Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 253 (2009) 161–178
www.elsevier.com/locate/entcs

Analyzing a Pattern-Based Model of a Real-Time Turntable System

Davor Slutej [1]

School of Innovation, Design, and Engineering

John Håkansson [2]

Department of Information Technology,
Uppsala University,
Uppsala, Sweden

Jagadish Suryadevara, Cristina Seceleanu, Paul Pettersson [3]

Mälardalen Real-Time Research Centre,
Mälardalen University,
Västerås, Sweden

Abstract

Designers of industrial real-time systems are commonly faced with the problem of complex system modeling and analysis, even if a component-based design paradigm is employed. In this paper, we present a case-study in formal modeling and analysis of a turntable system, for which the components are described in the SaveCCM language. The search for general principles underlying the internal structure of our real-time system has motivated us to propose three modeling patterns of common behaviors of real-time components, which can be instantiated in appropriate design contexts. The benefits of such reusable patterns are shown in the case-study, by allowing us to produce easy-to-read and manageable models for the real-time components of the turntable system. Moreover, we believe that the patterns may pave the way toward a generic pattern-based modeling framework targeting real-time systems in particular.

Keywords: Components, real-time, embedded systems, specification, verification, modeling patterns, case study

1 Introduction

Developing industrial real-time systems is difficult and sets high requirements to system safety and reliability. The short development cycles demand a reliable en-

[1] Email: davor@slutej.com
[2] Email: johnh@it.uu.se
[3] Email: paul.pettersson@mdh.se, jagadish.suryadevara@mdh.se, cristina.seceleanu@mdh.se

doi:10.1016/j.entcs.2009.09.034

gineering method, with predictable costs. The state-of-the-art is dominated by an ad-hoc mixture of methods and tools, and system validation is mostly done by extensive testing at the implementation level. However, testing is done already too late in the design process, and bugs may still exist even in well-tested models. In this context, techniques for managing complexity and ensuring critical system properties during design become a necessity.

A promising design approach is to employ a *formal component-based* development technique. In such an approach, components are introduced as executable software units that can be deployed into a system. One of the key issues of realizing the component-based software paradigm is to ensure that the separately specified components do not conflict with each other when composed, resulting in blocking the system. A potential solution to this issue is *formal modular verification* of component-based software via *model checking*.

In this paper, we present a case-study in formal modeling and analysis of a real-time, component-based turntable system, for which the components are described in the SaveCCM language [8]. For verification, we use an integrated development environment for SaveCCM, connected via a plug-in with Uppaal port, an extension of the model-checker Uppaal, which implements a partial order reduction technique [10] for efficient model-checking. The technique exploits the topology of the network of components and consequently improves the scalability of the verification method.

Our experience with this case-study and other similar examples is that, beside making the model-checking efficient, an as demanding task is to produce manageable and easy-to-grasp design models for components and their composition. This has motivated us to try to extract some common behavioral patterns that occur frequently in the design of real-time systems, and represent them in a finite-state-machine like notation. Such notation lets us apply these patterns at high-levels of software development, as shown in the paper, while simplifying the produced models. We believe that employing patterns in designing component-based systems might also help in documenting the associated software, through pattern-based reverse engineering. However, this is out of the scope of this paper.

General purpose program design patterns are well-known in the object-oriented design community for a while now [9]. Nevertheless, in the design of component-based real-time systems, some different aspects might need to be represented in the modeling patterns; for instance, the semantics of our SaveCCM components is a read-execute-write semantics, hence a run-to-completion pattern can prove beneficial in the design. Similarly, the reusable modeling of the sequence of visited states during the execution of a component, or reducing the time-wise non-determinism of the real-time component behavior, by providing systematic means to associate a *deadline* with the behavior, through a pattern, might also help the designer in the modeling phase. In this paper, we introduce the just mentioned abstractions of common real-time component behaviors, as the run-to-completion, history, and execution-time patterns, respectively. Next, we apply them in modeling the component-based turntable production cell.

The remainder of the paper is organized as follows. In section 2, we briefly recall

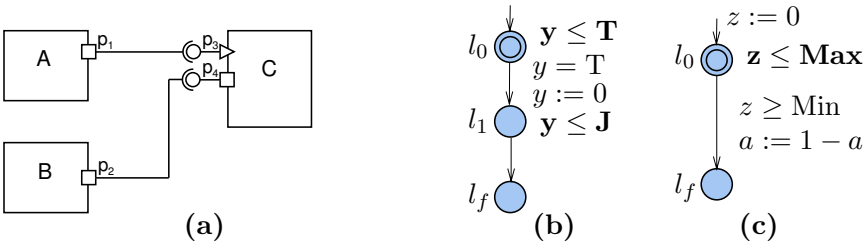

Figure 1. An example of **(a)** a composition where components A, B and C are composed by connecting port p_1 to p_3, and p_2 to p_4, and timed behaviors: **(b)** a clock with period T and jitter J, **(c)** a computation updating data variable a after between Min and Max time units.

the basics of the SaveCCM language used for modeling the components in our case-study. The three modeling patterns are introduced and described as finite state machines in section 3, after which we present the real-time turntable production cell example, including the formal models of the constituent components, in section 4. The system's formal requirements and verification results are displayed and discussed in sub-section 4.3. We compare our approach to related ones, in section 5. Finally, section 6 concludes the paper and outlines possible directions for future work.

2 SaveCCM

In this section we briefly present the Save component modeling language [8], which will be used in the case study of this paper. The language is part of a larger framework, called SaveCCM, for component-based design of real-time and embedded system [1]. The SaveCCM language consists of a graphical syntax and an associated formal semantics. Due to space limitation, the presentation in this section is restricted to a short informal overview of SaveCCM. For a complete description of the language we refer the reader to [8].

In SaveCCM, systems are built from interconnected components with well-defined interfaces consisting of input and output ports. The communication style is based on the pipes-and-filters paradigm, but with an explicit separation of data transfer and control flow. The former is captured by connections between *data ports* where data of a given type can be written and read, and the latter by *trigger ports* that control the activation of components. Figure 1(a) shows an example of the graphical SaveCCM notation. Triangles and boxes denote trigger ports and data ports, respectively.

A component remains passive until all input trigger ports have been activated, at which point it first reads all its input data ports and then performs the associated computations over this input and an internal state. After this, the component writes to its output data ports, activates the output trigger ports, and returns to the passive state again. This strict "read-execute-write" semantics ensures that once a component is triggered, the execution is functionally independent of any concurrent activity.

Components are composed into more complex structures by connecting output ports to input ports of other components. In addition to this "horizontal"

composition, components can be composed hierarchically by placing a collection of interconnected components inside an enclosing component. From the outside, such a composite component is indistinguishable from other component where the behavior is given by a single model or piece of code.

To support analysis of SaveCCM models, it is required that each component is associated with a behavioral model consisting of a timed automaton [3] with a distinct exit location (see Figure 1(b-c)), and a mapping between component data ports and the internal automata variables. When a component is triggered, the port values are copied to the internal variables of the timed behavior which then proceeds as specified in the timed automaton. Whenever it reaches the exit location, variable values are copied to the output ports according to the given mapping, and the output trigger port is activated.

The timed automata modeling language used in SaveCCM is based on the language used in the UPPAAL tool [16]. It extends the timed automata language originally introduced by Alur and Dill [3] with a number of features that will be used in the case study, including: global and local bounded integer variables and arithmetic operations over such variables, arrays, and a small C-like programming language that can be used to define functions and predicates. For a detailed description of the timed automata language, we refer the reader to [5].

3 Component Modeling Patterns

A modeling pattern is a way of designing a model with a clearly stated intent and structure. In this section, we propose three modeling patterns for common behaviors of real-time components, in order to ultimately provide the designer with useful abstraction mechanisms for the high-level modeling and analysis of CB real-time systems. We chose to define the patterns by a finite-state-machine like (FSM) notation, which we call *Pattern-FSM* (or PFSM) in this paper. The patterns can be instantiated, separately or in combination, in specific formal frameworks, to increase the readability of the models and their suitability for verification. To justify our claim, in section 4, we apply the proposed patterns, as combinations, to the CB modeling of an industrial real-time turntable system (see for instance Figure 10). The analysis framework is the Timed Automata (TA) language of UPPAAL [5,16].

Generic PFSM Definition and Graphical Notation. Let V be a set of data variables, G be a set of boolean conditions (*guards*) over V, and A a set of actions that update the variables. Then PFSM is a tuple $\langle S, \mathsf{start}, \mathsf{exit}, E, Att\rangle$, where S is a set of states, start is the *entry* state, exit is the *exit* state, $E \subseteq S \times G \times A \times S$ is the set of transitions between states, and Att is a set of timing attributes, e.g. execution time, deadline, etc.

The execution of a PFSM starts in the special control state start. At a given state, an outgoing transition may be executed only if its associated guard evaluates to true; in this case we say that the transition is *enabled*. In case more than one outgoing transitions are enabled, one can be executed non-deterministically. A filled circle denotes the start control state and a semi-filled circle denotes the exit control

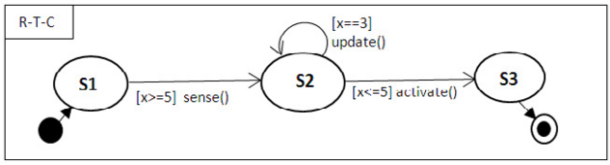

Figure 2. PFSM specification of a component behavior

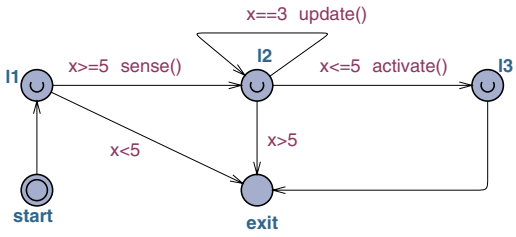

Figure 3. An equivalent timed automata model with run-to-completion pattern

state (see Figure 2). Different attributes of a PFSM, e.g. execution time, deadline etc. can be added to the graphical representation of a PFSM model (e.g. Figure 7).

3.1 Run-to-Completion Pattern

In the run-to-completion (RTC) execution model, the component is executing in indivisible steps, without interruption from any concurrent activity. The key advantage of the RTC semantics is simplicity and guaranteed absence of deadlocks. Another advantage is that it might prune away unnecessary interleavings, thus speeding up formal verification and bringing the model closer to implementation. The pattern is commonly used in high-level behavioral modeling languages like Statecharts and its variants [12,17]. In Statecharts, the events are handled in an RTC manner, along possibly compound transitions (i.e., paths of adjacent arrows).

Pattern description. In this pattern, we assume that the component execution proceeds with changing states by firing enabled transitions until it reaches a state for which no outgoing transitions are enabled. At such a point, the execution terminates.

To implement the pattern, one needs to translate the corresponding PFSM into a timed automaton (TA). Run-to-completion can be implemented by introducing new edges in the automaton, which describe termination of component execution. Let L be the set of locations $l_i,\ i \in \{1,..,n\}$ in the corresponding TA. For each location $l_i \in L$, we assume that $g_j,\ j \in \{1,..,m\}$ are the guards of the respective outgoing edges. The exit edge from l_i connects l_i with the exit location. The guard of the l_i exit edge is $\neg(\bigvee_j g_j)$.

Example. Figure 2 represents a PFSM specification of a simple component behavior obeying our run-to-completion pattern. Figure 3 describes the equivalent behavior as a timed automaton, which serves as the pattern implementation. The states S1, S2, and S3 of the PFSM are mapped onto locations l1, l2, and l3, respectively, in the equivalent TA.

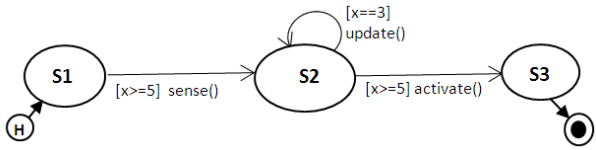

Figure 4. PFSM specification of a component behavior with history

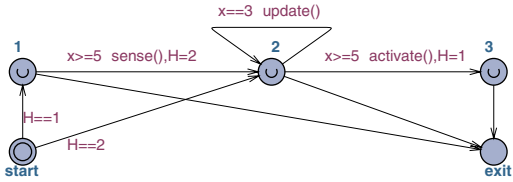

Figure 5. A timed automata behavior with history pattern

3.2 History Pattern

Execution history is a core feature of behavior modeling techniques [2,12]. The history mechanism of a behavior remembers which state was last visited during execution, before exiting. This state can then be re-entered next time the execution re-starts. In the hierarchical state-machine modeling of Statecharts [12], an inner state may be exited and re-entered directly, by using the history mechanism. A similar approach is adopted in CHARON, a formal modeling framework for hybrid systems [2].

Pattern description. The pattern provides a mechanism to remember the execution history in the behavioral models of components. Assuming the execution as a sequence of states, the pattern has means of remembering the last state, or a particular state for that matter, reached during execution. Hence, the next time, the execution can resume from the state stored through the history mechanism. Similar to Statecharts, in a PFSM representation, the history mechanism is denoted as an H within a circle, and acts as the start state.

The pattern is implemented as a TA, by using an integer variable H, which is updated along each edge connecting any states different from the start, and exit states, with the corresponding location identifier. Special edges connect the start state to each of the states of interest, while appropriately testing the variable H. In addition, exit edges connect each state of interest to the exit control state. Variable H can be re-initialized appropriately when entering a specified final location.

Example. Figure 4 represents a component behavior with history pattern. The history is denoted by the encircled H symbol, in the start state. In Figure 5, we give the equivalent behavioral model as a TA, which implements the history pattern. The states in Figure 4 are mapped onto locations 1, 2, 3 in the TA. Variable H is initialized to an initial location, i.e., $H = 1$. The edges that connect the start location to locations 1, and 2 are due to the pattern, and are guarded by conditions H==1, and H==2, respectively. Also, the history variable H is updated with the location identifier along each edge entering that respective location (edges that leave and

enter the same location may be skipped, e.g., location 2 in Figure 5). Finally, H is re-initialized at location 3 of Figure 5.

3.3 Execution-Time Pattern

For embedded and real-time systems, it is often interesting to specify and analyze the best or worst execution time of components. The variation in execution time also gives rise to, e.g., non-deterministic timing, jitter, and varying end-to-end timing, which represent phenomena that are important to analyze (and master) at design time. In the following, we introduce a pattern for specifying the best and worst execution times of components.

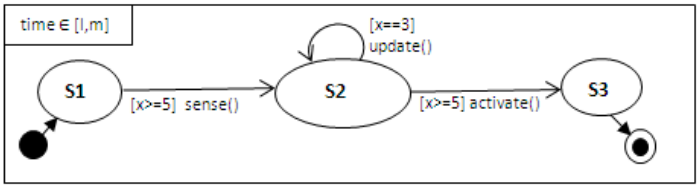

Figure 6. Annotation of time attributes on PFSM models for execution-time pattern

Pattern description. In this pattern, we assume that the total accumulated time of executing a component is within an interval where the lower and upper bounds are the shortest and longest possible execution times, respectively. Hence, the component will produce output (data and trigger) at some time instance, in the interval.

We also assume that the component is annotated with an interval specifying the lower and upper bound on the execution time. To implement the pattern, we use a dedicated clock, say exec, which is used to measure the time since the component was triggered. The clock is reset on the edge outgoing from location start. We further introduce a location, say delay, and an edge from location delay to the exit location. Location delay is annotated with an invariant over exec, corresponding to the upper bound of the execution interval, whereas the exit edge is decorated with a guard corresponding to the lower execution bound.

Example. Figure 6 represents a PFSM specified using the execution time pattern. Its execution time is in the (closed) interval [l, m]. Figure 7 shows a timed automaton implementing the pattern. Note that when the exit location is reached, the value of clock delay is in the interval [l, m].

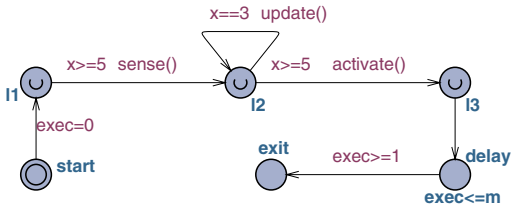

Figure 7. A timed automata behavior with execution-time pattern

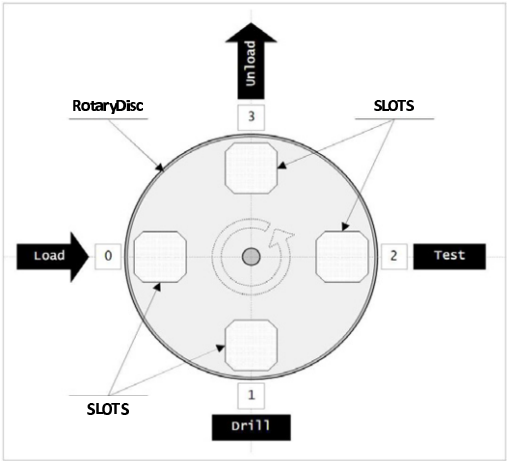

Figure 8. Schematic diagram of a Turntable system

4 Turntable Production Cell

In industry automation, a production cell is a part of an overall production system — a factory. In this section, we present a formal model of a turntable production cell, previously described in [6,19]. The case study is designed using the component framework described in Section 2 and the patterns introduced in Section 3. By employing the patterns, we get simple and understandable component models for our case-study, as shown in the following subsections.

The turntable cell is illustrated in Figure 8. It consists mainly of a rotary disc with four product slots. A product is *loaded* into a slot at position 0, and is then rotated to position 1 where it is *drilled*. It is then rotated into position 2 where it is *tested*, and finally to position 3 where it is *unloaded* (or possibly left to be redrilled in the next cycle). The positions are aligned with various tools for loading, drilling, testing, and unloading.

Drilling and testing are the most critical tool positions, as the overall purpose of the production cell is the verified drilling of products that flow through the cell. All slots of the rotary disc may be occupied at the same time, and products are processed in parallel. When a cycle completes, meaning that all positions complete their functionality, the rotary disc rotates 90 degrees thus positioning the products for the next phase of processing. As the rotation is initiated by signals from tools that are not time deterministic, there is no fixed period between rotation of the slots.

4.1 System Design

Following the informal description of the system, we can identify the system as consisting of five main software components: Turntable, Loader, Driller, Tester, and Unloader, corresponding to the functionalities of the cell. The components interact with several sensors and actuators, such as position sensors, clamping, and drilling devices, which do not require explicit modeling. Further, as we focus on modeling

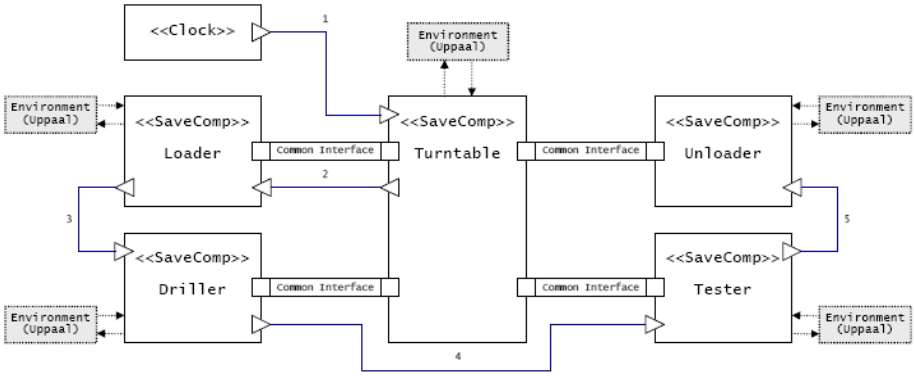

Figure 9. Software architecture design layout of Turntable system

Table 1
Common interface for components Loader, Driller, Tester, and Unloader

Port	Data type	Description
status	int	An input representing the current known status of the product in the tool position (0 indicates an empty slot).
result	int	An output that holds the status of the product after processing.
start	bool	An input that initiates tool processing.
finished	bool	An output that signals when the tool controlled by the component has completed its processing.

and analysis of the functional and timing behavior of the system, we make assumptions regarding error situations, e.g., no fault situations like broken tools, etc. This simplifies the system model without loss of generality.

We now describe in detail the software components in terms of their interfaces and behaviors. Figure 9 shows the software architecture of the turntable system. An interface of a component defines the access point to its behavior, in our case in terms of data ports and trigger ports. The Turntable component acts as a central controller in the system, and all other components are independent of each other and have a similar interface with Turntable. The common interface approach supports reuse, as well as the flexibility to extend or modify the system architecture. We define a common interface for each component, except Turntable, as shown in Table 1.

Data flow is defined by connections between data ports, within the common interfaces and with external sensors and actuators. The control flow is modeled separately from the data flow, by connections between triggering ports. As illustrated by Figure 9, the flow starts from the Clock component and ends at the Unloader.

The component behaviors are modeled as finite state machines under the assumption of the modeling patterns defined in previous section. The history and the run-to-completion patterns are combined to achieve the modeled finite state machine behavior of the components, eventhough the components will be executed in a time-triggered fashion. The execution time pattern is applied to model the time required to execute each component. As such, the models present intuitive conceptual modeling retaining the analysis capability of the underlying formalism, i.e., timed automata. The modeled behaviors execute under the semantics of SaveCCM

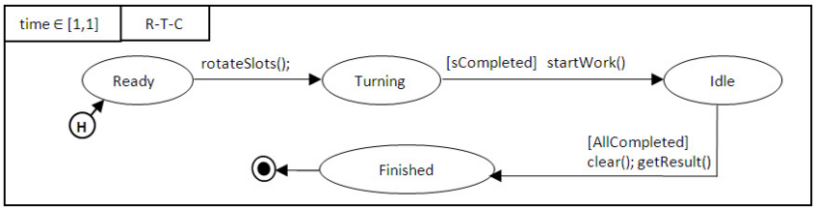

Figure 10. Behavioural model Turntable component.

```
rotateSlots() is
    temp : int := status_0
    aRotate := true
    status_0 := status_3 ; status_3 := status_2
    status_2 := status_1 ; status_1 := temp
end

startWork() is
    for positions i do start_i := true
end

getResult() is
    for positions i do status_i := result_i
end

clear() is
    for positions i do start_i := false
end

allCompleted iff ∀i : finished_i
```

Figure 11. Functions and predicates used by Turntable.

component model and the semantics of the patterns. In the following, we describe each of the component behaviours along with their associated functions and predicates, defined in terms of variables associated with the data and trigger ports of the corresponding component.

4.1.1 The Turntable *Component*

The interface of the turntable controller consists of two trigger ports, a sensor input, an actuator output, and four instances of the *common interface*. A clock component generates trigger signals to periodically activate Turntable, which in turn activates the Loader component. The actuator output aRotate is connected to a motor turning the rotary disc, and the sensor inputsRotated senses when the rotation is completed. The behavior of the Turntable component coordinates the rotation of the disc with the execution of other components.

Initially it rotates the disc, and sets ports of other components appropriately. It then waits for the other components to signal that their processing has stopped, before restarting the main loop by turning the disc again [4] . Starting from an empty system, it will take at least four rotations for all components to work in parallel. The first rotation only starts processing of the Loader, which then loads the first product onto the table. In addition to controlling the rotation of the disc, the component also maintains status information for each position. The status information is shifted one step each time the table rotates. The detailed behavior is modeled in Figure 10, in terms of associated functions and predicates (listed in Figure 11). The internal variables $\mathsf{status}_i, \mathsf{start}_i, \mathsf{finished}_i, \mathsf{result}_i$ represent the data values of the corresponding *common* interface ports of position i.

[4] Hence, even though Turntable is triggered periodically, the period of the rotation of the disc depends on the processing time in the four slots.

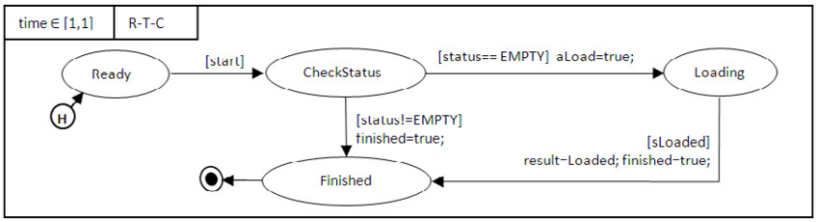

Figure 12. Behavioral model of Loader component.

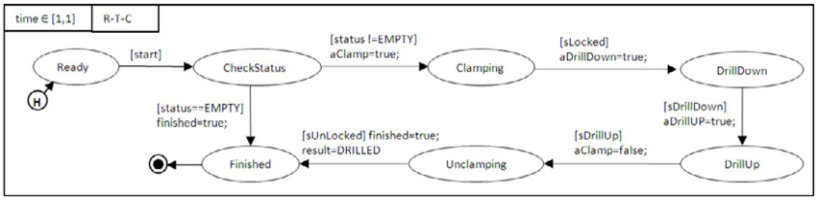

Figure 13. State machine model of the Driller component.

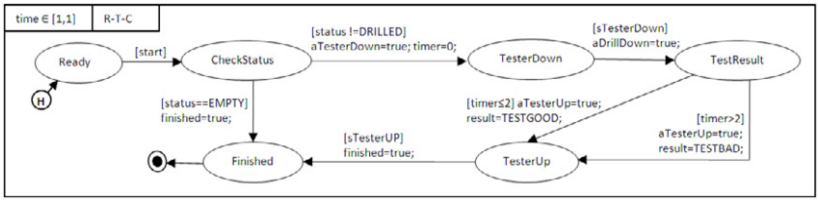

Figure 14. State machine model of the Tester component.

4.1.2 The Loader Component

As mentioned, Loader shares a common interface with, and receives a trigger, from Turntable. It also has a trigger output to the Driller, sensor input sLoaded, and actuator output aLoad. The behavioral model is shown in Figure 12. When triggered the component checks the status of the slot at position 0. If a previous product is present, forwarded by the Unloader for reprocessing, the product is left in the slot for repeated drilling. Otherwise a new product is loaded into the slot, to be drilled in the next cycle.

4.1.3 The Driller Component

Figure 13 shows a model of the Driller component behavior, which interacts with actuators and sensors for clamping and drilling the product. When triggered the component checks the status of the slot at position 1. If empty, the driller does nothing, otherwise the product in the slot is fixated (clamped), the drill starts spinning and is lowered. When the drilling is completed, the drill is lifted and stopped, and the status of the slot is updated accordingly.

4.1.4 The Tester Component

The behavioral model of Tester is shown in Figure 14. Its input trigger is received from Driller, and its output trigger output is sent to Unloader. Similar to the driller, it interacts with actuators and sensors to move a tool into the product. The tool of

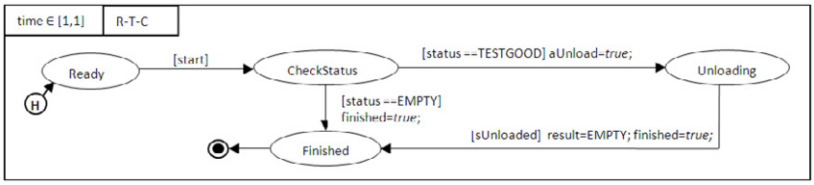

Figure 15. State machine model of the Unloader component.

the tester is a sensor sTesterDown, that measures the hole within 2 time units since the beginning of the test process. When triggered the component checks the status of the slot at position 2. If empty, it does nothing, otherwise it measures the hole drilled in the product, and updates the status according to its verdict.

4.1.5 The Unloader *Component*

Figure 15 shows a model of the Unloader behavior. The status of the drilled product at position 3 indicates the verdict determined by the previous tester component. If the product was faultily drilled, it is not unloaded, otherwise, the component activates an actuator to unload the product. If the slot is empty, as in initial rotations, the Unloader does nothing.

4.2 Modeling a Closed System

For verification purposes we define a closed system, that is, a system with no inputs or outputs. A closed model of the turntable is created by composing the turntable controller software with an Uppaal timed automata model of the environment that is affected by actuators, and affects sensors. The software architecture of the turntable controller is presented in Figure 16 (as it appears in the SaveCCM syntax in the Save-IDE). The behavior of each component, as modeled in the previous section, is translated into TA, following the modeling patterns presented in section 3.

The environment of the turntable control software is modeled with appropriate abstractions of the complex real world aspects, in such a way that the behavior (and timing) of the real physical environment is included in the model. Further, as mentioned earlier, the model is done under the assumption of normal behavior, meaning no exception handling or error conditions such as faulty sensors or actuators may occur. The environment of the turntable system is modeled as timed automata (TA) in the Uppaal tool. The environment essentially consists of the actuators and sensors associated with the system and its components. Due to space limitation, we leave out some of the environment automata, and we refer the reader to our recent work [19] for a more detailed environment model.

The communication interface between the system and its environment is facilitated by shared variables. These variables correspond to the communication ports between the modeled system software and its sensors and actuators, as well as test automata that drive the verification process. The interface, and its initialization, is given in Table 2. To simplify the modeling process, and reduce the state space

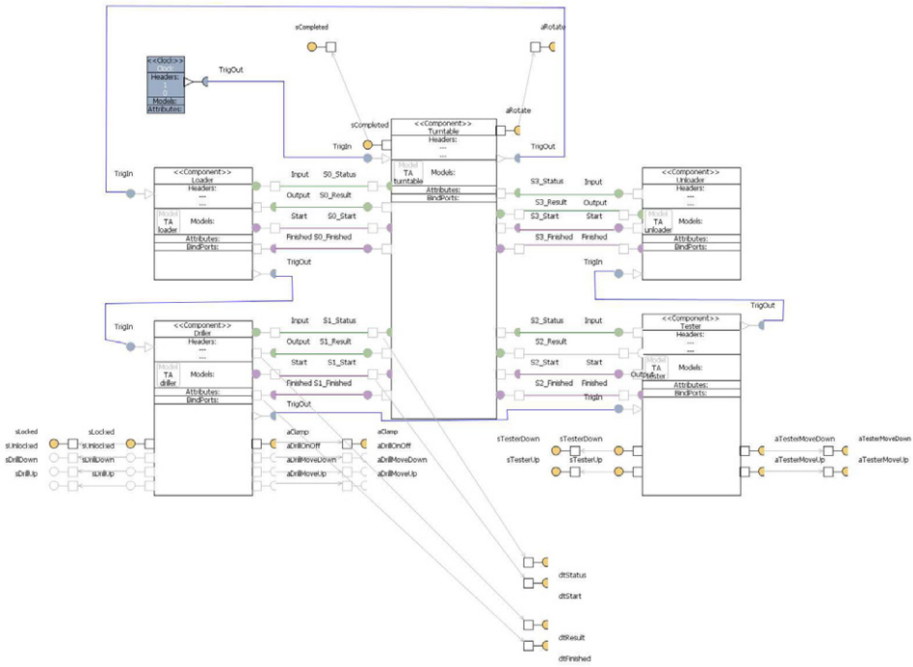

Figure 16. Control structure and system architecture of the turntable system as modeled in Save-IDE.

of the model, all aspects of a system are not modeled explicitly. Instead, models focus on critical aspects of the system. The environment model used for the formal verification of the turntable consists of the behaviors Disc, Clamp, Drill, and TestTool.

The drilling tool is modeled in terms of its two controllable parts: Clamp and Driller. The behavior of these environment models are presented in Figures 17 and 18, respectively. The function of the clamp is to lock the product in place so that the drilling can be carried out. The timed automaton is initially in the location UnLocked, and transitions to the location Locking when the edge guard aClamp goes high (value becomes 1). It can remain in the location Locking as long as the associated invariant claCLK $\leqslant$ ClampTime holds. The same happens when the clamp is in location UnLocking. This models the continuous behavior of the Clamp.

The function of Driller is to make holes in the product. The timed automaton

Table 2
Interface of the environment components

TA	*Variables*	*Data type*	*Initially*
Disc	aRotate, sCompleted	bool	false
Clamp	aClamp,	bool	false
	sLocked, sUnlocked		
Drill	aDrillDown, aDrillUp	bool	false
	sDrillDown, sDrillUp		
TestTool	aTesterDown, aTesterUp	bool	false
	sTesterDown, sTesterUp		

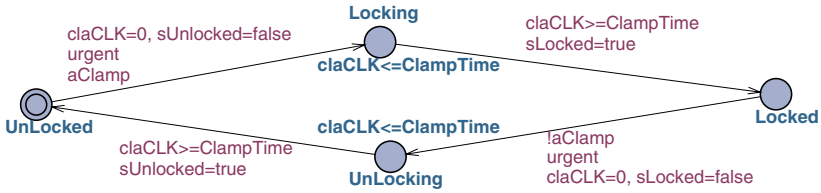

Figure 17. Behavior of the Clamp environment model.

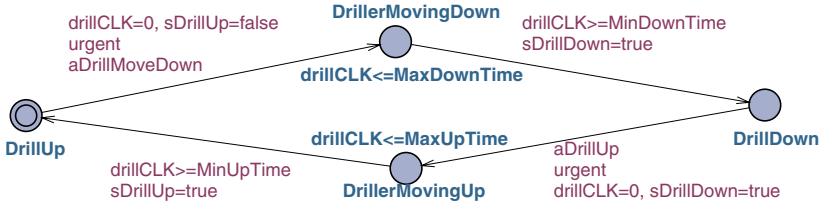

Figure 18. Behavior of Drill of the environment model.

(Figure 18) is initially in the DrillUp location, and transitions to DrillerMovingDown when the guard aDrillMoveDown goes high. It can remain in this location as long as the associated invariant $drillCLK \leqslant$ MaxDownTime holds to model the maximum time the drilling can take place. The same happens when the drill is in location DrillerMovingUp. The driller moves out from the continuous behavior of drilling down or drilling up after MinDownTime or MinUpTime, respectively.

The TestTool works similarly to the drill, moving down by command from an actuator until a sensor is activated, and then moving up again by command from a different actuator until the corresponding sensor is activated. Also Disc is modeled with two states, wait and turning. The transition from wait to turning is initiated by the actuator aRotate, clears the sensor value sCompleted, and resets a clock ensuring the transition back to wait within TURN_TIME time units, when the sensor value sCompleted is also set.

4.3 Requirements and Verification

In this section, we present the verification aspects of the turntable system. The work has been performed in the SAVE-IDE, an integrated development environment for SaveCCM. For modeling, the Save-IDE provides graphical editors for architectural and behavioral modeling. For system (symbolic) simulation and verification by model-checking, the tool UPPAAL PORT [11,10], an extension of UPPAAL [16], is integrated through a plug-in. The representation of the system architecture and component behaviors is represented in the SaveCCM XML file format [8], and the environment is stored in an UPPAAL XML file. UPPAAL PORT connects system inputs and output to global variables in the environment model.

A set of properties concerning the safety and liveness of the Turntable control system have been verified. In UPPAAL, liveness properties can be specified as *leads to* properties in the form $P \rightsquigarrow P'$, meaning that if a system has reached a state with P satisfied, it will eventually reach a state where P' is satisfied. We discuss a few representative properties below. The first property specified is:

$$A\Box\, \neg \textbf{deadlock} \tag{1}$$

Property 1 is a safety property, specifying the absence of deadlock situations. A deadlock occurs when the system can not progress further. In a real-time system, this is often caused by two tasks mutually excluding each other from acquiring a resource (e.g. semaphore). It can also be caused by a fault in the environment model. The property is verified as listed above. The A is a universal quantifier, and refers to the property to be verified on all execution paths of the statespace. The box $\Box$ is a universal quantifier over all states in a path. The states are defined by values of all variables as well as locations of automata. The keyword *deadlock* represents a state in the execution where there is no outgoing (delay or action) transition. The turntable system is verified to be *deadlock free*.

The absence of a deadlock does not mean that the system is guaranteed to make progress. The control system could be continuing with the component trigger without the components progressing through their respective finite state machines. The following set of properties verify that the turntable system is progressing. It checks that the central component Turntable continuously moves between Idle and Turning states. This is specified using *leads to* properties. The diamond $\Diamond$ is an existential quantifier over states in the path, meaning that the property is eventually satisfied by a state in the path (all paths in this case).

$$A\Diamond\, \mathsf{Turntable.Turning} \qquad \begin{array}{l} \mathsf{Turntable.Turning} \leadsto \mathsf{Turntable.Idle} \\ \mathsf{Turntable.Idle} \leadsto \mathsf{Turntable.Turning} \end{array} \tag{2}$$

The properties 2 establishes that the component Turntable always progresses. This is possible only when the individual components too are progressing following the design strategy. The progress of individual components can be verified as below.

$$\mathsf{Loader.Ready} \leadsto \mathsf{Loader.Finished} \tag{3}$$

The above leads-to property 3 verifies that Loader always progresses. We can verify a similar property for all other components. Further, we verify an important safety property stating that when the Turntable component is executing, no other components are executing:

$$A\Box(\mathsf{Turntable.Turning} \Rightarrow (\mathsf{Loader.Ready} \wedge \mathsf{Tester.Ready} \wedge \mathsf{Unloader.Ready} \wedge \mathsf{Driller.Ready})) \tag{4}$$

Property 4 models the fact that while the Turntable is turning the other components are just waiting in their Ready location, according to the design strategy.

Property 5 establishes a state correspondence between an environment component and the corresponding SaveCCM component. The property ensures that whenever the Turntable is not turning, the Disc component is not turning either:

$$A\Box(\neg\mathsf{Turntable.Turning} \Rightarrow \neg\mathsf{Disc.Turning}) \tag{5}$$

The next property (6) specifies that the control model never sends two conflicting signals to its environment. Here, it checks that the system does not activate both

actuators associated with the Driller component, simultaneously, as they move the Drill in opposite directions:

$$A\Box\ \neg(\text{Driller.aDrillDown} \wedge \text{Driller.aDrillUP}) \tag{6}$$

5 Related Work

There are a number of component based development (CBD) frameworks for embedded systems described in the literature. The BIP framework and the toolkit IF [4] are intended for predictable embedded systems development by supporting *correctness-by-construction* and compositional verification. While BIP offers bottom-up design of systems, our approach supports CBD in a bit more pragmatical traditional top-down design, with support of modeling in Save-IDE [18] and formal verification using the Uppaal port toolkit [11,16].

The Charon toolkit [2] supports modular specification of embedded systems, based on the notions of *agents* and *modes*, for architectural and behavioral specifications, respectively. Our behavioral specification language of components shares some features of the modes in Charon, but without hierarchy, and in our approach the execution history of a component is provided by using a simple design pattern.

The Statemate toolkit [14] is an early working environment for the development of complex reactive systems. Modularity of the system development is provided in terms of different *views*, such as structure, functionality, and behavior. Our approach for behavior specification of components (modules in Statemate) is similar to the Statecharts [13], the behavioral language of Statemate. Though not hierarchical, our FSM notation for component behaviors (see Section 3), combined with the patterns proposed in this paper, is similar to the Statechart features run-to-completion and execution history.

The case study of Turntable production system, presented in this paper, has previously been analyzed using different methods and tools. In [7], a turntable model is specified in χ [20], a simulation language for industrial systems, and translated into Promela, the input language of the Spin model-checker to verify several properties of the model. In [6], a χ model of the turntable system was translated into the specification languages of three model-checkers: CADP, Spin, and Uppaal comparing both the ease of conversion, the expressiveness of each of the specification languages, and the abilities and performances of the respective model-checkers. In [15], the turntable production system was implemented in the COMDES-II component-based software framework. The authors developed a semantic transformation of the COMDESS-II model into an Uppaal timed automata model, allowing for formal verification of a set of properties similar to those in [6].

6 Conclusion

In this paper, we have presented how the SaveCCM component-based approach for development of embedded systems has been applied in a case study, to model and

verify an industrial turntable production system. We have presented a component-based system architecture model, as well as the detailed behavioral models of the system components. To produce a manageable and easy-to-grasp design model of the turntable, we have used three simple, but useful, design patterns. The finite behaviors of components are specified in a finite state machine notation, using two design patterns for encoding run-to-completion semantics, and history states. Timing is introduced using a third design pattern for specifying the execution time and order of components. We also describe how the design specifications are syntactically transformed into the modeling framework used in SaveCCM, for further analysis using UPPAAL PORT.

Throughout the case study, we have been using Save-IDE and its connection to UPPAAL PORT, for editing models, as well as for performing (symbolic) simulation, and verification by model-checking. As a modeling result, we believe that we have produced a very intuitive component-based model of the turntable system. As verification results, we have shown that the system model satisfies all the requirements specified for the system, formalized as safety and liveness properties in TCTL.

As future work, we intend to develop an enriched behavioral modeling language and formal analysis support for the successor of SaveCCM, called ProCom. The language will be based on the design patterns described in this paper, and possibly on other newly developed, more involved patterns that might prove useful in simplifying both the formal models and their verification.

References

[1] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[2] R. Alur, D. Thao, J. Esposito, H. Yerang, F. Ivancic, V. Kumar, P. Mishra, G.J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, January 2003.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM*, pages 3–12, 2006.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[6] E. Bortnik, N. Trčka, A.J. Wijs, S.P. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a χ model of a turntable system using Spin, CADP and Uppaal. *Journal of Logic and Algebraic Programming*, 65(2):51–104, 2005.

[7] V. Bos and J.J.T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17:185–198, 2001.

[8] J. Carlson, J. Håkansson, and P. Pettersson. SaveCCM: An analysable component model for real-time systems. In *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing. AddisonWesley Publishing Company, Reading, Massachusetts, 1995.

[10] J. Håkansson and P. Pettersson. Partial order reduction for verification of real-time components. In *Proc. of 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science. Springer–Verlag, 2007.

[11] John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-based design and analysis of embedded systems with uppaal port. In *6th International Symposium on Automated Technology for Verification and Analysis*, pages 252–257. Springer–Verlag, October 2008.

[12] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[13] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[14] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-trauring, and D Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16, 1991.

[15] Xu Ke, P. Pettersson, K. Sierszecki, and C. Angelov. Verification of comdes-ii systems using uppaal with model transformation. *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 153–160, Aug. 2008.

[16] K.G. Larsen, Paul Pettersson, and Yi. Wang. Uppaal in a nutshell. *Int. J. on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[17] Bran Selic. An efficient object-oriented variation of the statecharts formalism for distributed real-time systems. In *Proceedings of the 11th IFIP International Conference on Computer Hardware Description Languages and their Applications - CHDL '93*, volume A-32 of *IFIP Transactions*, pages 335–344. North-Holland, 1993.

[18] Sverine Sentilles, John Håkansson, Paul Pettersson, and Ivica Crnkovic. Save-ide an integrated development environment for building predictable component-based embedded systems. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[19] Davor Slutej. Component-based modeling and analysis of embedded systems. Master's thesis, Department of Computer Science and Engineering, Mälardalen University, September 2008.

[20] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129 – 210, 2006.