition = false;



/****************************************
    Declare environment variables Here
****************************************/

//Turn table state values
```

```
//const   int READY   =     0;
const     int LOADING =     1;
const     int LOADED  =     2;
const     int DRILLING      =     3;
const     int DRILLED =     4;
const     int TESTING =     5;
const     int TESTOK  =     6;
const     int TESTBAD =     7;
const     int TESTED  =     8;
const     int UNLOADING     =     9;
const     int UNLOADED      =     10;
const     int ROTATING      =     11;
const     int ROTATED =     12;

//message definition of TurnTable
typedef struct{
    meta int[READY, ROTATED] status;
    meta bool updated;
}TTurnTable;

TTurnTable TurnTable = {READY, false};


// Slots state values
const     int EMPTY   =     13;

//message definition of slots
typedef struct{
    int[LOADED, EMPTY] status;
    int[0,100] depth;
}TTTSlot;

TTTSlot  TTSlot[4]  =  {{EMPTY,  0},  {EMPTY,  0},  {EMPTY,
0}, {EMPTY, 0}};


//message definition of Input from environment
typedef struct{
    meta int[READY, LOADED] status;
}TInput;

TInput Input = {READY};


// DrillClamp state values
const     int LOCK    =     14;
const     int STARTDRILL    =     15;
const     int MOVEUP  =     16;
```

```
const     int STOPDRILL     =     17;
const     int UNLOCK  =     18;

//message definition of DrillClamp
typedef struct{
    meta int[READY, UNLOCK] status;
    meta bool updated;
    int[-1,1] PWM;
}TDrillClamp;

TDrillClamp DrillClamp = {READY, false, 0};


//message definition of Tester
typedef struct{
    meta int[READY, TESTED] status;
    meta bool updated;
    //int[-1,1] PWM;
}TTester;

TTester Tester = {READY, false};


// Environment Output state values
const     int MARKOK  =     19;
const     int MARKBAD =     20;

//message definition of Output to environment
typedef struct{
    meta int[READY, MARKBAD] status;
}TOutput;

TOutput Output = {READY};


//message definition of TablePosition
typedef struct{
    bool rotated;
    meta int[0,360] setpoint;
    int[0,360] rotation;
    //int[0,1] PWM;
}TTablePosition;

TTablePosition TablePosition = {false, 90, 0};


//message definition of DrillPosition
typedef struct{
```

```
    bool top;
    bool down;
}TDrillPosition;

TDrillPosition DrillPosition = {true, false};


//message definition of TesterPosition
typedef struct{
    bool top;
    bool down;
    bool BAD;
}TTesterPosition;

TTesterPosition TesterPosition = {true, false, false};


//message definition of HoleDepth

const     int HoleDepthSetpoint = 30; // Hole depth is
30mm



//message definition of EnvClamp

const     int LOCKED  =     21;
const     int UNLOCKED      =     22;

typedef struct{
    int[LOCKED, UNLOCKED] status;
}TEnvClamp;

TEnvClamp Clamp = {UNLOCKED};


int[0,100] drill_position = 0;



/*******************************************************
Actions performed in output drivers of a specific task
*******************************************************/

void TurnTableSupervisorOutput(int taskID){
    int[LOADED, EMPTY] slotTempStatus;
    int[0,100] slotTempDepth;
    int[0,3] slotIndex;
```

```
      TurnTable.status = task[taskID].mode;
      TurnTable.updated = task[taskID].modeUpdated;
      if(task[taskID].modeUpdated)
      {
            if(task[taskID].mode == LOADED)
            {
                  TTSlot[0].status = LOADED;
            }
            else if(task[taskID].mode == DRILLED)
            {
                  TTSlot[1].status = DRILLED;
            }
            else   if(task[taskID].mode   ==   TESTOK   ||
task[taskID].mode == TESTBAD)
            {
                  //TTSlot[2].status = task[taskID].mode;
            }
            else if(task[taskID].mode == UNLOADED)
            {
                  TTSlot[3].status = EMPTY;
            }
            else if(task[taskID].mode == ROTATED)
            {
                  slotTempStatus = TTSlot[3].status;
                  slotTempDepth = TTSlot[3].depth;
                  for(slotIndex    =    3;    slotIndex>0;
slotIndex--)
                  {
                        TTSlot[slotIndex].status         =
TTSlot[slotIndex-1].status;
                        TTSlot[slotIndex].depth          =
TTSlot[slotIndex-1].depth;
                  }
                  TTSlot[0].status = slotTempStatus;
                  TTSlot[0].depth = slotTempDepth;

                  if(TablePosition.setpoint < 360)
                  {
                        TablePosition.setpoint += 90;
                  }
                  else
                  {
                        TablePosition.setpoint = 90;
                        TablePosition.rotation = 0;
                  }
            }
      }
  }
```

```
  void TurnTableControllerOutput(int taskID){
      if(task[TurntableSupervisor_priority].modeUpdated)
      {
            if    (task[TurntableSupervisor_priority].mode
== ROTATING)
            {
                  if(TablePosition.rotation             ==
TablePosition.setpoint)
                  {
                        TablePosition.rotated = true;
                        //TablePosition.PWM = 0;
                  }
                  else
                  {
                        TablePosition.rotated = false;
                        //TablePosition.PWM = 1;
                        TablePosition.rotation += 90;
                  }
            }
            else                                        if
(task[TurntableSupervisor_priority].mode == ROTATED)
            {

            }
      }
  }

  void DrillClampSupervisorOutput(int taskID){
      DrillClamp.status = task[taskID].mode;
      DrillClamp.updated = task[taskID].modeUpdated;
      if (task[taskID].modeUpdated)
      {
            if(task[taskID].mode == LOCK)
            {
                  Clamp.status = LOCKED;
            }
            else if(task[taskID].mode == UNLOCK)
            {
                  Clamp.status = UNLOCKED;
            }
            else if(task[taskID].mode == DRILLING)
            {
                  if(TTSlot[1].depth                    >=
HoleDepthSetpoint) // Start to stop driller
                  {
                        DrillPosition.down = true;
```

```
                  }
            }
      }
  }

  void DrillControllerOutput(int taskID){
      if(task[DrillClampSupervisor_priority].modeUpdated)
      {
            if(task[DrillClampSupervisor_priority].mode
== DRILLING)
            {
                  DrillPosition.top = false;
                  if(drill_position++ == 10)
                  {
                        TTSlot[1].depth += 1;
                        drill_position = 0;
                  }
            }
            else
if(task[DrillClampSupervisor_priority].mode == MOVEUP)
            {
                  DrillPosition.top = true;
                  DrillPosition.down = false;
            }
      }
  }

  void TesterSupervisorOutput(int taskID){
      Tester.status = task[taskID].mode;
      Tester.updated = task[taskID].modeUpdated;
  }

  void TesterControllerOutput(int taskID){
      if(task[TesterSupervisor_priority].modeUpdated)
      {
            if(task[TesterSupervisor_priority].mode     ==
TESTING)
            {
                  TesterPosition.top = false;
                  if(TTSlot[2].depth                    !=
HoleDepthSetpoint)
                  {
                        TesterPosition.down = false;
                        TesterPosition.BAD = true;
                  }
                  else
                  {
                        TesterPosition.down = true;
```

```
                        TesterPosition.BAD = false;
                  }

            }
            if(task[TesterSupervisor_priority].mode     ==
TESTBAD    ||    task[TesterSupervisor_priority].mode    ==
TESTOK)
            {
                  TesterPosition.top = true;
                  TTSlot[2].status                           =
task[TesterSupervisor_priority].mode;
                  TTSlot[2].depth = 0;
            }
      }
  }


  // Input function of tasks
  void taskInputDrivers(int taskID){
      /*insert code of input drivers here*/

  }

  // Output function of tasks
  void taskOutputDrivers(int taskID){
      /*insert code of output drivers here*/

      if   (taskID   ==   TurntableSupervisor_priority)   //
Turntable Supervisor
      {
            TurnTableSupervisorOutput(taskID);
      }
      else  if(taskID  ==  TurntableController_priority)  //
Turntable Controller
      {
            TurnTableControllerOutput(taskID);
      }
      else  if  (taskID  ==  DrillClampSupervisor_priority)
// DrillClamp Supervisor
      {
            DrillClampSupervisorOutput(taskID);
      }
      else   if(taskID   ==   DrillController_priority)   //
Drill Controller
      {
            DrillControllerOutput(taskID);
      }
```

```
      else  if(taskID  ==  TesterSupervisor_priority)  //
Tester Supervisor
      {
            TesterSupervisorOutput(taskID);
      }
      else  if(taskID  ==  TesterController_priority)  //
Tester Controller
      {
            TesterControllerOutput(taskID);
      }
  }


  /************************************
      Defining task functions
  *************************************/

  // Release a task
  void release(int taskID){
      if(task[taskID].status == READY)
      {
            task[taskID].status = ACTIVE;
            task[taskID].timeSinceReleased = 0;
            task[taskID].computationTimer                =
task[taskID].executionTime;
      }
  }

  // Schedule the highest priority active task to run
  void run(){
      int i = TASKS_NUM;
      for(i; i>0; i--)
      {
            if(task[i].status == ACTIVE)
            {
                  if(task[i].computationTimer != 0)
                  {
                        task[i].computationTimer        -=
SCHEDULER_PERIOD;
                  }
                  if(task[i].computationTimer == 0)
                  {
                        task_state_transition = true;
                        task[i].status = COMPUTED;
                  }
                  return;
            }
      }
```

```
  }

  // When the running task finishes, schedule it back to
READY status
  void finish(int taskID, int mode, bool modeUpdated){
      task[taskID].mode = mode;
      task[taskID].modeUpdated = modeUpdated;
      task[taskID].status = FINISHED;
      if(task[taskID].deadline == 0)
      {
            taskOutputDrivers(taskID);
            task[taskID].status = READY;
      }
      task_state_transition = false;
  }


  /****************************************************
    I/O actions of tasks performed at specific triggering
instant and deadline
  ****************************************************/

  void outputAction(){
      int i = TASKS_NUM;
      for(i; i>0; i--)
      {
            if
(task[i].timeSinceReleased*SCHEDULER_PERIOD             ==
task[i].deadline && task[i].deadline != 0)
            {
                  if(task[i].computationTimer  >  0)  //  if
a non-zero deadline task is not schedulable
                  {
                        task[i].status = ERROR;
                  }
                  else if (task[i].status == FINISHED)
                  {
                        taskOutputDrivers(i);
                        task[i].status = READY;
                  }
            }
      }
  }

  void inputAction(){
      int i = TASKS_NUM;
      for(i; i>0; i--)
      {
```

```
            // input actions for periodic tasks
            if (task[i].period != 0)
            {
                  if  (task[i].timeSinceReleased  ==  0  ||
task[i].timeSinceReleased*SCHEDULER_PERIOD              ==
task[i].period)
                  {
                        if (task[i].deadline == 0)
                        {
                              if (task[i].computationTimer
> 0) // if a zero deadline task is not schedulable
                              {
                                    task[i].status       =
ERROR;
                                    return;
                              }
                        }
                        taskInputDrivers(i);
                        release(i);
                  }
                  task[i].timeSinceReleased++;
            }
            // input actions for aperiodic tasks
            else
            {
                  if(task[i].status     ==     ACTIVE     ||
task[i].status == FINISHED)
                  {
                        task[i].timeSinceReleased++;
                  }
            }
      }
  }

  void OIRActions(){
      outputAction();
      inputAction();
      run();
  }
```

System Declarations:

```
// Place template instantiations here.

TurntableSupervisor_I =
TurntableSupervisor(TurntableSupervisor_priority);

TurntableController_I =
TurntableController(TurntableController_priority);

DrillClampSupervisor_I =
DrillClampSupervisor(DrillClampSupervisor_priority);

DrillController_I =
DrillController(DrillController_priority);

TesterController_I =
TesterController(TesterController_priority);

TesterSupervisor_I =
TesterSupervisor(TesterSupervisor_priority);



// List one or more processes to be composed into a
system.

system TurntableSupervisor_I, TurntableController_I,
DrillClampSupervisor_I, DrillController_I,
TesterController_I, TesterSupervisor_I, EnvInput,
EnvOutput, Scheduler;
```

Timed Automata Models:

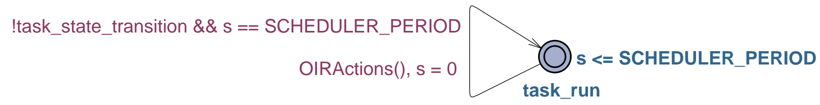

Fig. 25. Timed automata for discrete-time scheduler

Fig. 26. Timed automata for *TurntableController* actor functional behavior

Fig. 27. Timed automata for *DrillController* actor functional behavior

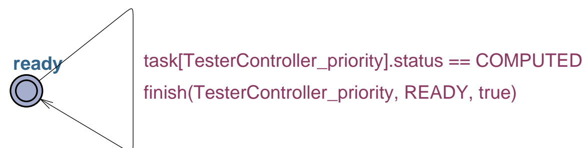

Fig. 28. Timed automata for *TesterController* actor functional behavior

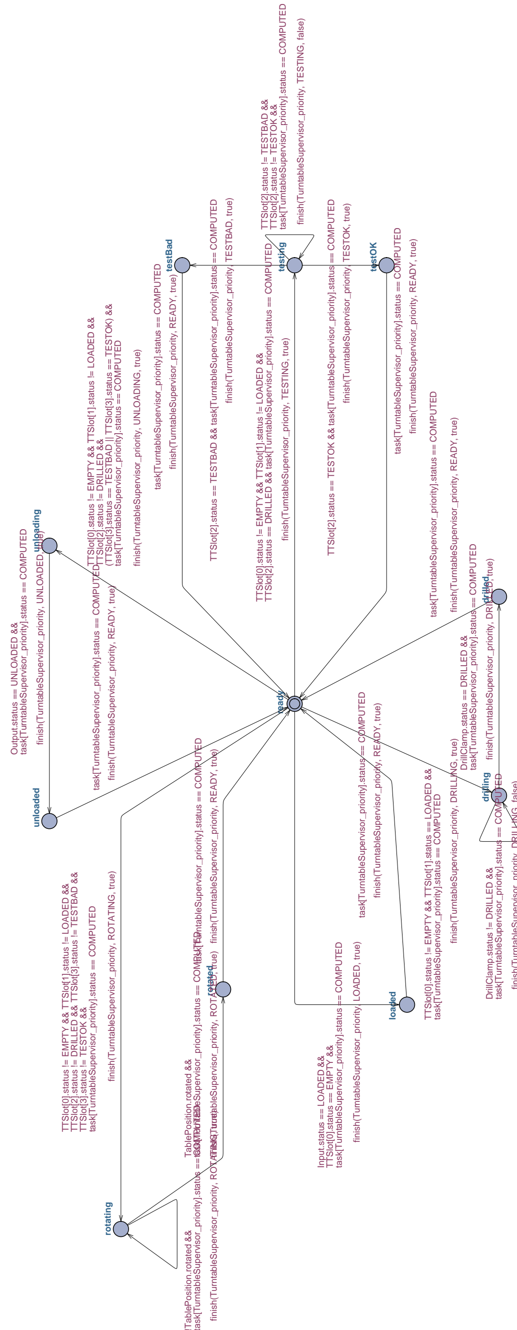

Fig. 29. Timed automata for *TurntableSupervisor* actor functional behavior

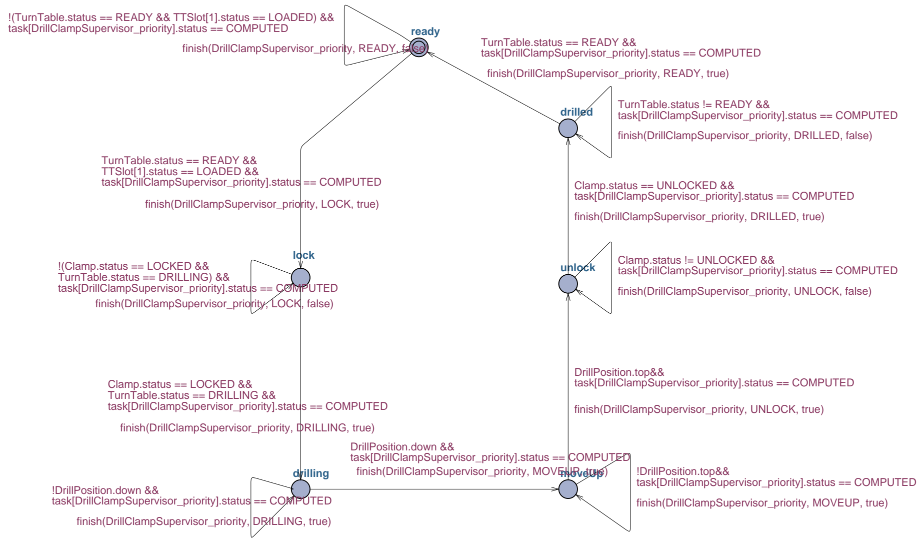

Fig. 30. Timed automata for *DrillClampeSupervisor* actor functional behavior

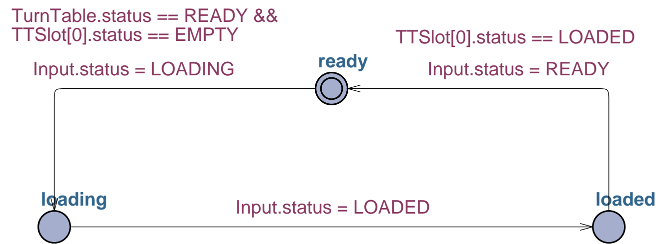

Fig. 31. Timed automata for *Input* system

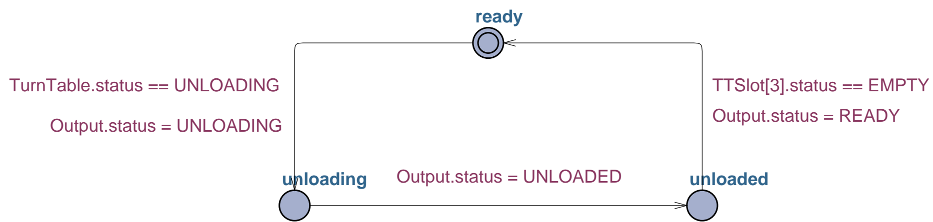

Fig. 32. Timed automata for *Output* system

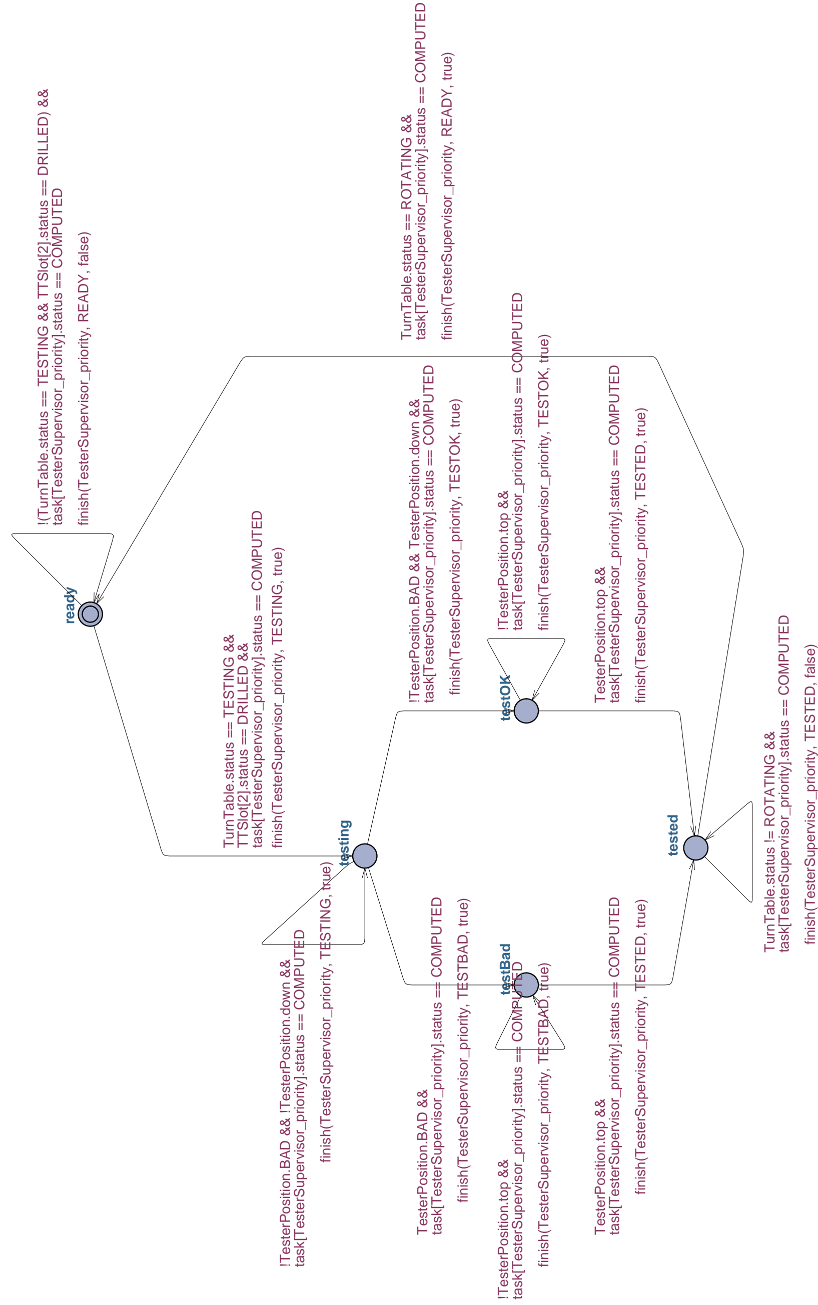

Fig. 33. Timed automata for *TesterSupervisor* actor functional behavior