Pattern-driven Support for Designing Component-based Architectural Models

Jagadish Suryadevara, Cristina Seceleanu and Paul Pettersson
Mälardalen Real-Time Research Centre (MRTC)
Mälardalen University
Västerås, Sweden
e-mail: {jagadish.suryadevara,cristina.seceleanu,paul.pettersson}@mdh.se

***Abstract*—The development of embedded systems often requires the use of various models such as requirements specification, architectural (component-based), and deployment models, across different phases. However, there exists little design support for obtaining suitable component-based designs that satisfy specified requirements and timing constraints. In order to provide guided support for the design process of embedded systems, we introduce several component templates, referred as patterns, which we also formally verify against relevant properties. To illustrate the usefulness of the approach, we have applied the proposed patterns to obtain a component-based design of a temperature control system.**

***Keywords*-clock constraints; UML/Marte; components; embedded system;**

I. Introduction

To achieve behavioral predictability of an embedded system, one might need to use, during system development phases, extensive modeling and analysis, prior to the actual system implementation. In general, these phases rely on various models, such as, requirements specification, design, and deployment models, before the implementation stage. As different models are based on different semantics, and focus on different aspects of the system, a guided design process from one phase to another is needed. Further, the design process should ensure that the system aspects, such as, functional requirements, timing constraints etc, are met by all system models, from any design phase to the subsequent ones.

Designing an embedded system in a *component-based* style has become an attractive approach. With benefits ranging from simplification and parallel working to pluggable maintenance and reuse, the advantages are significant. In this context, an embedded system consists of identifiable, relatively independent and generally replaceable units of composition, called *components*, which encapsulate complex functionality. A *component model* defines syntax and semantics of a component language through its architectural level elements, such as, components, ports, connections, and connectors to build system parts and their compositions. There exist several component models such as JavaBeans [1], Koala [2], SOFA [3], [4], and ProCom [5], [6], to name a few.

In this paper, we present a pattern-based design process to develop component based designs of an embedded system that preserve the specified functional requirements and related timing constraints. We propose a set of component templates, called component patterns in this paper, to transform a specification model together with functional and timing constraints, into a corresponding component design. The proposed patterns are described in ProCom [5], [6], a language for component-based design of embedded systems. Further, the patterns are formally verified to satisfy relevant timing properties. This is done by translating the pattern specifications in ProCom, into corresponding timed automata models, and model-check the resulting models using UPPAAL [7].

To specify the functional requirements, and related timing constraints of a system, we use an extended form of UML statemachines [8] together with UML/Marte timing profile [9], as the specification models (also referred as *modemachines* in this paper). The timing constraints are specified using the standard constructs of Marte CCSL (Clock Constraints Specification Language).

Finally, to illustrate the applicability of our approach, we apply the patterns in the development of a ProCom based component design for a Temperature Control System (TCS).

The rest of the paper is organized as follows. In Section II, we present an overview of the ProCom component language. As a running example, we describe a Temperature Control System (TCS), in Section III. In Section IV, we present the specification language for modeling the functionality, and timing constraints of an embedded system. In Section V, we propose a set of component patterns for modeling "timers", "clocks", "controllers", as well as the periodic and sequential behaviors. Also, the formal verification of the patterns with respect to relevant properties is described in Section VI. In Section VII, a complete ProCom design of TCS is presented. Related work is discussed in Section VIII. Finally, in Section IX, we conclude the paper with future directions of work.

II. ProCom Component Model: An overview

In this section, we present an overview of ProCom[1] [5], [6], a recently developed component model for designing *real-time* embedded systems in the vehicular and telecom domains. To address the different concerns that exist on

[1]Developed at MRTC, Mälardalen University, Sweden.

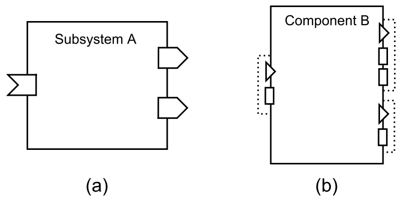


Figure 1. a) A ProSys subsystem and b) A ProSave component.

different levels of granularity or various phases of system development, ProCom is organized into two distinct layers: ProSys and ProSave. The layers also differ in terms of architectural style and communication paradigm. In ProSys, a system is modeled as a collection of communicating *subsystems* that execute concurrently, and communicate by asynchronous messages sent and received at typed output and input *message ports*.

On the other hand, the lower layer, i.e., ProSave consists of passive units, and is based on a pipes-and-filters architectural style with an explicit separation between data and control flow. The former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components. Data ports always appear in a group together with a single trigger port, and the ports in the same group are read and written together in a single atomic action. In addition to simple connections from output- to input ports, ProSave contains *connectors* that provide detailed control over the data- and control flow, including forking, joining and dynamically changing connection patterns. For detailed description of these elements, we refer to ProCom language reference manual [5].

Fig. 1 (a) shows the graphical representation of a ProSys subsystem with one input port and two output ports, and (b) shows a simple ProSave component with one input port group and two output port groups. Triangles and boxes denote trigger- and data ports, respectively.

ProCom arcitectural elements have a precise formal semantics [10]. The semantics is described in terms of finite state machines extended with notions of urgency, timing, and priorities. Below, we informally describe the semantics of ProSave elements used in this paper. For through details, we refer the reader to [11].

- $Components$: internally, a ProSave component may be described by code or other inter-connected sub-components. The functionality of a component is captured by a set of services.
- $Services$: the services of a component are triggered individually and can execute concurrently, while sharing only data. A service consists of one input port group and zero or more output port groups, and each port group consists of one trigger port and a number of data ports. When triggered, the input ports are read in one atomic step, and then the service switches to an executing state, where it performs internal computations and writes (atomically) at its output port groups. Before the service returns to idle, each of the associated output port groups must have been activated exactly once. This restriction serves for tight read-execute-write behavior of a service.
- $Connections$: the migration of data or trigger over a ProSave connection is loss-less, atomic, and follows a push model. However, the trigger signals are not allowed to arrive to any port before all data have arrived to all end destinations. This should hold also in case when data passes through a connector. In case more data (trigger) connections are enabled at the same time, the order is non-deterministic.
- $Connectors$: together with connections, connectors can be used to define complex data and control flow for a ProSave composition. ProSave defines different kinds of connectors such as *data fork, control fork, data or, control or, control join* and *selection*. A connector is a stateless component and executes atomically.
- $Clocks$: it is a special type of construct that has one output trigger port, which is activated periodically at a given rate. Clocks are not allowed to drift, but it is not assumed that all clocks are initially synchronized.

Both layers of the ProCom are hierarchical, meaning that subsystems as well as components can be nested. The way in which the two layers are linked together is that a primitive ProSys subsystem (i.e., one that is not composed of other subsystems) can be further decomposed into ProSave components. Thus, a mapping has been defined between the message passing in ProSys and the trigger/data communication used in ProSave. At the bottom of the hierarchy, the behavior of a primitive ProSave component is implemented as a C function.

III. Example: A Temperature Control System (TCS)

As the running example of the paper, we consider a Temperature Control System (TCS), for a heat producing reactor [12]. It has a collection of control rods that can be inserted into the core of the reactor, to control the heat producing (chain) reaction. If inserted, a control rod absorbs neutrons and consequently the reaction is slowed down, with the temperature inside the core decreasing at a fixed rate, depending on the rod inserted. When pulled out, the reaction speeds up, and temperature increases in the core. The main functionality of the TCS is to maintain the temperature in the reactor between the specified `MIN` and `MAX` values. However, when a rod has been used for cooling for a fixed duration, say "T" time units, it is then unavailable for a certain time duration, proportional to T.

In the next section, we propose a language for specification of functional, and timing properties of an embedded system. We use the language to specify the functionality, and timing constraints of the TCS system.

IV. Our Specification Language: Modemachine + Marte CCSL

A specification is a way of explicitly stating system requirements and behavior. In this section, we propose a language for the abstract specification of system functionality, and related timing constraints of an embedded system. We use an extended form of statemachines that we call *Modemachines* (see Fig. 2). A modemachine adds to the original statemachine the system behavior, defined externally, which could be in turn a finite statemachine, a timed automaton etc. Also, in a modemachine, one can specify clock constraints, by using UML/Marte CCSL (Clock Constraint Specification Language) [9]. Graphically, a Modemachine is similar to UML statemachines [8].

A. Modemachine Definition and Graphical Notation

A *modemachine* is a tuple $\langle \mathcal{M}, \mathcal{B}, \mathcal{T}, \mathcal{C}, \mathcal{A}, \mathtt{s} \rangle$, where $\mathcal{M}$ is a set of *modes*, $\mathtt{s}$ is the *entry* mode, $\mathcal{B}$ is a set of externally defined *behaviors*, $\mathcal{A}$ is a set of *events*, $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{A} \times \mathcal{M}$ is the set of transitions between *modes*, and $\mathcal{C}$ is a set of *clock constraints*.

If a mode contains other modes, it is called a *composite* mode. A mode with no internal modes is called an *atomic* mode. The elements of a modemachine are further described below, informally.

B. Modes, and Behaviors

A mode consists of a set of *behaviors*, where a *behavior* denotes the specific functionality of the system. A mode instance is the set of active behaviors at a particular instance of time. Behaviors can be externally specified, for example using external modeling tools such as Matlab/Simulink, UML Rhapsody, etc, or denote the reusable code of system functionalities. Within a mode or submode, behaviors execute concurrently, sequentially, or periodically, based on the associated mode constraints. Mode changes occur when a corresponding event or timeout occurs, or implicitly when all behaviors in the mode terminate. Further, an enabled mode change due to a timeout, has higher precedence over other simultaneously enabled mode changes, if any.

C. Events, Triggers, and Timeouts

The execution of a behavior is triggered by the occurrence of an external event or time event. For an embedded system, the external *events* are generated by its *environment* consisting of sensors and actuators. A *trigger* denotes a periodic time event, and it is usually generated by "system clocks" (e.g., IdealClock in UML/Marte, for measurement of discrete chronometric time). Triggers can be used to specify periodic behaviors within a mode. A *timeout* denotes the expiry of the specified amount of (discrete) time duration. Timeouts

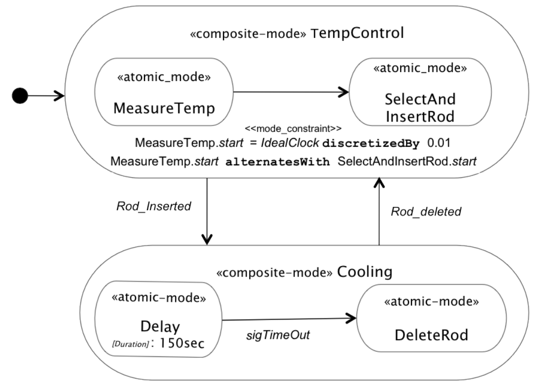


Figure 2. Modemachine specification of a temperature control system.

are useful to model delays associated with an embedded system. A timeout can be associated with atomic modes, making them delay in particular states of the system model. The expiry of a timeout is denoted by the internally signaled `sigTimeOut` event.

D. Mode constraints using UML/Marte CCSL

The recently adopted UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)[9] provides necessary and relevant features for modeling software of the real-time and embedded domain. Further, it aims at bringing interoperability between existing languages and formalisms of the real-time embedded domain. MARTE defines an expressive *Time-Model* for a generic timed interpretation of UML models. CCSL is a language annexed to MARTE specification. It is a declarative language that specifies constraints imposed on the clocks, i.e., both physical and logical, denoting the activation conditions of a model. Some of the constraints used in the paper are briefly discussed below:

- $discretizedBy$: specifies a discrete clock from a dense chronometric clock (e.g., *IdealClock* defined in Marte Time package). Expression (1) below defines a clock, whose period is 0.01 $\mathtt{s}$, where $\mathtt{s}$ is the time unit of the IdealClock.

$$IdealClock \ \mathtt{discretizedBy} \ 0.01 \qquad (1)$$

- $isPeriodicOn$: specifies a discrete clock from another discrete clock of finer granularity (or faster clock). Expression (2) below defines a discrete clock that ticks 10 times slower than C (a tick of C' comes with every 10th tick of C):

$$C' \ \mathtt{isPeriodicOn} \ C \ \mathtt{period} \ 10 \qquad (2)$$

- $alternatesWith$: implies causality between two clocks. Expression (3) states that each instance of clock C precedes and causes the corresponding instance of clock C'.

$$C \texttt{ alternatesWith } C' \tag{3}$$

- *NFP_duration*: supports the description of duration values with respect to an ideal chronometric clock. A *NFP_Duration* value is defined, in the non-functional types model library in Marte (i.e., MARTE::BasicNFP_Types), as a tuple containing a real value and a time unit.

Marte constraints in (1) and (2) are related to the basic synchronous constraint `coincidesWith` (also denoted by "=") which can also be used in specifying a mode constraint. The Marte constraint `alternatesWith` (see (3) above), is useful to specify constraints over logical clocks (e.g., non-periodic event occurrences, beginning and termination of behavior paths, etc), for instance, to specify causality dependencies between behavioral paths. Another useful Marte constraint for component models is *delayedFor* (e.g., "a `delayedFor` n `on` b", i.e., every n^{th} tick of b following a tick of a). Together with *precedes* relation ($\preceq$), it can be used to specify complex timing constraints of particular behaviors, like timing relationships between the start and the end of a behavior (e.g., B1.*finish* $\preceq$ (B1.start `delayedFor` 3 `on` C)).

E. Example Specification: TCS Modemachine

A *modemachine* specification of TCS is given in Fig. 2. At the top level, it contains two composite modes TempControl, and Cooling. Transitions between the modes are enabled with occurrence of events Rod_inserted, and Rod_deleted.

The composite mode TempControl contains two atomic submodes MeasureTemp, and SelectAndInsertRod. Further, the submode MeasureTemp contains a periodic behavior as specified by the associated Marte constraint `discretizedBy`. Also, the sequential dependency i.e., causality between behaviors of MeasureTemp, and SelectAndInsertRod is specified by the Marte constraint `alternatesWith`. SelectAndInsertRod contains the detailed behavior based on the data inputs received, for e.g., a rod selection and insertion is skipped when the current temperature value, communicated by the MeasureTemp component, is within the specified interval `MIN` and `MAX`, as described in Section III.

The duration of the delay is specified using the Marte NFP_duration property. For instance, the composite mode Cooling contains a *delay* mode Delay characterized by a duration of 150 sec. When the timeout expires, it triggers the atomic mode DeleteRod.

Now let us assume a repository of ProCom components, which should be used for the architectural design of the TCS. Therefore, we will transform the above modemachine into a component-based design. To accomplish this, we need to tackle the following design issues/challenges:

- How to transform a composite mode with periodic, and sequential behaviors, into a component-compliant description?
- How to transform the control structure of a modemachine e.g. event, and signal based transitions?
- How to represent timeout in a component based-design?
- How to integrate different design aspects into a complete system design?

In order to address the above design issues, we introduce a set of component patterns that guides a designer in transforming a modemachine-based specification, e.g, the specification model of the temperature control system (TCS) in Fig. 2, into a corresponding ProCom based architectural design. The patterns are described in the next section.

V. Component Patterns

The component patterns, proposed in this section, provide simple mechanisms for modeling the time, and event based executions of system *behaviors* through reusable, easy to understand component designs. The patterns are described in ProCom component language (see Section II). To illustrate our approach, we apply the proposed pattern-based support, in transforming the elements of the *modemachine* specification of the TCS system (see Fig. 2), into a corresponding design aspects in ProSave.

For ProCom-based pattern descriptions, we assume that the components are triggered, where necessary, by a clock, say *MainClock*, of fixed periodicity, say "P". In turn, the *MainClock* itself can be defined by the clock pattern (described below) using the *IdealClock* from the Marte time library. The *MainClock* is denoted by the conventional clock-icon symbol in the pattern descriptions below. Further, we specify additional constraints on the resulting designs (referred as "pattern constraints"), if any, by the pattern description (in a dotted text box, e.g., Fig. 3).

The set of patterns proposed below, address the design issues identified earlier, in the previous section: the "Timer Pattern" characterizes a time out in a component based design; the "Discrete Clock Pattern" addresses the clock synchronization problem between clocks of different granularity; the "Periodic Behavior Pattern" represents the design of periodically executed components; finally the "Controller Pattern" addresses event-triggered executions in a component-based design, and the development of a complete system design.

A. Timer Pattern

Timers, and timeouts constitute important aspects of an embedded system behavior. The pattern models a timeout (or delay) behavior of a system or its parts. It is triggered by a discrete, chronometric clock e.g., the `IdealClock`

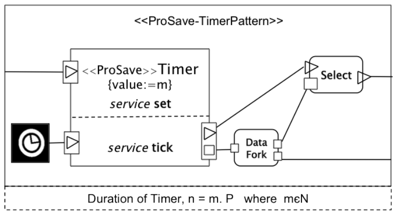


Figure 3. The timer pattern in ProSave.

in UML/Marte time package. When triggered, time is internally measured (using a state variable) until the specified duration/delay units are expired. The output, i.e., the time-out `sigTimeOut`, is indicated as both data and control. This facilitates using the timeout as either sampled data or reactive trigger (design choice based on the specified timing constraints). Further, the pattern specifies the timer mechanism set to assign the timer value (the value itself can be assigned statically or dynamically).

A ProSave description of the pattern is given in Fig. 3. The component Timer contains two services set, and tick. The service set, when triggered, sets the timer value (based on the statically assigned duration value through corresponding data port). The service tick, the periodic behavior triggered by the *MainClock*, decrements the timer value, if set, during each execution cycle and generates a timeout (denoted by `sigTimeOut`) when the value becomes zero. The connectors Select, DataFork are required to differentiate the final timeout output from trigger outputs corresponding to individual executions of Timer component (due to the semantics of a ProSave component).

The timer pattern corresponds to the Marte clock constraint in (4) below, where *n* is a natural number, and **s** is the time unit. However, the timer pattern suffers from a *jitter* of one period of the triggering clock i.e., the *MainClock* (as verified in Section VI). This implies the need for choosing a suitable granularity for the *MainClock*. Further, this should be taken into consideration while evaluating other timing aspects of the design, such as, end-to-end latency.

$$NFP\_duration = n \ \ \mathbf{s} \tag{4}$$

Application of the timer pattern to TCS: The pattern can be applied to transform a *delay* mode (i.e. an atomic mode with NFP_duration value) of a modemachine into a corresponding design in ProSave. For example, in TCS specification, the internal mode Delay (within the composite mode Cooling, see Fig. 2), is translated into a ProSave design, as shown in Fig. 4. Further, if the delay mode is not connected by a transition to any other internal mode, its timeout i.e., `sigTimeOut` is communicated to the controller (described below, by the *controller pattern*) of a containing composite mode. For TCS, the *time-out* from Delay mode triggers the mode DeleteRod.

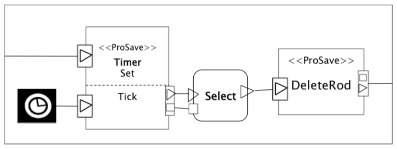


Figure 4. Transformation of the composite mode Cooling of TCS into a ProSave Design, by applying the timer pattern.

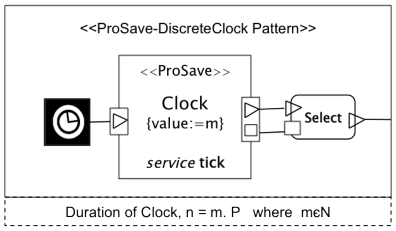

<<ProSave-DiscreteClock Pattern>>
<<ProSave>>
Clock
{value:=m}
service tick
Select
Duration of Clock, n = m. P where m∈N

Figure 5. The discrete clock pattern in ProSave.

B. Discrete Clock Pattern

Clocks are central to embedded system behavior. The pattern models a coarse-grained discrete clock (i.e., a slower clock) triggered by a finer-grained clock (e.g., the *MainClock*). Also, the pattern facilitates the synchronization of various clocks within a component-based design.

The pattern is similar to the timer pattern described above, but does not require a set operation (as the state variable is simply incremented, when triggered). Further, unlike the timer pattern, the output of a discrete clock pattern is always a trigger rather than data as this is justified by the fact that clock *ticks* represent causality, between the clock and the triggered component, in a component-based design.

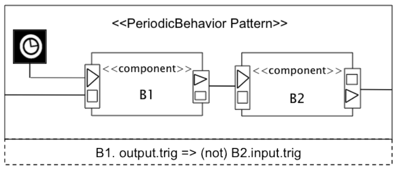
A ProSave description of the pattern is given in Fig. 5. The service tick, when triggered by the finer-grained clock, e.g., *MainClock*, increments the value of the state variable, modulo **m** (see the associated pattern constraint). The connector Select is needed to output the trigger only when the specified period expires, indicated by the associated boolean data output port. Hence, the final output trigger corresponds to a *tick* for every **m** ticks of the triggering clock.

The discrete clock pattern corresponds to the following Marte clock constraints as shown in (5), (6) below. Except for the initial *tick*, the discrete clock pattern does not suffer from any *jitter* (as verified in Section VI). This is consistent with the ProSave clock semantics.

$$MainClock \ \texttt{discretizedBy} \ n \tag{5}$$

$$\texttt{isPeriodicOn} \ MainClock \ \texttt{Period} \ P \tag{6}$$

Application of the discrete clock pattern to TCS: The pattern can support the design of periodic behaviors in a component-based design, also described by the following patterns below. Additionally, it can be used to synchronize different clocks in a design. This not only simplifies the



<<PeriodicBehavior Pattern>>
<<component>>
B1
<<component>>
B2
B1. output.trig => (not) B2.input.trig

Figure 6. The periodic behavior pattern in ProSave.

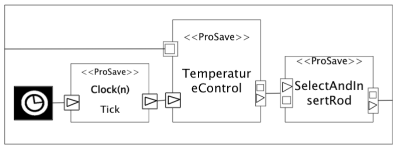


Figure 7. Transformation of composite mode TempControl of TCS into a ProSave design, by applying the periodic behavior pattern.

design, increasing its readability, and understandability, but avoids clock jitters and corresponding unpredictable delays, if any, which can be caused by different clocks. In the case of TCS component-based design, as shown in Fig. 11, the MainClock triggers both the Controller component, and the Timer component. Additionally, it could also trigger the Clock component, which instead is triggered by the Controller, for further simplification of the design.

C. Periodic Behavior Pattern

An embedded system is commonly described as a collection of periodic *behaviors*. The pattern describes two *behaviors*, say B1, and B2, where the periodic behavior B1 triggers the execution of B2. This causality makes the behavior B2 sequential, and also periodic. However, it is generally important for behavior B2 to act on the output generated from B1, which entails the constraint that "B2 *must be* at idle state when B1 completes the execution".

In Fig. 6, we give the ProSave description of this pattern. The component B1 (containing the *behavior* B1) is triggered by a clock of corresponding periodicity (can be designed using the discrete clock pattern described above). Further, the output of B1 triggers the component containing the *behavior* B2. However, the design must ensure that the specified pattern constraint is preserved. That is, B2 must be *idle*, when B1 completes its execution. The formal verification of the pattern (see Section VI), verifies the conditions for the constraint to hold, in terms of the period of B1, and also the end-to-end response time of B1, and B2.

The pattern corresponds to the following Marte clock constraint in (7) below.

$$B1.\mathit{finish}\ \texttt{alternatesWith}\ B2.\mathit{start} \quad (7)$$

Application of the periodic behavior pattern to TCS: The pattern can be used in transforming a mode with periodic behaviors into corresponding component-based design. In the TCS modemachine (Fig. 2), the composite mode TempControl contains atomic submodes with periodic, and sequential behaviors MeasureTemp, SelectAndInsertRod, respectively. Thus, the composite mode TempControl can be translated into a ProSave design as shown in Fig. 7. The Clock component is designed by the application of the discrete clock pattern, and based on the periodicity of MeasureTemp mode behavior (represented by TempControl component), as specified by the corresponding timing constraint (see Fig. 2). When triggered by the Clock component periodically, the TempControl executes by reading the temperature data and provides output, temperature deviation within the allowed interval. This value is read by SelectAndInsertRod component to determine if a control rod is required to be inserted or not.

D. Controller Pattern

The behavior of an embedded system consists mainly of event-, or time-triggered *behaviors*. We have already covered the time-triggered behaviors by the patterns described above. Here, we introduce the controller pattern, to describe the event triggered execution of system *behaviors*. This corresponds to the reactive part of the system behavior.

In principle, a component-based design is based on time-triggered, and control-, data-flow semantics. Hence, the pattern transforms event-based execution of mode *behaviors* into the executions based on sampling of the environment data corresponding to sensors, and actuators. Within a component-based design, events can be represented by predefined predicates over the environment data [13]. When triggered by a system clock (e.g., the *MainClock* discussed previously), the data is sampled to determine the occurrence of events, through the evaluation of corresponding predicates.

Fig. 8 shows the ProSave design of the pattern. The Controller component is triggered by a system clock, e.g., *MainClock*, periodically (in an implementation, the controller thus becomes a periodic task in the system). The periodicity of the clock is to be determined by the periodicity of data occurrences or their criticality. Also, there can exist multiple clocks of different periodicity (can be or-ed using ProSave connector ControlOr). Further, the controller implements the mode change behavior of a modemachine(e.g., Fig. 2). It also implements a datastructure representing the predicate-event mapping ([13]) described above. The controller can be triggered by internal signals i.e., `sigTimeOut`, when the signal represents a trigger rather than data (as described in the Timer pattern previously).

Application of the controller pattern to TCS: The pattern can be used in transforming the control structure of a modemachine specification into corresponding component-based design. For example, the event-based transitions corresponding to the top-level control structure of the TCS

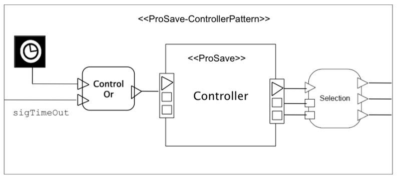


Figure 8. The controller pattern in ProSave

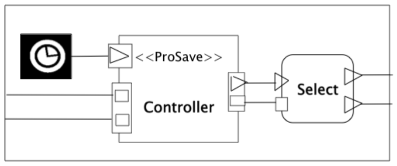


Figure 9. Transformation of the top level mode transitions of TCS into a ProSave design, by applying the controller pattern.

modemachine specification (Fig. 2), is transformed into the corresponding component-based design in Fig. 9. When triggered, the Controller evaluates the predicates corresponding to the event occurrence Rod_inserted, and Rod_deleted, respectively. In this case, no timeout is communicated to the controller, hence using the control-or connector, shown in the pattern, is not required.

VI. Pattern Verification

In this section, we describe the formal verification with respect to component patterns, presented in the previous section. The approach is based on the formal semantics of the architectural elements of the ProSave language [10], [11]. The formal semantics is based on an extension of finite state machine formalism with the notions of urgency, priority etc. The semantics of the formalism itself was given in terms of timed automata (TA) [14]. This provides a mechanism to formally verify ProSave designs using UPPAAL, the timed automata based model-checker [7] .

For formal verification, a component pattern is translated into the corresponding network of timed automata, based on the underlying semantics of constituting ProSave elements.

A. *Verification of periodic behavior pattern*

For periodic behavior pattern (in Fig. 6), the corresponding network of timed automata is given in Fig. 10. Each of the timed automata ClockTA, ClockToB1, B1, B1ToB2, B2 correspond to the periodic trigger to B1, trigger connection to B1, component B1, trigger connection from B1 to B2, and component B2, respectively. Also, the end-to-end response time (say, R) of components B1, B2 are modeled in corresponding TA (3, 4 in this example). ChannelsTA denotes the timed automaton that contains complementary channels, corresponding to urgent channels A, B of other TA.

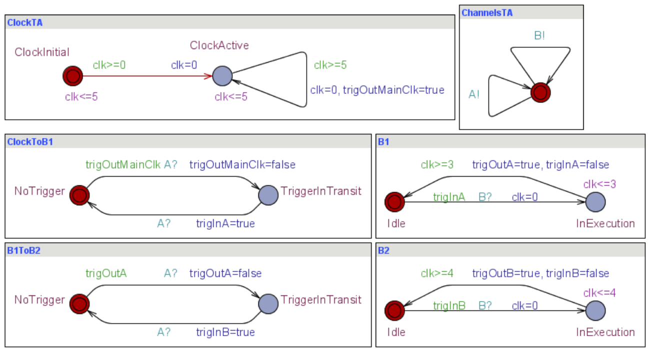


Figure 10. Translation of the periodic behavior pattern in ProSave into the corresponding network of timed automata.

On the above models, we have verified with UPPAAL [7], that the following properties are satisfied by the periodic behavior pattern:

$$\texttt{A[] not deadlock} \tag{8}$$

$$\texttt{A[]}\ \ B1.trigOut\ \texttt{imply}\ (\texttt{not}\ B2.InExecution) \tag{9}$$

Property (8) states that there is no deadlock in the pattern. Though basic, this is a very important model feasibility property. Property (9) verifies the main constraint of the pattern i.e., that B2 is idle, that is, ready to begin execution, whenever B1 terminates its execution. However, it is observed that this property is satisfied, provided that the following conditions hold:

$$\text{Periodicity of B1}= \begin{cases} \leq E_{B1} & \text{if} \quad E_{B1} > E_{B2} \\ > E_{B2} & \text{if} \quad E_{B2} > E_{B1} \end{cases} \text{where}$$

E_{B1}, E_{B2} denote the response time of B1, B2 respectively.

B. *Verification of other Patterns*

In addition to "no deadlock", we have verified other properties like the ones expressed by (10 - 12), this time for the Timer pattern (see Fig. 3). Expression (10) describes a liveness property (also called *leads to*, or *response* property [7]), as follows: when the timer is set, it eventually performs a timeout. Formulae (11), (12), verified to be satisfied by the pattern, indicate that the timer duration has a jitter of one period of the *MainClock*.

$$timerValue == Set.N \leadsto timerValue == 0 \tag{10}$$

$$\texttt{A[]}\ (TimeOut\ \texttt{imply}\ (obsClk >= (m-1)*P)) \tag{11}$$

$$\texttt{A[]}\ (TimeOut\ \texttt{imply}\ (obsClk <= (m+1)*P)) \tag{12}$$

For the discrete clock pattern, we have model-checked the corresponding network of automata against the following properties: deadlock freedom, liveness (as given by (10)), and jitter freedom (see (13)). The latter means that the Clock discretizes the *MainClock* perfectly, without any jitter, unlike the timeout duration of the timer pattern:

$$\texttt{A[]}\ TimeOut\ \texttt{imply}\ (obsClk = m * P) \qquad (13)$$

VII. Temperature Control System: A Complete ProCom Design

In this section, we complete the approach introduced in sections IV and V, by describing the final steps leading to a complete ProCom system design.

We have applied the *Timer* pattern, of the Fig. 3, to transform the composite mode Cooling of the TCS specification (Fig. 2), which contains the delay mode Delay, and an atomic mode DeleteRod with its corresponding *behavior*. The expiration of the timeout, signaled by the event `sigTimeOut` from the Timer component, triggers the execution of the *behavior* in DeleteRod.

To recall, Fig. 5 presents a ProSave design corresponding to the composite mode TempControl in the TCS specification. The corresponding design in ProSave, is obtained by applying the discrete clock pattern to the composite mode.

Also, the top level control structure, corresponding to the reactive behavior of the TCS modemachine (Fig. 2), with respect to the events Rod_inserted, and Rod_deleted is translated into the corresponding ProSave design in Fig. 8, through the controller pattern.

The complete ProCom design of the TCS is presented in Fig. 11. For simplicity, the complete system is represented as a single ProSys subsystem (see Section II). For integrating different design parts, for instance, those described above, the following design steps are applied:

- *Synchronize the clocks using discrete clock pattern*: different clocks in the component-based design can be synchronized by applying the discrete clock pattern, and a finest-grained clock, e.g., the *MainClock*. This not only simplifies the component design, but also minimizes clock jitters, if any. For TCS, the different clocks, due to Controller, PeriodicBehavior, and Timer patterns, are synchronized using the MainClock.
- *Interconnect ProSys message ports with ProSave ports*: the system can be designed as a basic ProSys system (also called ProSave Subsystem) by connecting message ports to ProSave control, and data ports (as shown in Figure 11). Sensor and other data values, received as messages through ProSys message ports are forwarded to the internal ProSave components through their ports.

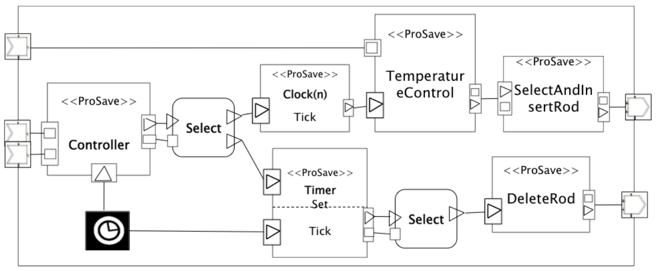


Figure 11. The Temperature Control System in ProCom: a ProSys component made of ProSave components.

VIII. Related work

In the domain of synchronous languages [15], mode automata and the notion of running modes have been introduced, to reduce the gap between the initial design of a system and the program written for it. The formalism has been proposed to support both dataflow, and imperative styles. The *modemachine* described in this paper corresponds to the event-based, hierarchical, high-level control structure of the system and associated timing constraints.

Sandén proposes the "state-machine" pattern [16], for designing concurrent real-time software in Ada [17]. Many possible implementations of the pattern, corresponding to concurrent, reactive, and time-triggered behaviors, are described. Also, patterns for non-functional aspects such as resource usage, quality-of-service have been proposed [18]. However, such patterns focus on the design or implementation phase of the system. The patterns proposed in this paper support the design process, by directly mapping the specification aspects, with associated timing constraints, into the corresponding design elements.

Maxwell et al. have proposed a formal framework [19] for heuristics-based transformation of architectural designs. The authors capture heuristics in a structured and formal manner, such that the architectural transformations can be performed for optimizing the non-functional qualities of a system. Denford et al. have proposed an architectural refinement method [20] that focuses on non-functional requirements e.g., reliability, performance, while still addressing the functional requirements. While these works focus on non-functional aspects such as performance, we address architectural designs through timing constraints of embedded systems. However, this is done by including the functional requirements also.

UML/Marte profile is extensively used in the context of AADL (Architecture analysis and design language [21]) for component-based designs of real-time, embedded systems [22], [23]. AADL supports the modeling of both software components such as thread, subprogram, process, and platform components, e.g., bus, memory, processor, and device. However, AADL introduces avoidable redundancies that obscure the model and may even lead to design inconsistence. To address this deficiency, the Marte clock constraints have been used [23] to precisely specify both event, and time triggered communications for AADL models, and to compute end-to-end flow latency. These works focus on models related to software and platform mapping. In this paper, we offer formally verified support for component-based system design, in the form of patterns based on timing constraints.

EastADL [24] is a layered architecture language for model-based development of automotive software. To address various concerns of system's life-cycle development, it provides abstraction layers such as feature level, requirements, analysis, design, and implementation. Mallet et al. have described Marte CCSL specification of EastADL timing requirements [25]. This enables the use of Marte tools for timing verification of EastADL requirements. The work is similar to model driven aspects underlying the proposed patterns in this paper.

IX. Conclusions

In this paper, we have proposed a set of component based patterns for developing embedded system designs. The patterns are based on the specification of reactive, and time-triggered behaviors of an embedded system. An extended form of statemachine, referred as modemachine, combined with UML/Marte clock constraints is used as the specification language. We have proposed component patterns for clocks, timers, periodic, and reactive behaviors. Also, we have described the implementation of the proposed patterns in the ProCom language, in order to support the design process based on the specification of functionality, and timing constraints. Further, we have described the correspondence of the proposed patterns with related UML/Marte clock constraints.

To guarantee timing correctness aspects, we have formally verified our patterns, by model checking their corresponding timed automata models, in UPPAAL. This facilitates the development of component based-design models with precise timing aspects. We have demonstrated the approach, by transforming the modemachine specification of an example temperature control system, into a corresponding design in ProCom component model. The explicit representation of running modes in the design, by application of the proposed patterns, may be useful for developing efficient deployment models. However, this requires further validation. Also, we intend to extend the approach to other Marte constraints, and validate the approach with complex systems. Further, we plan to work on the compositional verification of timing properties of the resulting component-based system designs.

Acknowledgment

This research work was partially supported by Swedish Research Council (VR), the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and by the European Union under the ICT priority of the 7th Research Framework Programme (FP7) in the context of the Q-ImPrESS research project.

References

[1] R. Englander, *Developing Java Beans*. O'Reilly, 1997.

[2] R. van Ommering, F. van der Linden, and J. Kramer, "The Koala component model for consumer electronics software," in *IEEE Computer*. IEEE, March 2000, pp. 78–85.

[3] T. Bureš, P. Hnetynka, and F. Plasil, "SOFA 2.0: Balancing advanced features in a hierarchical component model," in *Proceedings of SERA 2006*. IEEE CS, August 2006, pp. 40–48.

[4] F. Plasil, D. Balek, and R. Janecek, "SOFA/DCUP: Architecture for component trading and dynamic updating," in *Proceedings of ICCDS 98*. IEEE CS, May 1998.

[5] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis, "ProCom – The Progress Component Model Reference Manual, version 1.0," Mälardalen University, Technical Report MDH-MRTC-230/2008-1-SE, June 2008.

[6] T. Bureš, J. Carlson, S. Sentilles, and A. Vulgarakis, "A component model family for vehicular embedded systems," in *Proceedings of the Third International Conference on Software Engineering Advances*. IEEE, October 2008.

[7] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, Oct. 1997.

[8] Object Management Group (OMG), "UML 2.0 Superstructure Specification, The OMG Final Adopted Specification," 2003.

[9] OMG, "A UML Profile for MARTE, Beta 1," August 2007, document number: ptc/07-08-04.

[10] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson, "Formal semantics of the ProCom real-time component model," in *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, August 2009.

[11] J. Suryadevara, A. Vulgarakis, J. Carlson, C. Seceleanu, and P. Pettersson, "ProCom: Formal semantics," Mälardalen University, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-234/2009-1-SE, March 2009.

[12] C. Seceleanu, A. Vulgarakis, and P. Pettersson, "REMES: A REsource Model for Embedded Systems," in *In Proc. of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. IEEE Computer Society, June 2009.

[13] J. Suryadevara, E.-Y. Kang, C. Seceleanu, and P. Pettersson, "Bridging the semantic gap between abstract models of embedded systems," in *13th International Symposium on Component Based Software Engineering (CBSE)*, L. Grunske and R. Reussner, Eds. Springer LNCS, vol 6092, June 2010.

[14] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[15] F. Maraninchi and Y. Rémond, “Mode-automata: a new domain-specific construct for the development of safe critical systems,” *Sci. Comput. Program.*, vol. 46, pp. 219–254, March 2003.

[16] B. I. Sandén, “The state-machine pattern,” in *Proceedings of the conference on TRI-Ada ’96: disciplined software development with Ada*. New York, NY, USA: ACM, 1996, pp. 135–142.

[17] A. Burns and A. Wellings, *Concurrency in Ada*. Cambridge University Press, 1995.

[18] J. P. Loyall, P. Rubel, R. Schantz, M. Atighetchi, and J. Zinky, “Emerging patterns in adaptive, distributed real-time, embedded middleware,” in *9th Conference on Pattern Language of Programs*, September 2002.

[19] C. Maxwell, T. O’Neill, and J. Leaney, “Formal architecture transformation using heuristics,” in *Engineering of Computer-Based Systems, 2007. ECBS ’07. 14th Annual IEEE International Conference and Workshops on the*, March 2007, pp. 15 –24.

[20] M. Denford, J. Leaney, and T. OŃeill, “Non-functional refinement of computer based systems architecture,” in *Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 168–. [Online]. Available: http://portal.acm.org/citation.cfm?id=998673.999275

[21] Society of Automotive Engineers (SAE). (2006, June) Architecture analysis and design language (AADL). [Online]. Available: http://www.sae.org/technical/standards/AS5506/1

[22] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard, “MARTE: Also an UML profile for modeling AADL applications,” in *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, 2007, pp. 359 –364.

[23] F. Mallet, R. de Simone, and L. Rioux, “Event-triggered vs. time-triggered communications with UML MARTE,” in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, 2008, pp. 154 –159.

[24] ATESST (Advancing Traffic Efficiency through Software Technology). (2008, March) East-ADL2 specification. [Online]. Available: http://www.atesst.org, 2008-03-20

[25] F. Mallet, M.-A. Peraldi-Frati, and C. Andre, “Marte CCSL to execute East-ADL timing requirements,” in *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC ’09. IEEE International Symposium on*, March 2009, pp. 249 –253.