Towards Simulative Environment for Early Development of Component-Based Embedded Systems

Marin Orlić[1] marin.orlic@fer.hr
Aneta Vulgarakis[2] aneta.vulgarakis@mdh.se
Mario Žagar[1] mario.zagar@fer.hr

[1] *Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia*
[2] *Mälardalen Real-Time Research Centre, Mälardalen University, Sweden*

Abstract

As embedded systems become more and more complex the significance of predictability grows. The particular predictability requirements of embedded systems, call for a development framework equipped with tools and techniques that will guide the design and selection of system software. Simulation and verification are two complementary techniques that play a valuable role in achieving software predictability already at early design stage. Simulation is scalable and can be very useful in debugging and validating the system design. Moreover, it can be used as a supplement to verification for visualizing diagnostic traces produced by the verification tool and for rerunning counterexamples in cases when the verification property is not satisfied.

In this paper we introduce an idea of a simulative environment for early development of component-based embedded systems. By using it, the designer can navigate and debug the design and behavior of such systems at early stages of the system lifecycle.

1 Introduction

As the complexity of embedded systems grows their development becomes more and more difficult. An appealing approach to manage the embedded systems software complexity, reduce time-to-market and decrease development costs lies in the adoption of component-based development [10]. The specific predictability demands of embedded systems, require the designer to employ a framework equipped with tools and techniques that can be applied to deal with requirements such as dependability, timing, and resource utilization, already at early-stage of development. Modeling, simulation and verification play increasingly important roles in achieving predictability, since they can help us to understand how systems function, validate the design and verify some important properties.

Simulation validates the behavior of a system for one execution path. Being relatively inexpensive in terms of execution time compared to verification, simulation is a valuable fault detection technique in early stages of system development. In general, it can be used to quickly verify a system prototype for desired properties and behavior and it can contribute to our studying of system design alternatives, in a controlled environment. Moreover, with simulation one can explore system configurations that are difficult to physically construct, and observe interactions that are difficult to capture in a live system. The ability of the simulation can be applied as a complementary activity to verification, which covers the exhaustive dynamic behavior of the system. A simulator can be used for visualizing diagnostic traces generated by the verification tool and for replaying counterexamples in cases when the verification property does not hold.

In this paper we introduce a simulative environment for development of component-based embedded systems. The simulative environment allows the designer to navigate the behavior of possibly complex and multilayered systems with respect to time and resource consumption and check behavior compliance to resource constraints. Here, we use the ProCom component model for describing the architecture of our embedded systems [9]. Additionally, we use the Remes dense-time state-based language [18] for modeling resource-wise behavior of ProCom components. Our main goal for the simulator is to be developer-friendly and usable by system modelers, engineers and developers with no prior knowledge of formal verification methodologies and tools. Finally, our intent is to present this environment to the user as a debugger with a familiar interface that will reduce the user learning effort.

The remainder of the paper is organized as follows. Section 2 reviews the ProCom component model and the associated behavioral model Remes needed to comprehend the rest of the work. Section 3 introduces our simulative environment and finally, Section 4 discusses our and related approaches and concludes the paper.

2 Preliminaries

2.1 The ProCom component model

The ProCom component model [21] is designed to address the key requirements and modeling issues coming from the embedded system domain. In particular, ProCom considers the need for the design of a complete system consisting of both complex and distributed functionalities on one hand, and small low-level control-based functionalities on the other. Therefore, ProCom is a hierarchical component model structured into two layers: ProSys and ProSave. The upper layer, ProSys, serves for modeling a system as a collection of active and distributive *subsystems* that execute concurrently, and communicate by asynchronous messages sent and received at typed output and input *message ports*. The lower layer, ProSave, models the internal design of subsystems as interconnected passive components with small functionality, whose communication is based on the pipe-and-filter paradigm with an explicit separation between data- and control flow. The former is represented by *data ports*, and the latter by *trigger ports*. The functionality of a ProSave component is captured by a set of *services*, which may execute concurrently while sharing only data, but no triggering. Components may be interconnected by simple connections from output- to input ports or by *connectors* that provide detailed control over data- and control flow. A ProSave component can be activated by a special type of construct, *clock*.

The ProSys and ProSave layer can be related to each other only in the lowest level of a ProSys hierarchy, where a ProSys component can be modeled out of ProSave components. For more details, see [9].

2.2 The REMES behavioral modeling language

The REsource Model for Embedded Systems REMES [18] is a dense time state-based behavioral modeling language, which is primarily intended to provide a basis for capturing resource-constrained and timing behavior of embedded systems. It introduces resources as first-class modeling entities that are characterized by their discrete (e.g., memory, access to external devices) or continuous (like energy) nature.

For formal analysis purposes, REMES models can be transformed into timed automata (TA) [2], or priced timed automata (PTA) [1], depending on the analysis type. We use REMES for modeling and (when translated to TA or PTA) for formally analyzing, the behavior of ProCom component-based systems.

The internal behavior of an embedded component is described by a REMES *mode* that can be either *atomic* (does not contain submodes), or *composite* (contains submode(s)). The discrete control of a mode is captured by a *control interface* made up of *entry*- and *exit* points, whereas the data transfer between modes is done through a *data interface*. Similar to other languages, each REMES mode may contain *local* or *global* variables that can be of types integer, natural, boolean, array, or clock.

Assuming that a component consumes resources, its REMES mode can be annotated with the corresponding resource-wise continuous behavior. The consumption is expressed by the first derivatives of the variables that denote resources, and which evolve at positive integer rates.

The control flow is given by *edges* (i.e., a set of directed lines) that connect the control points of (sub)modes. The continuous behavior of a mode is captured by *delay/timed* actions and their execution does not change the current mode. The discrete behavior is given by discrete actions (represented as edge annotations), which execution changes the mode. A discrete action can be executed only when the corresponding boolean *guard* that prefixes the action body holds. A REMES composite mode may contain *conditional connectors* that enable nondeterministic selection of one discrete outgoing action to be executed, out of many possible ones. A mode may also be annotated with *invariants* that bound from above the current mode's execution time. For more details about the REMES model, we refer the reader to [18].

3 An idea for a simulative environment

Starting from a given specification of a system (architecture- , behavior- and platform specification) we want to simulate the system behavior, with respect to timing and resource utilization. In order to achieve this goal we propose to build a simulator that would be able to accept a model fed in by a developer and allow the user to track the changes in component behavior, component activation and resource utilization. Ideally the user interface should be provided in a fashion that the user is comfortable with, in order to avoid the resistance associated with "learning yet another tool".

3.1 The 2+1 view of a system

To prepare a specification of an embedded system, we propose a three-fold view of the system. Architecture- and behavior specification define the desired system, and the third component - the platform specification, describes the execution platform for the system. The first two models are

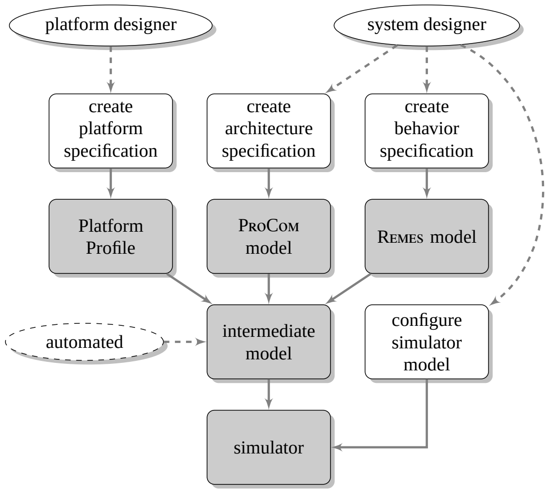
platform designer
system designer
create platform specification
create architecture specification
create behavior specification
Platform Profile
ProCom model
Remes model
automated
intermediate model
configure simulator model
simulator

Figure 1. Workflow steps involved in setting up the simulator

typically specified by the system designer, while the last one comes from the platform designer and is a common artefact shared between all systems or products based on the same platform.

Architecture specification comprises of systems, components and their connections conforming to the ProCom component model, and the behavior specification of the system is specified with Remes models, where each Remes model is corresponding to a ProCom component.

Platform specification declares available platform resources – CPU, available memory, energy consumption etc., within a platform profile. Once declared, resource consumption is modeled within component behavior – the resources are referenced as variables that cannot be read, only incremented and decremented. Platform profile also specifies constraints over resources. We propose to define constraints as minimum, maximum and average functions on either concrete resource values or resource changes (differences), as defined by the following grammar:

$$\begin{array}{lll} rc & ::= & (\ max \mid min \mid avg\ ) \\ & & \text{'('} \quad (resource \mid resource') \quad \text{')'} \\ & & (\, < \mid \leq \mid = \mid \geq \mid > \,) \quad value \end{array}$$

For example, the platform profile can define the constraints for CPU and memory resources such as: $\max(\mathsf{CPU}) < 200, \max(\mathsf{mem}) < 16384$, to define memory size to 16 Ki units and CPU usage to 200 units (or 200% usage, assuming two available CPU cores). In case of available energy the constraints could be: $\max(\mathsf{eng}') < 50, \max(\mathsf{eng}) < 15000$, to limit usage peaks to 50 units, with maximum total energy reserve of 15000 units. The choice of operators max, min and avg allows tracking and detecting peaks and spikes, as well as average resource usage.

To allow some degree of behavior parametrization, the platform profile can also define values for constants declared in Remes models. If a Remes model declares constants with no values assigned, it is assumed that such constants will finally be assigned values when a profile is added. This allows component behavior to use platform-dependant constants to declare resource usage, e.g. component initialization overhead.

During development of a system in compliance to a specific platform profile, the profile can ideally be replaced with another. Applying a new profile allows to check conformance with constraints of a different platform configuration, or a different platform version.

3.2 Generating the intermediate model

In order to prepare the simulation, the architecture- and behavior specifications are combined to form an integral intermediate model of the system. The purpose of the intermediate model is similar to that of object files obtained by compiling the source code of a programming language – it contains syntax-checked model information and resolved variable references. As a part of this process, expressions contained in component behaviors are translated to their corresponding abstract syntax trees and type-checked. Intermediate model joins the architecture and behavior using predefined mappings between components and behavior. Architecture and behavior are both copied to a single model namespace with the addition of a platform profile, forming a simulation specification. For example, architecture-behavior mappings map Remes variables used in Remes behavior models to input and output data ports of a ProSave/ProSys component. Connections between such

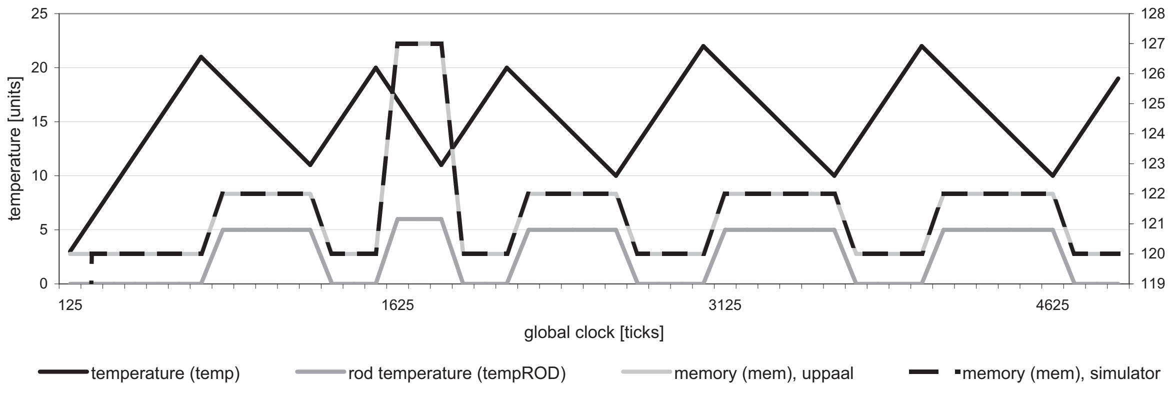

Figure 2. An example run of a simulator for Temperature Control System

data ports are converted to variable renamings (mappings between Remes variables in different behavior models) in the intermediate model.

The intermediate model is the input model for the simulator, therefore it should be complete and well-defined – references to unknown variables or type-invalid expressions would make simulation impossible.

The process of generating the intermediate model should be hidden from the user. Whenever an architecture- or behavior specification for a component changes, it's corresponding intermediate model should be automatically generated, simplified and checked. Figure 1 gives an overview of the model translations needed to prepare for the simulation. Actors are represented with ellipses, processes with white boxes, and artifacts with gray boxes. Platform- and system specification are essentially separate design processes. Platform designer specifies platform resources and constraints through a platform profile. System designer (possibly a team of engineers) is responsible for architecture and behavior of the system and creates ProCom and Remes models, respectively, to specify the two. An automated process transforms the three models (platform profile, architecture- and behavior models) to an intermediate model. Finally, system designer configures the simulation process with a simulator model.

3.3 Code generation and simulation

To simulate the system, we have utilized code generation from the intermediate model, using common model-to-text transformation tools. Compared to interpreter, generated code is simpler – the structure of the system is mirrored in generated code, and the simulator core can manipulate program objects directly, using language facilities, instead of manipulating model elements using model API or reflection. The core simulator process was designed after the time and action successor functions for timed automata [16]. As mentioned before, an alternative to this approach would be to perform the simulation using interpretation with a model visitor.

The simulator is configured with its corresponding simulator model. The simulator model contains links to intermediate model of a system, platform profile and, optionally, one or more simulator sensors. Sensors monitor data points (Remes variables or component data ports) and record value changes when triggered. Sensors can be triggered on Remes variable change or on component trigger port activation. Data collected from sensors is displayed in the simulator environment and stored for later analysis.

To show an example of a simulator run, we can look at a temperature control system (TCS) [18]. TCS models a cooling controller for a reactor system that has two cooling rods which are used to absorb excessive reactor heat thus maintaining reactor temperature within predefined boundaries. The primary purpose of the Remes behavior model of the TCS system is to illustrate resource consumption (e.g., CPU, memory and energy) during TCS system lifetime.

Figure 2 illustrates changes in core and rod temperatures and memory consumption for a sample run in both our simulator and the one in the Uppaal [*] tool. Both simulators were forced to follow the same execution trace when selecting transitions to perform. Slight differences in the results can be noted for memory consumption at the very beginning of the simulation. This is due to different resource initialization strategies – TCS model in Uppaal performs resource initialization at the time the components (Uppaal processes) are activated, while our simulator adheres to the Remes execution model and performs initialization the first time a component is activated and its corresponding behavior mode is entered.

The main benefit of simulating the TCS system is the ability of the simulator to track changes for each resource

[*]For more information on Uppaal , please visit `http://uppaal.com/`

Table 1. Mapping between common debugger objects and ProCom/Remes objects

Debugger object	ProCom/Remes object	Comment
Process	System	Container of execution for all objects
Thread	Active subsystem (ProSys)	Basic unit of parallel execution
Stack frame (method)	Component (in a hierarchy of components)	Unit of (hierarchical) sequential execution
Current instruction pointer	Active mode of component behavior	Smallest unit of execution
Variable	Mode variables	Variables and resources

separately. The current implementation of Uppaal Cora [†] is somewhat limited – model checking or simulation can be performed over a single monotonically rising cost variable, with occasional errors in the simulator. To track resource changes on Figure 2 we have manually tracked memory resource change using Uppaal and its simulator. Note that in Uppaal Cora all resources need to be combined to a single cost variable. This approach does not allow to track each resource separately. Therefore, we have used the Uppaal simulator to track memory changes for comparison with our simulator. The downside of this approach is that only discrete model transitions can update resources, as Uppaal cannot model continuous variable change. However, in the sample TCS system memory resource consumption is not affected by delay transitions but only discrete transitions of the automata.

3.4 Simulative environment from a user's perspective

Our main goal for the user interface is to reuse the existing UI as much as possible, and reduce the effort needed to use the simulator facilities. With this in mind, we propose to integrate the simulator with a well-known IDE platform, similar to what was done with SaveIDE [19] (Eclipse-based) and Uppaal Port [22], but reuse the platform even further and present the simulator as a debugger.

Figure 3 illustrates the user's perspective. Architecture and behavior models are created using graphical editors, as seen on the left. These models are then automatically translated into their intermediate model counterparts (in the middle). Platform profile (top right) is linked with the two using a simulator configuration model (middle right) which is used to generate the simulator classes (bottom right). The intermediate model consists of several submodels, as both architecture and behavior models can be split over several submodels, e.g. for each component in the system.

Users accustomed to modern IDEs are also familiar with the concept of debuggers – every programming language comes with one, and users are familiar with core debugging concepts. Debuggers deal with objects that model execution elements like processes, threads, stack frames, current instruction pointers and variables, and features of modern IDEs are built to support manipulation of these objects. In a system modeled by ProCom and Remes we can distinguish active elements such as subsystems, components and modes of behavior that have some similarity to traditional execution elements. An example of relation between some common debugger objects and ProCom/Remes objects is given in Table 1. During simulation, we can manipulate these objects in the same fashion as during debugging of a program process – pause execution, switch between active elements, inspect current state and so on.

In this way we can reuse metaphors of threads, stack frames and current instruction pointers to follow the active model elements (subsystems, components and modes). Mode variables correspond to debug variables. This allows the user to navigate possibly complex and multilayered system through both its architecture and behavior in a familiar fashion – as debugging equally complex structures defined in traditional programming languages. It is our hope that this approach will increase the appeal of the simulative environment to a wider audience.

During debugging, the user needs to be able to navigate the system model(s). To enable this, the simulative environment needs to be integrated with the development environment for ProCom – the Progress-IDE [13, 20]. Progress-IDE is built on Eclipse Platform [12] which provides rich editors and a Debug platform among other facilities. The environment is component-centric, and system and component structure are modeled in ProCom. Work on support for behavior modeling with Remes, simulator-debugger interface and automating tasks required to generate intermediate models is in progress.

4 Discussion

4.1 Assumptions

There are several assumptions (or limitations) built into the simulation process, which we list in the following.
Static architecture specification – the architecture specification is implied to be static. Although ProCom component model doesn't explicitly prohibit dynamic reconfiguration of components and their connections, the simulation

[†]For details, visit `http://www.cs.aau.dk/~behrmann/cora/`

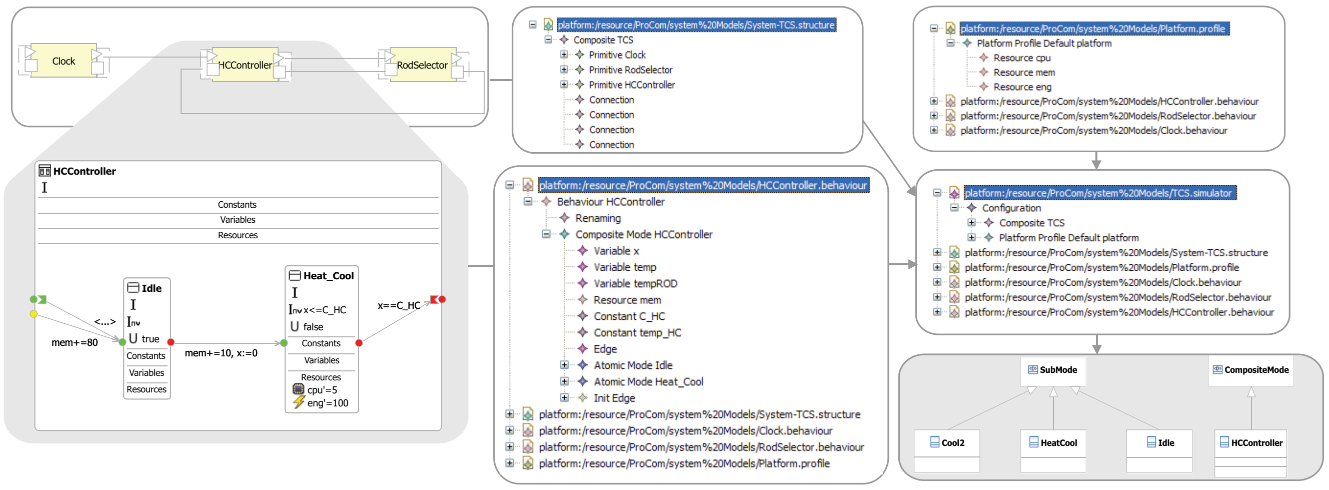
Clock
HCController
RodSelector
HCController
Constants
Variables
Resources
Idle
Inv
U true
Constants
Variables
Resources
<...>
mem+=80
mem+=10, x:=0
Heat_Cool
Inv x<=C_HC
U false
Constants
Variables
Resources
cpu'=5
eng'=100
x==C_HC
platform:/resource/ProCom/system%20Models/System-TCS.structure
Composite TCS
Primitive Clock
Primitive RodSelector
Primitive HCController
Connection
Connection
Connection
Connection
platform:/resource/ProCom/system%20Models/Platform.profile
Platform Profile Default platform
Resource cpu
Resource mem
Resource eng
platform:/resource/ProCom/system%20Models/HCController.behaviour
platform:/resource/ProCom/system%20Models/RodSelector.behaviour
platform:/resource/ProCom/system%20Models/Clock.behaviour
platform:/resource/ProCom/system%20Models/HCController.behaviour
Behaviour HCController
Renaming
Composite Mode HCController
Variable x
Variable temp
Variable tempROD
Resource mem
Constant C_HC
Constant temp_HC
Edge
Atomic Mode Idle
Atomic Mode Heat_Cool
Init Edge
platform:/resource/ProCom/system%20Models/System-TCS.structure
platform:/resource/ProCom/system%20Models/Clock.behaviour
platform:/resource/ProCom/system%20Models/RodSelector.behaviour
platform:/resource/ProCom/system%20Models/Platform.profile
platform:/resource/ProCom/system%20Models/TCS.simulator
Configuration
Composite TCS
Platform Profile Default platform
platform:/resource/ProCom/system%20Models/System-TCS.structure
platform:/resource/ProCom/system%20Models/Platform.profile
platform:/resource/ProCom/system%20Models/Clock.behaviour
platform:/resource/ProCom/system%20Models/RodSelector.behaviour
platform:/resource/ProCom/system%20Models/HCController.behaviour
SubMode
CompositeMode
Cool2
HeatCool
Idle
HCController

Figure 3. From architecture- and behavior specification to simulator code: architecture- and behavior models (left) are translated into intermediate models (middle) and combined with platform profile to form the simulator configuration (right), which is then used to generate simulator source code (bottom right).

assumes that components cannot migrate to different underlying hardware resources or change connections.

Simplified view of system runtime environment – processing elements such as CPUs are described by their processing speed/rate in a simple resource model. This is a simplification as modern CPUs cannot be characterized with just their clock frequency when many parameters such as processor architecture, cores, cache, pipelining, instruction dispatch and general platform architecture come into play. To some extent this assumption originates in REMES, with an intent that behavior models stay platform-independent.

Limited support for concurrency – execution parallelism in target hardware platform and concrete component allocation to hardware nodes is not taken into consideration. It is instead left for analysis in later system design stages when deployment/allocation models are introduced.

4.2 Simulation strategy

In section 3.1 we introduced platform resource usage as incrementing and decrementing referenced resources (that behave as scalar variables). The simulator actually manipulates resources as intervals with open or closed bounds (endpoints). When a discrete transition and its corresponding action is performed, it can increment or decrement the resource variable – in effect, translating the resource interval by a specified amount. When a delay transition is performed, the resource update is calculated depending on the duration of the transition – in effect, arbitrarily changing the resource interval bounds. Resource updates therefore depend on the timed execution of model, as described in [18].

Simulation is performed in steps guided by minimum time intervals for next discrete transition, similar to global execution strategy described in [3]. In essence, the list of active modes and possible transitions is traversed to calculate time intervals till next discrete transition. From the list of intervals, a minimum interval is selected, time is let pass within this interval and the system state is updated accordingly. Transition prioritization and selection (perhaps on user intervention) can easily be performed during mode list traversal in each round.

4.3 Trace visualization

Simulator can be used to visualize traces generated by the verification tools and inspect counterexample states in detail. Our approach of presenting the simulator as a debugger can easily be adapted to this purpose – the process of transition selection for the next simulation round should be guided by the generated diagnostic trace, instead of usual selection rules. When following a trace, the designer can monitor state change, and in any moment divert from the generated trace to investigate a different dynamic execution path. In combination with model-checkers for verification, a proposed tool could be used for both quick prototyping at an early system design stage, and system verification of a complete system model.

4.4 Partially-specified systems

An interesting topic for further consideration is the possibility to simulate and analyze partially-specified systems. To illustrate, imagine a system designer working on an early system design. She has specified overall structure, but has yet to define behavior specifics of each component. However, when designing for a concrete platform, some details (e.g. component overhead resource consumption) are already known as they are dictated by the platform. With this in mind, it should be possible to simulate the early system model based on *default* component behaviors provided within platform profile. We intend to extend the platform profile with the description of default behaviors for components, the support for this in REMES remains a topic for discussion.

4.5 Related approaches

Several approaches, based on simulation models derived from UML diagrams, have beed suggested. Extended UML can be used to specify system models directly, and de Miguel et al. [11] propose extensions (with UML profiles) to express temporal requirements and resource usage. Annotated diagrams are then automatically transformed to scheduling and simulation models using Analysis- and Simulation Model Generators, respectively. Similar to our approach, application element models are transformed to simulation submodels which are combined to form an integrated simulation model. A second approach, proposed by Arief and Spiers [6] uses UML to specify system details needed for simulation with a process-oriented simulation model. System simulation is built using a predefined Java-based Simulation Modeling Language (SimML) framework with key elements such as components, processes, queues and messages.

Balsamo and Marzolla in [7, 15] propose a similar tool for simulation of performance for process-oriented systems. Annotated UML diagrams, such as Use Case, Activity and Deployment diagrams, are used to describe system performance parameters. UML model elements are closely related those of the simulator, and simulator structure and behavior closely follow the structure and behavior of the UML model. A discrete-event simulation model is automatically extracted from the diagrams, and simulation results are reported back as tagged values in diagrams.

A notable approach is that of Palladio Component Model (PCM) [8, 17]. PCM describes component-based systems with structure, behavior, allocation and usage models and derives a simulation model from them. PCM can model resource demands of discrete component actions and provide statistical results, such as processing rate, throughput and response time per component. Simulation workload is generated using domain-specific experts' knowledge contained in the usage model. A development and analysis environment is provided.

When discussing embedded systems, we should not forget to consider approaches using Matlab and Simulink, as these tools have established themselves as standard tools for embedded system design and analysis. COMDES [4, 5, 14] is a framework for hard real-time distributed control systems that uses actor diagrams representing subsystems and signals exchanged between them, and state-machines or functional block diagrams to specify behavior. COMDES translates the model to Simulink for simulation.

Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, Croatian Ministry of science, education and sports via the research project SOFTWARE ENGINEERING IN UBIQUITOUS COMPUTING, and the Unity Through Knowledge Fund via the DICES project.

References

[1] R. Alur. Optimal paths in weighted timed automata. In *In HSCC'01: Hybrid Systems: Computation and Control*, pages 49–62. Springer, 2001.

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

[3] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular Specification of Hybrid Systems in Charon. *Hybrid Systems: Computation and Control, Third International Workshop*, LNCS 1790:6–19, 2000.

[4] C. Angelov, K. Sierszecki, and N. Marian. *Component-Based Design of Embedded Software: an Analysis of Design Issues*, volume 3409, pages 1–11. Springer Berlin / Heidelberg, 2005.

[5] C. Angelov, K. Sierszecki, N. Marian, and J. Ma. *A Formal Component Framework for Distributed Embedded Systems*, pages 206–221. Springer Berlin / Heidelberg, 2006.

[6] L. B. Arief and N. A. Speirs. *A UML tool for an automatic generation of simulation programs*, volume 21. ACM Press, New York, New York, USA, 2000.

[7] S. Balsamo and M. Marzolla. A simulation-based approach to software performance modeling. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 363–366. ACM, 2003.

[8] S. Becker, H. Koziolek, and R. Reussner. Model-Based Performance Prediction with the Palladio Component Model. In *Proceedings of the 6th international workshop on Software and performance*, page 65. ACM, 2007.

[9] 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.

[10] I. Crnkovic. Component-based Software Engineering for Embedded Systems. pages 71–90. IESTE, Ltd, 2006.

[11] M. de Miguel, T. Lambolais, M. Hannouz, S. Betgé-Brezetz, and S. Piekarec. UML extensions for the specification and evaluation of latency constraints in architectural models. In *Proceedings of the second international workshop on Software and performance - WOSP '00*, pages 83–88, New York, New York, USA, 2000. ACM Press.

[12] Eclipse. `http://www.eclipse.org/`.

[13] J. Feljan, L. Lednicki, A. Petričić, and S. Sentilles. Requirements on the system design phase for Progress-IDE, DICES technical report. `http://www.fer.hr/dices/resources` accessed 14/4/2010.

[14] N. Marian and Y. Ma. Translation of Simulink Models to Component-based Software Models. In *Proc. of the 8th International Workshop on Research and Education in Mechatronics REM'2007*, pages 262–267, Talinn, Estonia, 2007.

[15] M. Marzolla and S. Balsamo. UML-PSI: the UML performance simulator. *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pages 340–341, 2004.

[16] W. Penczek and A. Półrola. *Advances in verification of time petri nets and timed automata: a temporal logic approach*. Springer-Verlag New York Inc, 2006.

[17] R. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, and M. Kuperberg. The Palladio component model, 2007.

[18] 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.

[19] S. Sentilles, J. Håkansson, P. Pettersson, and I. Crnković. 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*, 2008.

[20] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic. 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.

[21] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic. A component model for control-intensive distributed embedded systems. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.

[22] Uppsala University, Aalborg University. UPPAAL Port. `http://www.uppaal.org/port/` accessed 14/4/2010.