Formal Semantics of the ProCom Real-Time Component Model

Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu and Paul Pettersson
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{*aneta.vulgarakis,jagadish.suryadevara,jan.carlson,cristina.seceleanu,paul.pettersson*}*@mdh.se*

***Abstract*—ProCom is a new component model for real-time and embedded systems, targeting the domains of vehicular and telecommunication systems. In this paper, we describe how the architectural elements of the ProCom component model have been given a formal semantics. The semantics is given in a small but powerful finite state machine formalism, with notions of urgency, timing, and priorities. By defining the semantics in this way, we (i) provide a rigorous and compact description of the modeling elements of ProCom, (ii) set the ground for formal analysis using other formalisms, and (iii) provide an intuitive and useful description for both practitioners and researchers. To illustrate the approach, we exemplify with a number of particularly interesting cases, ranging from ports and services to components and component hierarchies.**

***Keywords*-real-time systems; embedded systems; component model; finite state machines; timed automata**

I. Introduction

Designing embedded systems (ES) in a *component-based* fashion has become an attractive approach for embedded software development. With benefits ranging from simplification and parallel working to pluggable maintenance and reuse, the financial gains are significant. In this context, systems consist of identifiable, relatively independent and generally replaceable units of composition, called *components*, which encapsulate complex functionality.

Once a component is defined, it can be distributed and used in other applications. Examples of component models include JavaBeans [1], Koala [2], SOFA [3], [4], ProCom [5], [6] etc. Out of these, ProCom is a recently proposed component model tailored for developing *real-time* ES in the vehicular and telecom domains.

To achieve *predictability* throughout the development of the ES, the designer needs to employ a design framework equipped with analysis methods and tools that can be applied at various levels of abstraction, in order to provide estimations and guarantees of relevant system properties. Usually, embedded system designers deal with two kinds of requirements. *Functional* requirements specify the expected services, functionality, and features, independent of the implementation. *Extra-functional* requirements specify the use of available resources. For the same functional requirements, extra-functional properties can vary depending on a large number of factors and choices, including the overall system architecture and the characteristics of the underlying platform. Consequently, ES modeling must deal with both computation and physical constraints, which calls for an underlying semantic framework that abstracts away from both physical notions of concurrency and from all physical constraints on computation.

In this paper, we formalize the semantics of ProCom [5] architectural elements, while identifying potential trouble spots in modeling, which we describe in detail in Section II-B. To tackle the mentioned modeling issues of ES, ProCom consists of two distinct, but related, layers, which expose a number of modeling characteristics that pose challenges to the system designer. The upper layer, called ProSys, serves the modeling of the ES as a number of active and concurrent subsystems, communicating by message passing. The lower layer, ProSave, addresses the internal design of a subsystem down to primitive functional components implemented by code. ProSave components are passive and the communication between them is based on a pipes-and-filters paradigm. Bridging the semantic gap between the two communication paradigms is one particular modeling challenge that we show how to solve within the proposed ProCom formalization.

Another distinguishing characteristic of ProCom is the possibility to model both fully implemented components, described internally by code, and also design-time components, possibly modeled internally as inter-connected ProSave components that might co-exist with the implemented components.

In order to rigorously describe the above mentioned and all of the other behavioral features of ProCom models, and to provide support for formal analysis, we use an underlying *finite state machine* (FSM) formalism, with notions of urgency, timing and priority. The formal semantics of the FSM language, hence of the architectural elements of our component model, is expressed in terms of *timed automata* with priorities [7] and urgent transitions [8]. However, in the following, we chose to present just some of the most interesting cases, like the formal description of services, component hierarchy, and ProSys-ProSave linking. The formalism is intended to provide a high-level, abstract representation of ProCom semantics, understandable and appealing to both formalists and engineers. Our solution is based on a small semantic core to which the synthesis of ProCom-based models of real-time embedded systems should conform. Note that, although it sets the grounds for formal verification, our semantic descriptions focus only on describing the correct behavior of ProCom architectural elements, without consideration for efficiency in formal verification of the resulted models.

The remainder of the paper is organized as follows. In Section II, we briefly recall the ProCom component model and identify some of its particularities. Section III presents

our underlying formal notation and the actual formalization of the selected ProCom architectural elements. The comparison to related work is carried out in Section IV, whereas in Section V, we conclude the paper.

II. The Component Model

A. ProCom

The ProCom component model [6] is specifically developed to address the particularities of the embedded systems domain, including resource limitations and requirements on safety and timeliness.

To achieve efficiency, ProCom components are design-time entities that can comprise information about interfaces, internal structure, code, models, attributes, etc., rather than discernable, concrete units in the final system. Applications are build as a collection of interconnected components, and in the later stages of development this component-based design is transformed into executable units, such as tasks that can be handled by traditional real-time operating systems.

Another basis of the ProCom development approach is that various types of analysis are carried out throughout the development process, in order to ensure that the application will meet requirements on resource usage, safety and timeliness. Early analysis is particularly emphasized, as it allows potential problems to be discovered when the cost of resolving them is relatively low. At early stages, analysis is mainly based on models and estimates, and in later stages on, for example, source code and concrete design parameters. A key concern is to provide means to perform analysis on systems where fully developed parts, for example reused components, co-exist with parts in an early stage of development.

To address the different concerns that exist on different levels of granularity, spanning from the overall architecture of a distributed embedded system, to the details of low-level control functionality, ProCom is organized in two distinct, but related, layers: ProSys and ProSave. In addition to the difference in granularity, the layers differ in terms of architectural style and communication paradigm.

In ProSys, the top layer, 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*.

Contrasting this, the lower lever, 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.

Figure 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.

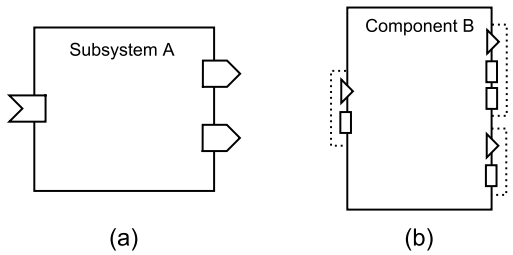

Figure 1. A ProSys subsystem and a simple ProSave component.

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.

Both layers 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. At the bottom of the hierarchy, the behavior of a primitive ProSave component is implemented as a C function.

For the purpose of analysis, it is possible to associate attributes with components and subsystems to specify different functional and non-functional characteristics. Some attributes can be represented by a single number, e.g., worst-case execution time or static memory usage, but in the case of more complex functional and extra-functional behavior (such as timing and resource consumption), a dense time state-based hierarchical modeling language called REMES [9] is used.

B. Particularities of ProCom

The ProCom component model imposes restrictions on the behavior of its constructs, which should be addressed and formally specified, in order to achieve predictable behavior. This section recalls the informal behavioral semantics of specific modeling constructs in ProCom: services, connections, component hierarchy and building active subsystems out of passive components.

The functionality of a ProSave component is captured by a set of *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. An input port group may only be accessed at the very start of each invocation, and the service may produce parts of the output at different points in time. 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 at its output port groups. The data and triggering of an output group of a service are always produced at the same time. 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. Since a

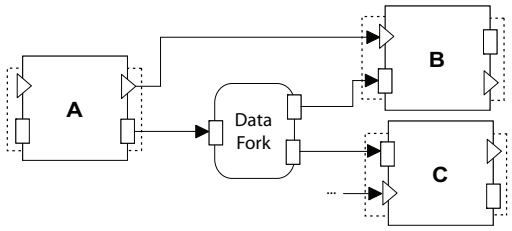

Figure 2. Example of a critical modeling of data and trigger transfer in ProCom.

service is a complex concept, its formalization is highly needed.

In the ProCom language, *connections* and *connectors* define how data and control can be transferred between ProSave components. Since ProSave components can not be distributed, the migration of data or trigger over a connection is loss-less and atomic. 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 the data passes through a connector. ProSave follows a push model for data transfer, so whenever there is data produced on an output port, it is forwarded by the connection to the input data port and stored there. In case more data (trigger) connections are enabled at the same time, the order in which they are taken is non-deterministic. Let us assume the following modeling scenario: three components A, B and C, are interconnected via a Data-Fork connector (see Figure 2). The Data-Fork connector is used to split data connections, so data written to the input data port is forwarded to the output ports. When component A has finished executing, component B should start executing. However, since the input trigger port of component B is directly connected to the output trigger port of component A, while the data is not transferred directly, but via a connector, there is a risk that the trigger signal may reach component B before the data has arrived. Hence, such a scenario in which trigger might arrive before data should be prohibited by the formalization.

Internally, a ProSave component may be described by code or other inter-connected sub-components. When a trigger of an output group is activated internally, all the data (assuming it is ready internally) and the trigger are atomically transferred to the corresponding output port groups of the enclosed component. This contributes to the fact that, externally, there is no difference between components, which allows the coexistence of fully developed components and early design units.

ProSys systems are active entities that communicate via message passing. In contrast, the communication between ProSave components is based on the pipes-and-filters paradigm. Internally, a ProSys system can be built out of other ProSys (sub)systems. At the lowest level of ProSys hierarchy, a subsystem can be internally modeled by ProSave components. In order to build active subsystems out of passive components, we use *clocks*. A clock 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. Additionally, a mapping is needed between the message passing in ProSys and the trigger/data communication used in ProSave.

Given the above, we identify the following issues that have motivated our formalism and that we show how to solve in Section III:

- The data and triggering of an output group of a service must always be produced atomically, and each of the service output port groups must have been activated exactly once before the service returns to idle state.
- All the data must arrive to its end destinations before the trigger signal. This rule should also hold in cases when data is transferred through a connector.
- Coexistence of both fully implemented components having well known inner structure, and early design black box components, should be supported.
- Bridging the two communication paradigms: message passing in ProSys and pipes-and-filters in ProSave.

III. Formal Semantics of Selected ProCom Architectural Elements

To describe the behavioral semantics of ProCom architectural elements, we introduce a high-level formalism as an extension of finite state machine (FSM) notation and semantics. Our FSM formalism is enriched with additional notions of urgency, priority and implicit timing, necessary for modeling semantics of component-based architectures of real-time systems. The formalism is small, but powerful enough to grasp all the information that is needed for proper formalization of ProCom. In addition, we believe that the language is intuitive enough to be used by developers/engineers, but also formalists/researchers. Yet this has to be proved by experiments that we leave for future work.

The FSM formalism and related graphical notation are introduced formally below.

A. Formalism and Graphical Notation

Let V be a set of variables, G a set of boolean conditions (or *guards*) over V, B the set of booleans, A a set of variable updates, and I a set of intervals of the form $[n_1, n_2]$, where $n_1 \leq n_2$ and n_1, n_2 are natural numbers. Our FSM language is a tuple $\langle S, s_0, T, D \rangle$, where S is a set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times G \times B \times B \times A \times S$ is the set of transitions between states, in which $B \times B$ represent priority and urgency (described below), and $D : S \rightarrow I$ is a partial function associating delay intervals with states.

The FSM language relies on a graphical representation that consists of the usual graphical elements, that is, states and transitions labeled with guards, priority, urgency, and updates, see first two columns of Figure 3. A transition can be either *urgent* or *non-urgent*, and it can have *priority* or no priority. As shown in Figure 3, a transition may be decorated with the non-urgency symbol *, and/or the

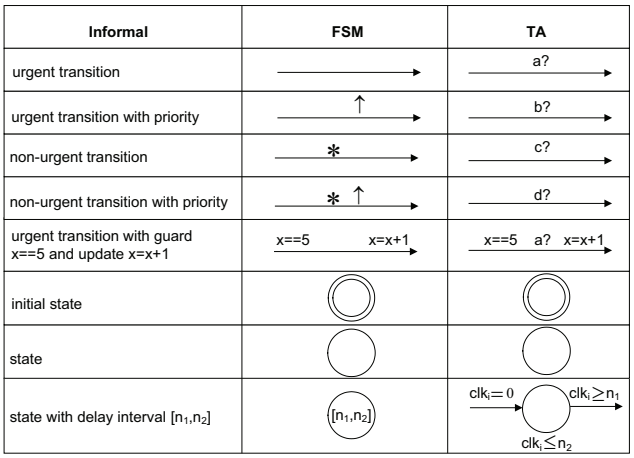

Figure 3. The graphical notation of the FSM elements and their translation into TA.

priority symbol ↑. Note that, a transition that is not annotated with * is urgent. A state can be associated with a delay interval, which is graphically located within the state circle.

Intuitively, the execution of an FSM starts in the initial state. At a given state, an outgoing transition may be taken only if it is *enabled*, i.e., its associated guard evaluates to true for the current variable values. If from the current state, more than one outgoing transition is enabled, one of them is taken non-deterministically, and prioritized transitions are preferred over non-prioritized transitions. In case all enabled outgoing transitions of a state are non-urgent, it is possible to delay in the state. On the other hand, if there are any outgoing urgent enabled transitions, one of them must be taken immediately. Thus, the notions of priority and urgency avoid unnecessary non-determinism among enabled transitions, clarifying the modeling aspects and possibly improving the performance of formal analysis. A state that is associated with a delay interval $[n_1, n_2]$ may be left anytime between n_1 and n_2 time units after it is entered.

In order to form a system, FSMs may be composed in parallel. The semantic state of the composed system is the combined states and variable values of the FSMs. The notions of urgency and priority are applied globally, and time is assumed to progress with the same rate in all FSMs.

B. Formal Semantics of the FSM Language

In this section, we formally define the semantics of our FSM language using timed automata (TA) [10] with priorities [7] and urgent transitions [8] as a semantic domain. The translation of each FSM element to TA is depicted in Figure 3. The FSM language has four kinds of transitions: urgent transition, urgent transition with priority, non-urgent transition, and non-urgent transition with priority. In TA we introduce four channels: a, b, c, and d. Channels a and b are urgent, and channels b and d have higher priority than channels a and c. Accordingly we map the transitions of FSMs into TA edges labeled with the appropriate channels, as defined in Figure 3. The translated TA edges need a timed automaton offering synchronization on the complementary channels (e.g., $a!$ complementary to $a?$), depicted in Figure 4.

Each FSM state results into a TA location. For every FSM with delay states, a clock clk_i is introduced. Accordingly, an FSM state with delay interval $[n_1, n_2]$ is translated into a corresponding TA location with invariant $clk_i \leq n_2$. The clock is reset on all ingoing edges and the guards of all outgoing edges are conjuncted with $clk_i \geq n_1$.

The system represented by a composition of FSMs can be translated into a network of TA in two steps. First, each FSM is translated into a timed automaton and then all TA are composed into a network together with the automaton of Figure 4.

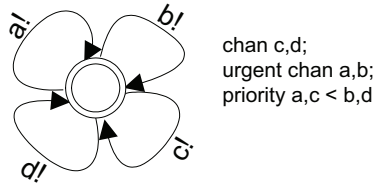

Figure 4. The automaton used for synchronization.

C. Overview of ProCom Formalization

In the formalization, each data and message port is represented by a variable with the same type as the port. The variables are storing the latest value written to the ports, respectively. Likewise, a trigger port is represented by a boolean variable determining the activation of that port. Ports of composite components are represented by two variables, corresponding to the port viewed from outside and from inside. Accordingly, in the ProCom formalization we assume the following set of shared variables through which the FSMs communicate:

- v_{d_i}: variable associated with a data port d_i of corresponding type.
- v_{t_i}: boolean variable associated with a trigger port t_i indicating whether the port is triggered, default false.
- v_{m_i}: variable associated with a message port m_i of corresponding type.
- v'_{d_i} and v'_{t_i}: internal variables for ports of composite components, corresponding to port variables v_{d_i} and v_{t_i}, respectively.

Additionally, we let ε be the null value of any type indicating that no data is present on a data or message port.

The complete formalization of ProCom is available in [11]. The semantics of all ProCom elements is defined as a translation to the FSM language, and the semantics of an entire ProCom system is defined by the parallel composition of FSMs for the individual constructs.

In the following, we chose the most representative, and semantically challenging, architectural elements of ProCom, and present their formalization. The elements are:

services, connections, components, clocks and message ports.

D. Services

Assume a ProSave component with one service, say S_1 and let S_1 consist of one input port group and two output port groups (Figure 5 (a)). The informal semantics of a service in ProSave is described in Section II. The formal semantics of a service, in this case, S_1, is described below and shown in Figure 5 (b).

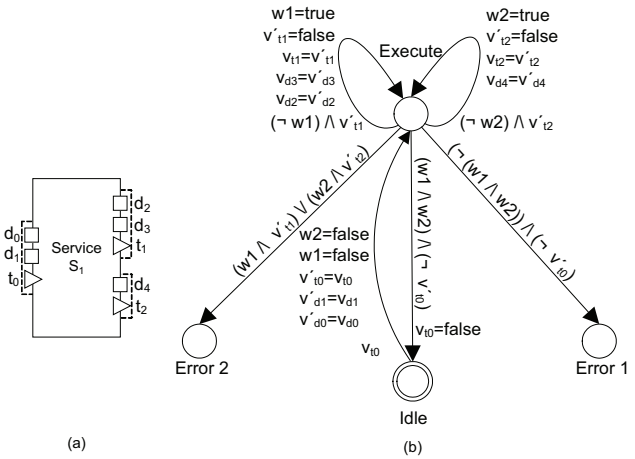

Figure 5. (a) A ProSave service S_1 and (b) its formal semantics.

Let $w1$ and $w2$ be boolean variables corresponding to the output port groups, respectively; the variables indicate whether the respective group has been activated or not. By associating boolean variables with the output port groups, we ensure that the groups are written only once during an execution instance of a service. While being in an Execute state a service may yield into two error scenarios:

- A service might try to go back to the Idle state before all output groups have been activated. In the formal semantics of a service this is depicted by the state Error 1.
- During execution, a service might try to activate an already activated output port group. This problem is captured by the state Error 2.

As such, the formal semantics, ensures the informal semantics described in Section II i.e., the triggering and data of a service is always produced atomically and each of the service output groups is activated exactly once before the service returns to the Idle state.

E. Data and Trigger Connections

We will now focus on the ProSave connections between two data ports d_0 and d_1 and two trigger ports t_0 and t_1. The formal semantics of ProSave connections is presented in Figure 6, for data connection, and in Figure 7, for trigger connection.

To ensure that data is transferred prior to trigger, and to avoid undesirable consequences otherwise, the transitions in the FSM formalism (Figure 6) are associated with priority in the case of data connections. This is also the case in the semantics of all connectors that forward data (detailed in [11]).

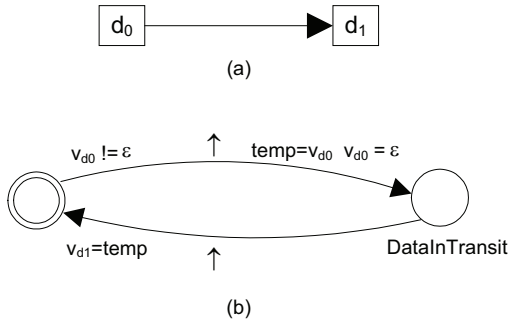

Figure 6. (a) A ProSave data connection and (b) its formal semantics.

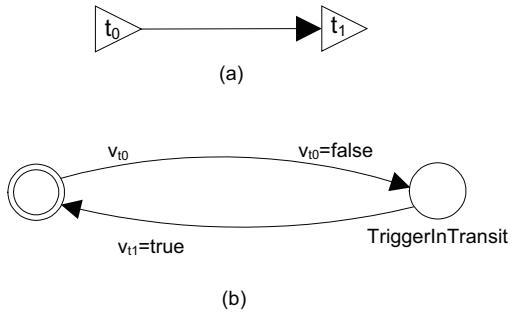

Figure 7. (a) A ProSave trigger connection and (b) its formal semantics.

F. Component Hierarchy

ProCom is a hierarchical component model, with each component being a parallel composition of services, executing concurrently and sharing data. The functionality of a ProSave component can be implemented by a single C function (primitive component) or by inter-connected internal components (composite component).

In early stages of development, a component may still be a black box with known behavior, but unknown inner structure. Later on, the component may be detailed and in the end implemented. However, all components follow the same execution semantics. In an early stage of development, when only the behavior of the component is assumed to be known, it is the responsibility of the behavior model to signal the end of execution, and to take care of the internal variables (data and trigger) of a component accordingly. In a later stage of development, when the inner structure of a composite component is known, its formalization is handled by the inter-connected subcomponents. In this case, we assume that there is a virtual controller in charge of signaling when the internal trigger of a component has become false i.e., all subcomponents have returned to the idle state. Consequently, in both cases, the internal variables are left to be modified by the behavior, code or inner realization, but the external variables of a component are always handled by the semantics of a service (defined in Section III-D). This emphasizes the fact that, from an external observer's point of view, there is no difference between early design black box components and fully implemented components.

G. Linking Passive and Active Components

By definition, ProSave components are passive and they communicate via data exchange and triggering. ProSave components can be used to define the internals of an active ProSys subsystem with some additional connector types: *clocks* (see Figure 9 (a)) and *input-* and *output message ports* (see Figure 10 (a) and Figure 11 (a), respectively). These connectors are not allowed inside a ProSave component, so the coupling between ProSave and ProSys is done only at the top level in ProSave. The use of these connectors is exemplified in Figure 8.

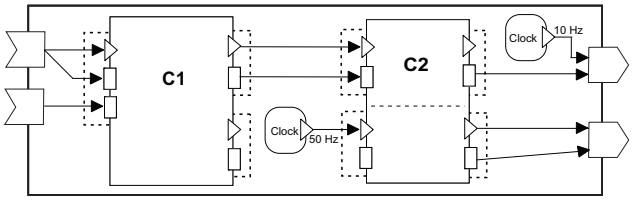

Figure 8. A ProSys subsystem internally modelled by ProSave.

A clock serves for generating periodic triggers. A ProSave component can be activated by receiving a periodic trigger with appropriate period. The formal semantics of a ProSave clock with period P is shown in Figure 9 (b). Thus, the formal semantics complies to the informal semantics of a clock, described in Section II.

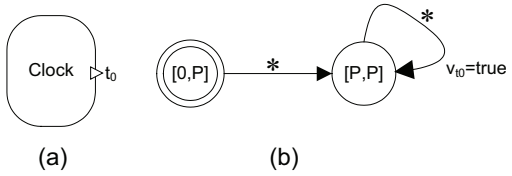

Figure 9. (a) A ProSave clock with period P and (b) its formal semantics.

Message ports bridge the gap between the two communication paradigms: pipes and filters in ProSave and message passing in ProSys. Each message port acts as a connector with a trigger and data port that may be connected to other ProSave elements. Whenever a message is received, the input message port writes this message data to the output data port, and activates the output trigger. Similarly, whenever the trigger from an output message port is activated, the output message port sends a message with the data currently present on its input data port.

We assume the following:

- todata(): is a function that translates messages into data.
- tomessage(): is a function that translates data into messages.

Given the above, the formal semantics of an input message port and an output message port can be described as in Figure 10 (b) and Figure 11 (b), respectively.

IV. Discussion and Related Work

As shown previously, the formalization of the relevant ProCom architectural elements can be subsumed by a small and simple FSM-like language, extended with an abstract representation of clocks, and also urgency and priority on transitions. To place our contribution in the right context and emphasize its strengths and weaknesses, in the following, we review some of the related work to which ours can compare.

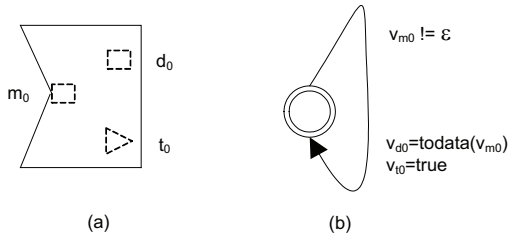

Figure 10. (a) A ProSave input message port and (b) its formal semantics.

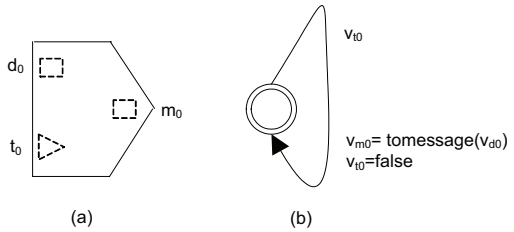

Figure 11. (a) A ProSave output message port and (b) its formal semantics.

The BIP (Behavior, Interaction model, Priority) component framework introduced by Gößler and Sifakis [12], [13] has been designed to support the construction of reactive systems. By separating the notions of behavior, interaction model, and execution model, it enables both heterogeneous modeling, and separation of concerns. The semantics of BIP is given in terms of Timed Automata (TA), on which priority rules are successively applied to enforce certain invariants of the expected real-time behavior. As opposed to our formal semantics, the BIP formalization targets directly the efficient verification of the considered models.

COMDES-II (Component-Based Design of Software for Distributed Embedded Systems) [14] is a development framework in which the functional units encapsulate one or more dynamically scheduled activities. Besides providing a clear separation of concerns (functional behavior from real-time behavior), in modeling, COMDES-II also offers support for formal analysis, by specifying the activity behavior in terms of hybrid state machines. The ProCom semantics presented in this paper does not focus on the transformational aspects of component and system behavior, but more on the reactive and real-time aspects, while emphasizing the co-existence of black-box and fully implemented components, via the component hierarchy.

The communication among SOFA components [3] can be captured formally, by traces, which are sequences of event tokens denoting the events occurring at the interface of a component. The behavior of a SOFA entity (interface, frame or architecture) is the set of all traces, which can be

produced by the entity. Such a formalization can be hard to comprehend, but the proposed formalization of ProCom might, on the other hand, be more difficult to implement and exploit towards efficient verification, due to its higher-level of abstraction.

A process-algebraic approach to describing architectural behavior of component models is advocated by Allen and Garlan [15], and Magee et al. [16], who formalize the component behavior in CSP (Communicating Sequential Processes) and via a labeled transition system with a possibly infinite number of states.

Koala [2] is a software component model, introduced by Philips Electronics, designed to build product families of consumer electronics. For Koala compositions, the extra-functional information is exposed at the component's interface. The prediction of extra-functional properties is carried out by measurements and simulations at the application level. In contrast, the ProCom semantics sets the ground for achieving predictability via formal verification (by translating our FSMs into timed automata [7]), prior to implementation.

ProCom's precursor, SaveCCM, is also an analyzable component model for real-time systems [17]. SaveCCM's semantics is defined by a transformation into timed automata with tasks, a formalism that explicitly models timing and real-time task scheduling. The level of detail of such a formal model is higher than in our FSM notation, making it more suitable for formal verification; however, the timed automata models of SaveCCM can be cluttered with variables whose interpretation is not necessarily intuitive, which makes the formal models less amenable to changes.

V. Conclusions

In this paper, we have presented the overall ideas and some lessons learned from defining a formal semantics of the ProCom component modeling language. The ProCom language is structured in two layers, and equipped with a rich set of design elements aimed to primarily support the application area of embedded systems. The ProCom language constructs include service interfaces, data and trigger ports, passive or active components, connections and connectors, hierarchies of components, timing, etc.

Clearly, a formalization of the language needs to deal with all concepts of the modeling language. Additionally, it has been our goal to make the formalization as simple and intuitive as possible, so that it can serve as a basis both for engineers using ProCom, as well as researchers developing analysis techniques, model-transformation tools, etc., within the ProCom framework. In order to meet these sometimes contradicting goals, we have used a small but powerful FSM language, in which the semantics of each ProCom element is described. The FSM language builds on standard FSM, enriched with finite domain integer variables, guards and assignments on transitions, notions of urgency and priority, as well as time delays in locations. The language assumes an implicit notion of time, making it easy to integrate with various concurrency models (e.g., the synchronous/reactive concurrency model, or a discrete-event concurrency model) [18]. Its formal semantics is expressed in terms of TA with priorities and urgent transitions, as shown in Section III-B. The FSM language has graphical appeal and it is simpler than the corresponding TA model, as it abstracts from real-valued variables and synchronization channels. Moreover, thanks to the TA formal semantics, the FSM models of ProCom systems can be analyzed in a dense-time underlying framework, as well as in a discrete-time one, since TA has been recently given a sampled semantics [19]. Hence, tools such as UPPAAL can be employed for early-stage verification of ProCom models, whereas discrete-time model-checkers, such as DTSpin [20], could be used for later-stage analysis, as a sampled time semantics is closer to the actual software or hardware system with a fixed granularity of time, and can become appealing at later stages of design.

To illustrate our approach, we describe in detail how the design constructs for services, data and trigger connections, component hierarchies, and passive and active components of ProCom have been formalized in this manner. These elements are deliberately chosen, since they represent the different types of design elements in the language, and expose the encoding techniques used in the ProCom-FSM translation.

As future work, we plan to develop support for model-based analysis techniques such as model-checking, based on the formalization given in this paper. In particular, we plan to integrate our recent work on modeling and analysis of embedded resources and the associated modeling language REMES [9] with the formal semantics of ProCom given in this paper.

Acknowledgment

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

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. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1508

[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] A. David, J. Håkansson, K. G. Larsen, and P. Pettersson, "Model checking timed automata with priorities using DBM subtraction," in *4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*. Springer-Verlag, September 2006, pp. 128–142.

[8] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Automated analysis of an audio control protocol using UPPAAL," *Journal of Logic and Algebraic Programming*, vol. 52–53, pp. 163–181, July-August 2002.

[9] C. Seceleanu, A. Vulgarakis, and P. Pettersson, "REMES: A resource model for embedded systems," in *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. IEEE Computer Society, 2009.

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

[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. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1625

[12] G. Gößler and J. Sifakis, "Priority systems," in *Proceedings of FMCO'03*, vol. LNCS 3188. Springer-Verlag, 2004, pp. 314–329.

[13] G. Gößler and J. Sifakis, "Composition for component-based modeling," *Science of Computer Programming*, vol. 55, no. 1–3, pp. 161–183, 2005.

[14] X. Ke, K. Sierszecki, and C. Angelov, "COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society, 2007, pp. 199–208.

[15] R. Allen and D. Garlan, "A formal basis for composing components," *ACM Transactions on SW Engineering and Methodology*, 1997.

[16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of the 5th European Software Engineering Conference*, 1995.

[17] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The SAVE approach to component-based development of vehicular systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 655–667, May 2007.

[18] B. Lee and E. A. Lee, "Interaction of finite state machines and concurrency models," in *32nd Annual Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[19] P. A. Abdulla, P. Krcal, and W. Yi, "Sampled universality of timed automata," in *10th International Conference Foundations of Software Science and Computational Structures, FOSSACS 2007, part of ETAPS 2007*, vol. LNCS 4423. Springer-Verlag, 2007, pp. 2–16.

[20] D. Bošnački and D. Dams, "Discrete-time Promela and Spin," in *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag, 1998, pp. 307–310.