Connecting ProCom and REMES

Aneta Vulgarakis, Séverine Sentilles, Jan Carlson, and Cristina Seceleanu
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{aneta.vulgarakis, severine.sentilles, jan.carlson, cristina.seceleanu}@mdh.se

Abstract

When component-based development is applied in the domain of distributed embedded systems, where applications are often safety-critical and subject to real-time constraints, it is of significant importance that reliable predictions of functional and extra-functional properties can be derived at design-time. Preferably, analysis should be performed in early development phases, where the cost of modifying the design is lower. Centered on an example application from the automation domain, we show how a component model specifically intended for embedded systems can be combined with a language for high-level formal behavior modeling. This allows modeling the behavior of individual components, in terms of functionality, timing and resource usage. In turn, this permits analysis of system level properties, while also supporting reuse of behavioral models when components are reused.

1 Introduction

As pointed out by Henzinger and Sifakis, designing embedded systems is not a straightforward application of either hardware or software design methods [11]. An embedded system involves computation that is subject to physical constraints. Some constraints refer to bounded resources like available processor speed, power, etc., which are derived from the implementation platform; on the other hand, constraints such as deadlines fall into the category of timing (execution) constraints, originating from the behavioral requirements. Therefore, to ensure its predictable behavior, an embedded system design needs to be formally checked against different requirements pertaining to various kinds of constraints including functional, timing, safety, and resource-usage constraints.

Designing an embedded system in a component-based manner, by building it from well-specified and verified components, intends to lower its complexity, in terms of implementation, but also modeling and analysis. Here, we adopt such a design perspective and describe the architecture of our embedded systems following the real-time component model ProCom [20]. The component model is structured in two layers, called *ProSys*, and *ProSave*, each addressing the concerns of a different level of granularity (see Section 3). The possibly complex extra-functional behavior of ProCom components is modeled in a dense-time, state-based hierarchical language called REMES [18]. REMES fits a component-based perspective on embedded system development, and is appropriate for modeling timing and resource-wise behaviors of components described by *modes*, as explained in Section 3.

In this paper, one of the contributions is packing a ProSys component, annotated with attributes, such as required resources, with its behavioral model represented in REMES. Then, both the interface and internal models of component behavior are seen as the actual reusable unit of composition, which can be employed as such, without modification, in adequate design contexts. To accomplish this in Section 4, we propose a way of mapping the ProSys component interface, consisting of incoming and outgoing messages used for asynchronous communication, onto the entry and exit variables of REMES modes, respectively, such that the two models become connected.

Most of the component-based system analysis frameworks known in the literature [5, 21] focus on either analyzing extra-functional behavior only, abstracting from checking correctness of system functionality, or just on analyzing interface behavior of components, via contracts [15]. This work contributes also by showing how to support formal analysis of various properties belonging to different categories (functional, safety, timing, resource-usage) within the same framework. This capability, together with the component-based design approach, might improve the efficiency of embedded system design, by allowing reuse of both models and analysis results. We show the modeling and analysis approach on a *turntable* example system introduced in Section 2, which we model as a collection of ProSys components that we connect to their associated behavioral REMES models. The latter are then formally analyzed against functional, safety and resource-related properties, as described in Section 5, on the underlying Priced

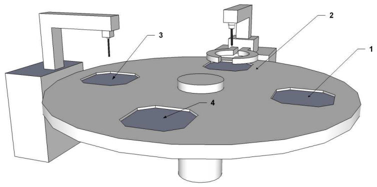


Figure 1. The turntable system (load and unload stations are not shown).

Timed Automata models. In Section 6 we compare to some of the relevant related work, before Section 7 concludes the paper.

2 Example: The turntable system

As an example, we have considered the turntable drilling system previously described by, e.g., Bos and Kleijn [7] and Bortnik et al. [6]. The system, depicted in Figure 1, consists of a rotating table that moves products between processing stations where they are drilled and tested.

The load station places new products on the table (1), after which they are moved to the drill station (2) by rotating the turntable 90°. Drilling requires that the product is securely held in place by a clamp mechanism. After drilling, the product is moved to the testing station (3) where the depth of the drilled hole is measured. Finally, the unload station (4) removes the product from the table, provided that it passed the test. If not, it remains on the table to be drilled and tested again. The turntable has four slots, each capable of holding one product. Thus, the stations can operate in parallel, so that while the first piece is being tested, a second piece can be drilled, etc.

Table 1 lists a few representative system requirements addressing functional correctness, timing and resource consumption, selected to include examples of both generic, local and system-wide properties. These properties will be used to illustrate how the proposed approach allows different types of analysis to be performed. In Section 5, we show how the architecture of the turntable system can be modeled in ProCom, and how modeling the behavior of each component in REMES allows us to verify that these requirements are satisfied.

Table 1. Requirements and properties of the turntable system.

Generic requirements:	
1	The system should be free from deadlocks.
Functional requirements:	
2	A product must be clamped when drilled.
3	The table should never turn when one of the stations is operating.
Timing requirements:	
4	Processing five products should never take more than 25 seconds (assuming that at most one drilling fails).
Resource properties:	
5	What is the minimum energy consumption for processing five products?

3 ProCom and REMES

3.1 The ProCom component model

ProCom [20] is a component model for design and development of distributed embedded systems, addressing key characteristics and development concerns inherent to the domain. In particular, ProCom considers the need for support throughout the development process, from early design to deployment, and for addressing the different concerns that exist when considering the system as a collection of complex and distributed functionalities on one hand, and the low-level control functionality of the individual parts on the other. As a consequence, ProCom is structured in two layers, each layer addressing the concerns of the corresponding level of granularity: The upper layer, called ProSys, is intended for modeling complex, active, concurrent and typically distributed subsystems, communicating via asynchronous message passing. The lower layer, called ProSave, serves for modeling of non-distributed, passive and small units of functionality, closer to function blocks or tasks. The two layers are not independent but relate to each other in the sense that a ProSys component can be modeled out of ProSave components. For more details, see [8].

This paper only addresses the top layer (ProSys) of the component model, although the intention is to use REMES for modeling the behavior of individual components also in the lower layer. Figure 2 exemplifies the graphical notation of ProSys.

3.2 Attribute framework

In ProCom, different functional and extra-functional characteristics can be associated as attributes with components, their services, ports, subcomponents, etc. The attributes may span from single number (e.g., static memory usage of a component) to complex models.

The attribute framework [19] provides a systematic way to support the management and integration of extra-functional properties during the development of a compo-

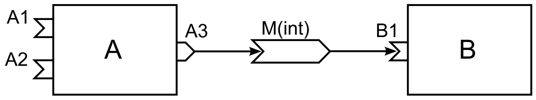


Figure 2. Example of the ProSys notation: Component A has two input ports (A1 and A2) and one output port (A3), and communicates with component B by sending messages (each carrying an integer) over the message channel M.

nent or a system. In it, extra-functional properties are represented by attributes consisting of a unique identifier and one or more values. The complete list of the attribute types that are available during the development is stored in an attribute registry together with the specification of each attribute, that is (i) the list of entities to which this attribute can be attached, and (ii) the valid format for its values (e.g. integer, interval, model, etc.). Providing that it is authorized by its specification, an attribute can be associated with any entity of a component model such as component, message port, connection or even component instance.

3.3 The REMES language for behavior modeling

In case of more elaborate functional and extra-functional behavior (such as timing and resource consumption), we use a dense-time state-based hierarchical modeling language called REMES [18].

The internal behavior of a component is depicted by a REMES *mode* that can be *atomic* (see Atomic mode 1, Atomic mode 2 in Figure 3) or *composite* (made of atomic modes). The discrete control of a mode is captured by a *control interface* that consists of *entry*- and *exit* points, whereas the data transfer between modes is carried out through a well-defined *data interface* that consists of typed global variables. A composite mode may also have a special *init* entry point where the global variables are initialized.

A composite mode executes by performing a sequence of *discrete steps*, via actions that, once executed, pass the control from the current submode to a different submode. An action, $A = (g, S)$ (e.g., $(y == b, d := u)$ in the figure), is a statement S (in our case $d := u$), preceded by a boolean condition, the *guard* ($y == b$), which must hold in order for the action to be executed and the corresponding outgoing edge taken. A REMES composite mode may contain *conditional connectors* (decorated with letter C) that allow a possibly nondeterministic selection of one discrete outgoing action to execute, out of many possible ones. Below, via C, one of the empty statement actions, $x \leq a \wedge d == v$ or $d \geq v$ can be chosen for execution.

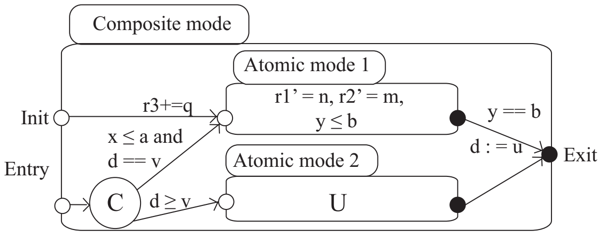


Figure 3. A REMES composite mode.

In REMES one may model timed behavior and resource consumption. Timed behavior is modeled by global continuous variables of specialized type *clock*, that is, x, y in our figure, evolving at rate 1. A boolean condition called *invariant* (e.g., $y \leq b$) may be used to specify for how long an atomic mode can be executed. Once the invariant stops to hold, the current mode is exited. In case a mode is exited instantaneously after its activation, the mode is called *urgent* (decorated with letter U).

The composite mode in Figure 3 has two continuous resources (r1 and r2) and one discrete resource r3. Consumption of the continuous resources is expressed by their first derivatives (r1' and r2'), see Atomic mode 1, which give the rates at which the mode consumes the resources, respectively (r1 is consumed at rate n, whereas r2 at rate m, where m and n are integers). Discrete resources are allocated through usual updates, e.g., $r3 += q$.

For enabling formal analysis, REMES models are semantically transformed into Timed Automata (TA) [2] or Priced Timed Automata (PTA) [3], depending on the analysis goals. We refer the reader to [18] for a thorough description of REMES.

4 Integrating REMES into ProCom

ProCom has been developed to facilitate the expression and analysis of functional and extra-functional properties, but does not, per se, provide any means to actually model them. It needs to be complemented with formalisms, complying with the component-based approach, that enable early formal analysis of relevant concerns. One step towards this support for formal analysis is the integration of REMES, by which functional behavior, resource consumption and timing can be addressed in a single modeling formalism.

Concretely, the integration is achieved by defining a new attribute type in the attribute registry. The attribute type can be attached to ProSys subsystem components, and has an attribute value consisting of a reference to the REMES model file in the component structure.

The relation between the ports of the component and the variables in the REMES model is given by a mapping, described below. As detailed in Section 4.2, we also need to slightly extend the REMES language to fit the active, nonterminating semantics of the ProSys components.

4.1 Connecting component interfaces and REMES modes

The connection between ProSave and REMES is done by mapping ProSave- to REMES interface. Each ProSave trigger port is mapped to a REMES boolean variable, and each ProSave data port is mapped to a REMES data variable of same type as the port type. In our previous work [18], we describe the connection between ProSave components and REMES modes on an abstracted version of a temperature control system. Here, we instead focus on connecting REMES to ProSys components.

Let P be the set of message ports of a ProSys component C. Each port $\mathsf{p}_{\mathsf{i}\in[1\ldots\mathsf{n}]} \in \mathsf{P}$ is a tuple $(\mathsf{Name}, \mathsf{Kind}, \mathsf{Type}, \mathsf{Value})$, where: Name is the port identifier, Kind models the input/output feature of the message port, Type encodes the port's data type, and Value stores the port's actual data value. Further, let M be a REMES mode that depicts the behavior of component C, and V the set of all variables of mode M that correspond to the ports of C. Each variable $\mathsf{v}_{\mathsf{j}\in[1\ldots\mathsf{n}]} \in \mathsf{V}$ is a tuple $(\mathsf{Name}, \mathsf{Kind}, \mathsf{Type}, \mathsf{Value})$, where: Name is an identifier of the variable, Kind distinguishes between read (global variable of the mode that may be written by other modes) and write (global variable of the mode that may be read by other modes) variables, Type encodes the variable's data type, and Value stores the actual variable value. The connection between mode M and the interface of component C is given by a mapping function $\mu : P \rightarrow V$ that maps component ports to REMES mode variables. Assuming non-empty messages ($\mathsf{Value}(\mathsf{p_i}) \neq \mathsf{NULL}$), the mapping is defined as follows:

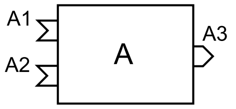
$$\mu(\mathsf{p_i}) = \overline{\mathsf{v}}_\mathsf{i}, \quad \overline{\mathsf{v}}_\mathsf{i} = (\mathsf{v}_{\mathsf{i}_1}, \mathsf{v}_{\mathsf{i}_2}),$$

such that the following boolean condition holds:

$$\begin{aligned}
&\mathsf{Name}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{Name}(\mathsf{p_i}) \wedge \mathsf{Name}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Name}(\mathsf{p_i}) + \text{“\_value''}\\
&\wedge\\
&\quad((\mathsf{Kind}(\mathsf{p_i}) = \mathsf{input} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{read} \wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{bool}\\
&\qquad\wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{false} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{read}\\
&\qquad\wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Type}(\mathsf{p_i}) \wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Value}(\mathsf{p_i}))\\
&\quad\vee\\
&\quad(\mathsf{Kind}(\mathsf{p_i}) = \mathsf{output} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{write} \wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{bool}\\
&\qquad\wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{false} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{write}\\
&\qquad\wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Type}(\mathsf{p_i}) \wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Value}(\mathsf{p_i})))
\end{aligned}$$

In case an empty message is received/sent ($\mathsf{Value}(\mathsf{p_i}) = \mathsf{NULL}$), the mapping function returns $\overline{\mathsf{v}}_\mathsf{i} = (\mathsf{v}_{\mathsf{i}_1}, \mathsf{NULL})$.

The parallel composition of the REMES modes associated to all ProSys components in the given system, together with representations of the ProSys message channels and connections, describe the whole system's behavior. Figure 4 exemplifies the mechanism of connecting the ProSys and REMES interfaces.



ProSys port	REMES variables
A1	bool A1 and float A1_value
A2	bool A2
A3	bool A3 and int A3_value

Figure 4. Example of how ProSys ports are mapped to REMES variables. Component A receives a message of type float via input port A1 and an empty message via input port A2, and it sends a message of integer type through output port A3.

4.2 REMES extensions

The traditional REMES modes described in Section 3.3 run to completion, and as such are suitable for depicting the behavior of ProSave components. In order to be able to capture the active behavior of a ProSys component, yet at the same time to ensure the termination of the internal behavior of a REMES mode, each REMES composite mode is enriched with a special write global variable called *history*, similar to CHARON [1]. Whenever an execution of a REMES composite mode would return to an already visited submode, the composite mode is exited and the control state of that mode is recorded into its history variable. Next time when the composite mode is entered, the control state of that mode is restored according to the value of the history variable. The history variable of a composite mode M contains the names of the submodes of M as values, or a special value null, which denotes that the mode is not active. A submode SM of a composite mode M is called active when the history variable of M has the value SM. Additionally, in the extended REMES for modeling behavior of ProSys systems, guards are modeled as conjunctions of boolean expressions over usual variables, to which constraints on history variables are added. Every time when there is a change in the history variable value of a certain composite mode its reactivation should be ensured. Access to the proper history variable values of every composite mode is done by System[name], where name denotes the name of the composite mode.

In addition, we have enriched REMES with a so called *non-lazy* mode. A non-lazy mode does not contain any invariant to specify how long it is allowed to delay in that mode. Time is allowed to pass in a non-lazy mode until at least one of the guards of the outgoing discrete actions evaluates to true. As such, in order to ensure the exit of a non-lazy mode, the disjunction of the action guards associated to the outgoing edges of that mode should always eventually become true.

5 Example revisited

In this section we describe a software architecture of the turntable drilling system adhering to the ProCom component model. We then show how the behavior — in terms of functionality, timing and resource consumption — of individual components can be modeled in REMES, and how this permits analysis of the whole system to determine whether the candidate design meets the identified system requirements. We consider a traditional component-based development process mixing top-down decomposition with a reuse of pre-existing components in a bottom-up manner [9], and assume that system requirements have already been captured in a previous phase.

5.1 Architecting the turntable in ProCom

Since the different stations are relatively independent, we model each station, and the turntable, with a separate component. Accordingly, we define the *Loader*, *Driller*, *Tester*, *Unloader* and the *Turntable* subsystems in ProSys. However, in order to ensure the synchronization between the stations and the table, e.g., guaranteeing that the table turns only when no processing station is operating (i.e., requirement 3 in Table 1), an additional subsystem is needed: the *Controller*.

Next, the interfaces of these identified components need to be specified. Since the component model has been imposed, the available communication mechanisms between components are restricted to asynchronous message passing for the active and independent parts of the system (synchronous control- and data-flows are available only in the lower layer). At this step, it is possible to browse component repositories to find pre-existing components which functionalities and possibly extra-functional properties that match the requirements. In the case of our turntable system, we assume that the *Loader* and *Unloader* components can be reused from a previous project. For the remaining components, the interfaces remain to be specified. Figure 5 illustrates the component interfaces and shows how components are connected.

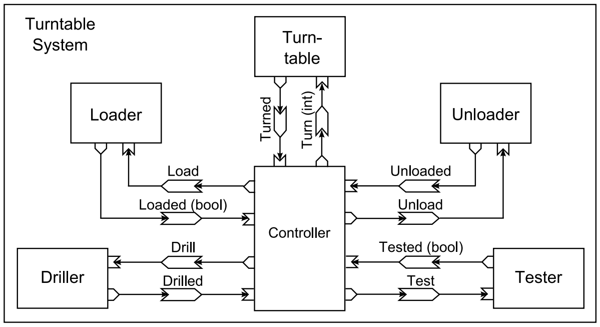


Figure 5. ProCom design of the turntable system.

The *Turntable* component receives a message when the table should be rotated. In order to make the component reusable in different systems (e.g., a turntable with more than four stations), the angle of rotation of the table can be specified in the message. When the table has been turned, a message is sent to inform other parts of the system.

The *Tester* and *Driller* have similar interfaces; an incoming message telling the station to start processing, and an outgoing message indicating that it has finished. The output message of *Tester* also contains a boolean value representing if the test succeeded or not.

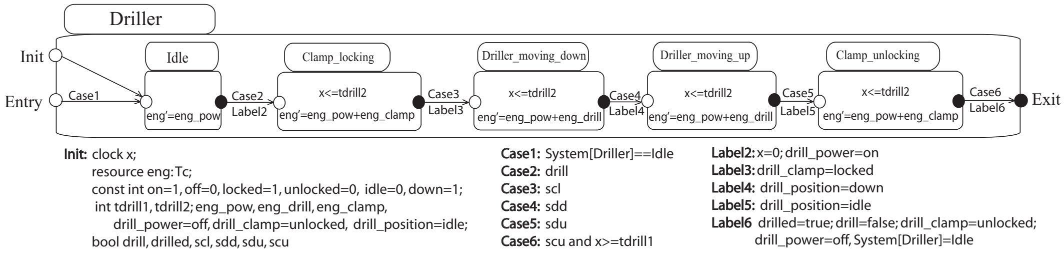
The *Controller* keeps track of the current status of the four slots, and activates stations accordingly, by sending messages to each stations and receiving messages back once they are done. Consequently, the interfaces of *Controller* must be compatible with the interfaces of the stations (including those of the reused *Loader* and *Unloader*).

It is possible to further decompose each of these ProSys components into either smaller ProSys components or into ProSave components according to the level of complexity of the functionality, and the potential for distribution. Before doing that, however, the developer may want to validate the feasibility of the design so far. Some properties can be analyzed from the ProCom design alone, for example that connected ports and channels match, but in order to reason about the requirements identified in Section 2, we need to model the behavior of the components identified so far.

5.2 Behavior modeling in REMES

We model the functional, timing and resource usage behavior of the turntable components as models in REMES. Since the *Loader* and the *Unloader* components are reused, they already have behavioral models, but the remaining components should be given REMES models. Because of space limitation, we only present the REMES models of *Driller* and *Controller*, depicted in Figure 6 and 7, respectively.

The *Driller* component is responsible for moving the



```
Init: clock x;
      resource eng:Tc;
      const int on=1, off=0, locked=1, unlocked=0,  idle=0, down=1;
       int tdrill1, tdrill2; eng_pow, eng_drill, eng_clamp,
          drill_power=off, drill_clamp=unlocked,  drill_position=idle;
      bool drill, drilled, scl, sdd, sdu, scu

Case1: System[Driller]==Idle
Case2: drill
Case3: scl
Case4: sdd
Case5: sdu
Case6: scu and x>=tdrill1

Label2: x=0; drill_power=on
Label3: drill_clamp=locked
Label4: drill_position=down
Label5: drill_position=idle
Label6  drilled=true; drill=false; drill_clamp=unlocked;
        drill_power=off, System[Driller]=Idle
```

Figure 6. The Driller modeled in REMES**.**

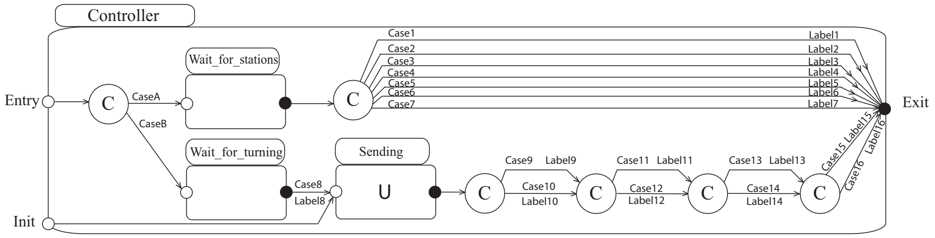


Figure 7. The Controller modeled in REMES**.**

drill up and down and for locking ad unlocking the clamp. In order to do this, it reads values from the drill and clamp sensors, modeled by boolean variables sdu (drill in upmost position), sdd (drill in downmost position), scl (clamp fully locked) and scu (clamp fully unlocked). Neither of the two message ports of *Driller* carries values, and thus they are mapped to two boolean variables drill and drilled, as described in Section 4.1.

The *Driller* remains in the non-lazy mode Idle until receiving a drill message. When this happens, the component goes through a sequence of submodes: Clamp_locking, Driller_moving_down, Driller_moving_up and Clamp_unlocking. Each of these submodes is exited as the result of a sensor value turning true. When exiting the last submode, a drilled message is sent, indicating that the operation is finished.

This REMES model also models the consumption of energy of the *Driller* subsystem. We assume the following: powering the *Driller* consumes eng_pow units of energy per time unit, locking or unlocking the clamp consumes eng_clamp units of energy per time unit and drilling consumes eng_drill units of energy per time unit. Moreover, we assume that the time of each *Driller* operation cycle is bounded to the interval [tdrill1, tdrill2].

The *Controller* component, depicted in Figure 7, keeps track of the states of the four slots and operates the stations and the turntable accordingly by exchanging messages with all of them. The behavior defined by the REMES mode consists of two main submodes, one in which the controller waits for messages from the stations, and one waiting for the turntable to finish turning.

The submode Wait_for_turning is exited when the turned message arrives. Depending on the current state of the four slots, messages are sent out to the respective station. This is managed by the four conditional connectors and the guards Case9, . . . , Case16. For example, the load message is only sent if the first slot is empty, and the drill message is only send if the second slot is occupied. The local variables signal_loader etc. are used to keep track of what messages were sent. When all messages are sent, the history variable System[Controller] is assigned the value Wait_for_stations. Thus, the *Controller* will continue executing in that submode when reentered.

In submode Wait_for_stations, the *Controller* waits until it receives a reply to one of the messages sent. Since this is a non-lazy mode, it must be exited as soon as the guard on one of the outgoing discrete actions (Case1, . . . , Case7) is satisfied. If the message carry a value (which is the case for loaded and tested), it is used to update the state of the corresponding slot. When all messages have been received, the message turn is sent to the *Turnable*, and the history variable is set to Wait_for_turning before exiting, meaning that the execution will be resumed in that submode.

5.3 Analysis

We have analyzed the model of the turntable system, transformed into a network of PTA models, in UPPAAL

Table 2. System properties and verification results.

Num	System property	Temporal logic formula	Verification result
1	The system should be free from deadlocks.	A[] not deadlock	Satisfied
2	A product must be clamped when drilled.	A[] Driller.Driller_moving_down imply Driller.drill_ clamp==locked	Satisfied
3	The table should never turn when one of the stations is operating.	A[] (Turntable.Turn1 or Turntable.Turn2) imply (Loader.Idle and Unloader.Idle and Tester.Idle and Driller.Idle)	Satisfied
4	Processing five products should never take more than 25 seconds (assuming at most one failed drilling).	A[] (not loaded_failed and time>25 and failed_products≤1) imply processed_products≥5	Satisfied
5	What is the minimum energy consumption for processing five products?	E⟨⟩ (processed_products==5)	14 300 units

CORA[1]. Currently, the semantic translation from REMES to PTA is done manually, as described in [18], although ideally this step should be fully automated.

After having provided UPPAAL CORA with the PTA model of the turntable system, the last step to verify the system design is to formulate the desired system requirements as temporal logic formulas. Table 2 lists the system requirements given in Section 2 together with their temporal logic formulas and verification results. Property 1 is a generic safety property, specifying the absence of a system deadlock, i.e., the system cannot come to a state from which it cannot continue operating. The turntable system is verified to be deadlock free. The next step is to verify that it satisfies the functional system requirements, here represented by properties 2 and 3. Properties 4 and 5 are examples of extra-functional properties, addressing time and resource usage, respectively.

5.4 Packaging as components

The activities illustrated in the above sections concern various functional and extra-functional properties of the components, such as architectural models, REMES behavioral models, PTA models, UPPAAL verification queries and results, among others. To promote their reuse together with the entity they describe, they are all packaged together in a "bundle" of development artifacts, which together constitute a component in the ProCom sense. This packaging is managed by the attribute framework, which provides a common structure to attributes of different kinds, such as metadata specifying the date when an attribute value was entered or edited.

In the integrated development environment providing support for development with ProCom, called *PrIDE*, the attribute framework is also responsible for registering editors by which complex attributes such as REMES models can be viewed and modified. An illustration of the use of the REMES attribute for the ProSys Driller subsystem is presented in Figure 8. Additional attributes could also be specified and used in a similar way, for example attributes corresponding to the timed automata or priced timed automata models.

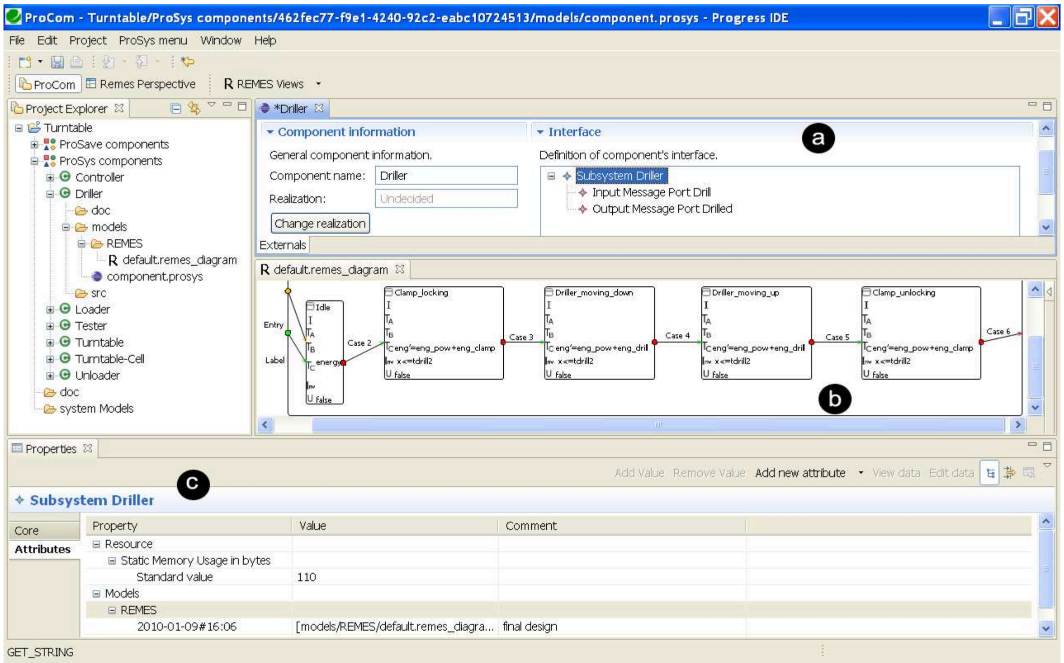
From the rich REMES model of a component — expressing how e.g., the resource usage changes over time or in response to arriving messages, or how consumption of different resources are related — it is possible to extract isolated extra-functional properties that can be stored as separate attributes. For example, from the model of *Driller* we can derive a bound on the additional power consumed as the result of a received Drill message (the bound is $\mathsf{tdrill2} * \max(\mathsf{eng\_clamp}, \mathsf{eng\_drill})$). Albeit very simple compared to the full REMES model, a MaxEnergy attribute attached to the Drill input port provides useful information about the component and could serve as input to other analysis techniques.

Ideally, any analysis result from a component analyzed in isolation should be stored as an attribute of that component. In the turntable case, the second property in Table 2 holds for the *Driller* subsystem regardless of how the rest of the system behaves. The property could be packaged as a reusable attribute of the *Driller* subsystem. However, the details of how such attribute should be specified are yet to be elaborated.

6 Related work

Few component models incorporate component extra-functional behavioral aspects (e.g. timing, resource usage, etc.) in their frameworks. This is especially true for widespread "commercial" component models such as EJB [15],

[1] See the web page www.cs.aau.dk/~behrmann/cora/ for more information about the UPPAAL CORA tool.



General component information.

Component name: Driller

Realization: Undecided

Definition of component's interface.

- Subsystem Driller
  - Input Message Port Drill
  - Output Message Port Drilled

Clamp_locking: I, T_A, T_B, T_C eng'=eng_pow+eng_clamp, Inv x<=tdrill2, U false

Driller_moving_down: I, T_A, T_B, T_C eng'=eng_pow+eng_dril, Inv x<=tdrill2, U false

Driller_moving_up: I, T_A, T_B, T_C eng'=eng_pow+eng_dril, Inv x<=tdrill2, U false

Clamp_unlocking: I, T_A, T_B, T_C eng'=eng_pow+eng_clamp, Inv x<=tdrill2, U false

Subsystem Driller

Property	Value	Comment
⊟ Resource		
⊟ Static Memory Usage in bytes		
Standard value	110	
⊟ Models		
⊟ REMES		
2010-01-09#16:06	[models/REMES/default.remes_diagra...	final design

Figure 8. Screenshot from PrIDE, with (a) the ProSys editor, (b) REMES editor, and (c) attribute framework.

COM [17], which provide little or often no support for modeling and analysis of extra-functional properties of components. In contrast, there is a substantial recognition from the research-oriented component models, such as BIP [4], Palladio [5], PECT [12], which generally support a predefined subset of extra-functional properties. For instance, Palladio checks performance prediction of timing properties (response time, throughput), or resource usage properties. Koala [21] considers static memory usage only. BIP focuses on timing properties such as worst-case execution time, or end-to-end delay.

Two ways of integrating behavioral models can typically be found in the research community: either the behavioral model is an intrinsic part of the component, as in BIP, or it is placed along the component, or the system, as in Palladio, for instance. Our approach positions itself in the middle: the behavioral model is placed alongside the components, but it also is an intrinsic part of the component specification, via the attribute framework. As different from BIP, our approach allows one to attach behavioral models not only to components, but also to individual services, for example. In comparison to Palladio, which is mainly checked by simulation, we use formal behavioral models that allow formal verification of behavior, in addition to simulation, hence increasing the level of trust in the functional and extra-functional behavior of components and systems. Moreover, we also facilitate model reuse, since the REMES behavioral models are part of the structure that constitutes a component. An alternative solution consists in using analytical interfaces jointly with a reasoning framework to perform property predictions such as in BlueArX [13] and PECT [12].

There is also a growing interest for applying model-driven development in early design and analysis of embedded systems, due to automated environments, such as MathLab Simulink-Stateflow [14], and the development of the UML profile for Modeling and Analysis of Real-time and Embedded systems, (MARTE) [16]. In contrast to a component-based approach, this methodology is not centered around the notion of components, and does not focus on reusability; instead, it considers the model (or a set of models comprising architectural, environment, resources, allocation models etc.) as the only development artifact.

7 Conclusions and future work

We have presented a component based approach for modeling both the architecture and behavior of distributed embedded systems. The architectural aspect is modeled according to the component model ProCom, to simplify the design process and facilitate design-time reuse. The behavior of individual components is modeled in REMES, in which functionality, timing and resource usage can be addressed together. Transformations of REMES models into timed automata or priced timed automata allow for model-checking of various properties, performed locally or at system level. By connecting REMES behavioral models to individual ProSys components, via a general attribute framework, we have addressed the important problem of model reuse. The applicability of the approach has been shown by employing our framework on modeling and analysis of a turntable drilling system.

Future work includes exercising the scalability of REMES and associated analysis techniques. Instead of generating a timed automata model of the entire system, compositional reasoning could be used to prove global system properties out of individual subsystems, or subsystem clusters properties. Another approach will envision developing specialized model checking optimizations, which exploit the topology of the ProSys architecture, similar to the work on UPPAAL PORT [10]. Moreover, the relation between REMES models and other, simpler, attributes should be investigated further, as well as the relation between the REMES model of a composite component and those associated with its subcomponents. The overall approach should also be further validated by case studies involving real industrial systems.

Acknowledgement

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

References

[1] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proceedings of the IEEE*, 8(3):231–274, 1987.

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

[3] R. Alur, S. La Torre, and G. J. Pappas. Optimal paths in weighted timed automata. *Theor. Comput. Sci.*, 318(3):297–322, 2004.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proc. of the 4th IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE, 2006.

[5] S. Becker, H. Koziolek, and R. Reussner. Model-Based Performance Prediction with the Palladio Component Model. *the 6th international workshop on Software and performance*, 2007.

[6] E. M. Bortnik, N. Trčka, A. Wijs, B. Luttik, J. M. van de Mortel-Fronczak, J. C. M. Baeten, W. 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(3):185–198, 2001.

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

[9] I. Crnković and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[10] J. Håkansson, J. Carlson, A. Monot, P. Pettersson, and D. 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.

[11] T. A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design. *Computer*, 40(10):32–40, 2007.

[12] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Packaging Predictable Assembly with Prediction-Enabled Component Technology. Technical Report: CMU/SEI-2001-TR-024, 2001.

[13] J. E. Kim, O. Rogalla, S. Kramer, and A. Haman. Extracting, Specifying and Predicting Software System Properties in Component Based Real-Time Embedded Software Development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[14] MathWorks. Simulink. `www.mathworks.com/products/simulink/`, accessed February 2010.

[15] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly and Associates, 2001.

[16] Object Management Group. A UML Profile for MARTE, Beta 1, August 2007. Document number: ptc/07-08-04.

[17] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

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

[19] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković. Integration of Extra-Functional Properties in Component Models. In I. P. Christine Hofmeister, Grace A. Lewis, editor, *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*. Springer Berlin, LNCS 5582, June 2009.

[20] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In M. R. Chaudron and C. Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.

[21] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.