Verification of COMDES-II Systems Using UPPAAL with Model Transformation

Xu Ke¹, Paul Pettersson², Krzysztof Sierszecki¹, Christo Angelov¹

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 DrillClampSupervisor 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.

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.

Fig. 1. Semantic anchoring process of CODMES-II
Verification of COMDES-II Systems Using UPPAAL with Model Transformation

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.

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.

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 \((\text{read})\) in logically zero time to acquire all input signals which are latched throughout the whole actor execution. The activated actor task will process \((\text{do})\) 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 \((\text{write})\) output signals when the corresponding actor \textit{deadline} expires. If the deadline of an actor is not be specified \((\text{i.e., deadline} = 0)\), the actor output drivers will be immediately executed when the actor task finishes its computation. This \textit{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 \((\text{control-flow})\) combined with multi-mode \((\text{modal})\) 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 \textit{event/guard} inputs, an \textit{event-driven} state machine model, and exactly two outputs: \textit{state} and \textit{stateUpdated}. 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: \textit{state} and \textit{stateUpdated}. Here, \textit{state} represents the currently active state, and \textit{stateUpdated} will be set \textit{true} if a state transition has happened, otherwise it is not blocked, but set as \textit{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 \((\text{states})\) 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 \textit{state} information provided from the supervisory state machine FB, whereas the enabledness of executing state is determined by the \textit{stateUpdated} 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 \textit{SMFB\_1} is illustrated as in Fig. 5. The \textit{SMFB\_1} contains three binary event inputs \(e1\), \(e2\) and \(e3\), an event-driven state machine model, and exactly two outputs: \textit{state} and \textit{state\_updated}. The internal state machine includes a dummy initial\textit{State} 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 \textit{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.

![Diagram](image)

\textbf{Fig. 5.} An example of state machine FB in COMDES-II

When the host actor is activated and then \textit{SMFB\_1} is executed, its internal state machine parses binary input event signals, determines current state and updates two outputs: \textit{state} and \textit{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 $x \cdot y \sim n$, where $x, y \in C$ and $n$ is a natural number.

Syntactically a timed automaton $A$ is a tuple $\langle N, l_0, E, I \rangle$ 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 $\langle l, u \rangle$, 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) $\langle l, u \rangle \xrightarrow{a} \langle l', u' \rangle$ if $\langle l, g, a, r, l' \rangle \in E$, $u \in g$, $u' = [r \rightarrow 0]u$ and $u' \in I(l')$
- (delay transitions) $\langle l, u \rangle \xrightarrow{d} \langle l, u \oplus d \rangle$ 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 $[r \rightarrow 0]u$ 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, $(A_1 \mid \ldots \mid A_n) \backslash Act$. Thus an edge labeled with action $a$ must synchronize with an edge labeled with an action complementary to $a$, and edges with the silent $\tau$ 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 $\tau$ 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 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

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

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:

```c
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 \texttt{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.

![Status transition graph of actor tasks](image)

**Fig. 6.** Status transition graph of actor tasks

Actor input/output drivers will be implemented within two functions in UPPAAL: \texttt{taskInputDrivers(int taskID)} and \texttt{taskOutputDrivers(int taskID)}. These two functions are application-dependent, and when a specific task \textit{i} is released or its deadline expires, they (\texttt{taskInputDrivers(i)} and \texttt{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 $\text{taskOutputDrivers}(i)$ of task $i$ will be performed by preceding the input action $\text{taskInputDrivers}(j)$ 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 ($i \to j$) 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](image)

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 \textit{ACTIVE} task to be running, i.e. the corresponding task \texttt{computationTimer} decrements with the invocation period of the scheduler. When the \texttt{computationTimer} of a specific \textit{ACTIVE} task reduces to zero before its deadline, the task status will be set as \textit{COMPUTED}, meaning that the computation effort has completed and the actor state transition may take place instantaneously, which is followed by the \textit{FINISHED} status. The \textit{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 \texttt{computationTimer} of an \textit{ACTIVE} task is greater than zero (the task has not finished its computation) when its deadline comes, the task will be scheduled into the \textit{ERROR} status.

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

- \texttt{release(int taskID)} takes an integer \texttt{taskID} as its argument and will be invoked when the activation condition of a specific \textit{READY} task \textit{i} becomes true (\texttt{release(i)}). This primitive will set the released task status as \textit{ACTIVE}, reset the \texttt{timeSinceReleased} entry in the corresponding task control block and initialize the \texttt{computationTimer} with the value of \texttt{executionTime} (see Fig. 7).

- \texttt{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 \textit{ACTIVE} task is detected, its \texttt{computationTimer} is decremented by the value of scheduler execution period. If the \texttt{computationTimer} reduces to be zero (i.e. the task completes its computation effort), the task status will be set as \textit{COMPUTED} meaning that the task state transition can now take place, and then the primitive is exited.

- \texttt{finish(int taskID, int mode, bool modeUpdated)} will be invoked immediately when a specific \textit{COMPUTED} task \textit{i} finishes its state transition activity. The primitive records the current operation state as well as the state updated information of task \textit{i} into the corresponding entries of the task control block, through its three arguments, and then set the task status as \textit{FINISHED} (see Fig. 7). In case that the deadline of a specific task \textit{j} is not specified (i.e. \texttt{deadline} = 0), the task output drivers (\texttt{taskOutputDrivers(j)}) will be immediately executed within the \texttt{finish()} primitive to generate the output signals, afterwards the status of task \textit{j} is set as \textit{READY} (see e.g. Fig. 6).

- \texttt{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 \texttt{timeSinceReleased} of each task with its deadline parameter if it is greater than zero (i.e. the task deadline is specified). Once the \texttt{timeSinceReleased} of a given task \textit{i} equals to its deadline, the task \texttt{computationTimer} will be checked to see whether the computation time violates the deadline or not. In case that task is schedulable (i.e. \texttt{computationTimer} is zero) and task has \textit{FINISHED} its control activities, the task output drivers (\texttt{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 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 is zero, its computationTimer will be firstly checked to determine the schedulability of task , which is true if the task computation has completed before the activation instant. In case that task is schedulable, it will be released by invoking the release primitive and its input signals sampled via the execution of taskInputDrivers(); otherwise the task status will be set as ERROR. 2) If the deadline of task 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:

\[
\text{task_run} <= \text{SCHEDULER_PERIOD} !\text{task_state_transition} \&\& s == \text{SCHEDULER_PERIOD} \rightarrow \text{OIRActions()}, s = 0
\]

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(), inputAction() 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 \texttt{COMPUTED} or not, as explained in Section 4.1.3. The condition expression \texttt{task[1].status == COMPUTED} is labeled in all transitions, in which \texttt{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 \texttt{finish()} primitive is invoked as the update action to perform some finishing activities. For instance in the transition from \( s1 \) to \( s2 \), \texttt{finish(1, 2, true)} will register current state index \( 2 \) and state updating information \( \texttt{true} \) into the corresponding entries (say, mode and modeUpdated) of task control block \texttt{task[1]}, and then change the task status to be \texttt{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 \( s2 \), \texttt{finish(1, 2, true)} will register current state index \( 2 \) and state updating information \( \texttt{true} \) into the corresponding entries (say, mode and modeUpdated) of task control block \texttt{task[1]}, and then change the task status to be \texttt{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 \( s2 \), \texttt{finish(1, 2, true)} will register current state index \( 2 \) and state updating information \( \texttt{true} \) into the corresponding entries (say, mode and modeUpdated) of task control block \texttt{task[1]}, and then change the task status to be \texttt{FINISHED}. All the recorded information will be used to help generate correct output signals at the task deadline instant.

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.

![Turntable System Setup](image)

**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

<table>
<thead>
<tr>
<th>Sensors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TTPositionSensor</code></td>
<td>indicates position (rotation) of the TurnTable</td>
</tr>
<tr>
<td><code>CLockedSensor</code></td>
<td>shows if Clamp is locked</td>
</tr>
<tr>
<td><code>CUnlockedSensor</code></td>
<td>shows if Clamp is unlocked</td>
</tr>
<tr>
<td><code>DTopSensor</code></td>
<td></td>
</tr>
<tr>
<td><code>DPositionSensor</code></td>
<td></td>
</tr>
<tr>
<td><code>DPositionDrive</code></td>
<td></td>
</tr>
<tr>
<td><code>DDrillingDrive</code></td>
<td></td>
</tr>
<tr>
<td><code>0 – loading</code></td>
<td></td>
</tr>
<tr>
<td><code>1 – drilling</code></td>
<td></td>
</tr>
<tr>
<td><code>2 – testing</code></td>
<td></td>
</tr>
<tr>
<td><code>3 – unloading</code></td>
<td></td>
</tr>
<tr>
<td>Sensor</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>DTopSensor</td>
<td>true if Drill is in top position</td>
</tr>
<tr>
<td>DPositionSensor</td>
<td>indicates position of the Drill</td>
</tr>
<tr>
<td>TTopSensor</td>
<td>true if Tester is in top position</td>
</tr>
<tr>
<td>TPositionSensor</td>
<td>indicates position of the Tester</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actuators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTPositionDrive</td>
<td>rotates TurnTable in counter-clockwise direction</td>
</tr>
<tr>
<td>CActuator</td>
<td>lockes/unlocks product during drilling</td>
</tr>
<tr>
<td>DPositionDrive</td>
<td>moves Drill up/down</td>
</tr>
<tr>
<td>DDInsingDrive</td>
<td>makes actual hole</td>
</tr>
<tr>
<td>TPositionDrive</td>
<td>moves Tester up/down</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th>Actor</th>
<th>priority</th>
<th>Period</th>
<th>WCET</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>TesterSupervisor</td>
<td>1</td>
<td>100</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>DrillClampSupervisor</td>
<td>2</td>
<td>100</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TurntableSupervisor</td>
<td>3</td>
<td>100</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TurntableController</td>
<td>4</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>DrillController</td>
<td>5</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>TesterController</td>
<td>6</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

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

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

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

Each actor accepts a set of messages, as well as 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 the latter one, as well as 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.
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 TTStateMachine: 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.

![Diagram of DrillClampSupervisor actor](image-url)

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.

![Diagram](image-url)  
**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.

![Diagram of TesterController actor](image)

**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.

![Diagram of Input behavior](image)

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

```c
// 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 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. TesterController 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:

```c
// 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 TTTurnTable 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

<table>
<thead>
<tr>
<th>Actor</th>
<th>Output Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TesterSupervisor</td>
<td>void TesterSupervisorOutput(int taskID)</td>
</tr>
<tr>
<td>DrillClampSupervisor</td>
<td>void DrillClampSupervisorOutput(int taskID)</td>
</tr>
<tr>
<td>TurntableSupervisor</td>
<td>void TurnTableSupervisorOutput(int taskID)</td>
</tr>
<tr>
<td>TurntableController</td>
<td>void TurnTableControllerOutput(int taskID)</td>
</tr>
<tr>
<td>DrillController</td>
<td>void DrillControllerOutput(int taskID)</td>
</tr>
<tr>
<td>TesterController</td>
<td>void TesterControllerOutput(int taskID)</td>
</tr>
</tbody>
</table>

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 TTStateMachine 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) ->
```
read right -> rotating ->

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

ready ->

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

ready -> drilling ->

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

ready -> testing ->

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

ready -> rotating ->

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

ready ->

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

ready -> drilling ->

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

ready -> testing ->

(testOK || testBad) (TTSlot[0].status == LOADED, TTSot[1].status == DRILLED, TTSot[2].status == (TESTOK || TESTBAD), TTSot[3].status == (TESTOK || TESTBAD)) ->
The document contains a description of a Turntable case study, focusing on the TurntableController actor and its functional behavior. The text details the state transitions and actions associated with the TurntableController actor, including its modal function block, TTCModal. It also explains the formulation of system properties and the verification of the system design using temporal logic expressions.
### Table 6. System properties and verification results

<table>
<thead>
<tr>
<th>System Requirement</th>
<th>Temporal Logic Formula</th>
<th>Verification Result</th>
<th>Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>If slot0 is available, a product will be loaded</td>
<td>Input.status == READY &amp;&amp; TTSlot[0].status == EMPTY --&gt; TTSlot[0].status == LOADED</td>
<td>Satisfied</td>
<td>18884 Kbytes</td>
</tr>
<tr>
<td>Every product is drilled</td>
<td>TTSlot[0].status == LOADED --&gt; TTSlot[1].status == DRILLED</td>
<td>Satisfied</td>
<td>23396 Kbytes</td>
</tr>
<tr>
<td>Every product is tested</td>
<td>TTSlot[1].status == DRILLED --&gt; (TTSlot[2].status == TESTOK</td>
<td></td>
<td>TTSlot[2].status == TESTBAD)</td>
</tr>
<tr>
<td>Products are drilled and tested with the same hole depth</td>
<td>A[] TTSlot[2].status != TESTBAD</td>
<td>Satisfied</td>
<td>22324 Kbytes</td>
</tr>
<tr>
<td>Every product leaves the turntable eventually</td>
<td>TurnTable.status == LOADED --&gt; TurnTable.status == UNLOADED</td>
<td>Satisfied</td>
<td>30236 Kbytes</td>
</tr>
<tr>
<td>The turntable does not rotate if any process (loading,</td>
<td>A[] not (TurnTable.status == ROTATING &amp;&amp; (Input.status == LOADING</td>
<td></td>
<td>DrillClamp.status == DRILLING</td>
</tr>
<tr>
<td>drilling, testing, unloading) is in operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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</td>
<td>A[] not ((Input.status == LOADING &amp;&amp; TTSlot[0].status != EMPTY)</td>
<td></td>
<td>(DrillClamp.status == DRILLING &amp;&amp; TTSlot[1].status != EMPTY))</td>
</tr>
<tr>
<td>The system should not perform redundant operations</td>
<td>A[] not ((Input.status == LOADING &amp;&amp; TTSlot[0].status == LOADED)</td>
<td></td>
<td>(DrillClamp.status == DRILLING &amp;&amp; TTSlot[1].status == DRILLED)</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>The system is deadlock free</td>
<td>A[] not deadlock</td>
<td>Satisfied</td>
<td>23240 Kbytes</td>
</tr>
<tr>
<td>Schedulability Analysis</td>
<td>A[] forall(i : int[1..TASKS_NUM]) task[i].status != ERROR</td>
<td>Satisfied</td>
<td>23232 Kbytes</td>
</tr>
</tbody>
</table>
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 Section 5. 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


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;
} TTTurnTable;

TTTurnTable 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{
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--){
        // code here
    }
}
// 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:

\[ \text{task\_run} \leq \text{SCHEDULER\_PERIOD} \]
\[ \text{OIRActions()}, s = 0 \]

Fig. 25. Timed automata for discrete-time scheduler

Fig. 26. Timed automata for \textit{TurntableController} actor functional behavior

Fig. 27. Timed automata for \textit{DrillController} actor functional behavior

Fig. 28. Timed automata for \textit{TesterController} actor functional behavior
Fig. 29. Timed automata for *TurntableSupervisor* actor functional behavior
Fig. 30. Timed automata for DrillClampSupervisor 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