ProCom: Formal Semantics

Jagadish Suryadevara, Aneta Vulgarakis, Jan Carlson,
Cristina Seceleanu and Paul Pettersson

Mälardalen University, Västerås, Sweden

Contents

1 Introduction

1.1 Overview of ProCom

The goal of PROGRESS is to provide theories, methods and tools to increase predictability, and reuse in the development of embedded systems in particular those in the vehicular domain. For reusability a component based approach is adopted for the overall development process.

In order to cover the whole development process of these systems and address the different concerns that exist on different levels of granularity i.e. both the design of a complete system and of the low-level control-based functionalities, ProCom component model is designed in a layered manner. Accordingly, ProCom is a two-layered component model that is introduced in [3], [4] and [6]. The top layer in ProCom is called ProSys, in which a system is modelled as a collection of concurrent, communicating subsystems. The components on this level are often meant to be allocated to different nodes in a distributed system. Even a single subsystem may consists of parts that end up on different nodes (the distribution is however specified by a separate deployment model). The modeling constructs of ProSys layer are: systems, subsystems, connections, and message channels.

The lower layer in ProCom is called ProSave, which defines a component-based design language for modelling subsystems with complex control functionality. A subsystem is constructed by hierarchically structured, interconnected *components*. These components are design-time entities that are typically not distinguishable as individual units in the final executing system. The modeling constructs of ProSave layer are: components (basic, composite, services, ports (data, trigger), connections, and connectors.

The ProCom language has other elements which do not belong to either of the above two layers, but serve for connecting the two layers together. These elements are useful to map and interrelate the concepts and design elements of the two layers. The additional elements which exist in this middle layer are clocks and message ports.

1.2 A Formalizing Approach

To achieve predictability, the component language and the related component model needs to be formally analyzable which in turn requires a formal semantics. In this report, we present a formalizing approach which is very suitable for overall design goals of ProCom (and PROGRESS). The formal semantics of ProCom layers are described using a higher-level formal language (separately defined for this purpose). The semantics description language is FSM-like i.e. presents an extension of finite state machine (FSM) notation with necessary constructs as required for formal semantics of elements of ProCom. The language is formally defined below. Using this language the formal semantics of ProCom is presented in three phase. First, we present the semantics of ProSave

elements, then those of ProSys, and finally we describe the remaining elements of ProCom.

1.3 Underlying Formalism

Definition 1. 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.

Graphical Notation 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 1. A transition can be either *urgent* or *non-urgent*, and it can have *priority* or no priority. As shown in Figure 1, a transition may be decorated with the non-urgency symbol *, and/or the 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.

Semantics 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.

1.4 Formal Semantics of the FSM Language

In this section, we formally define the semantics of our FSM language using timed automata (TA) [1] with priorities [5] and urgent transitions [2] as a semantic domain. The translation of each FSM element to TA is depicted in Figure 1. The FSM language has four kinds of transitions: urgent transition,

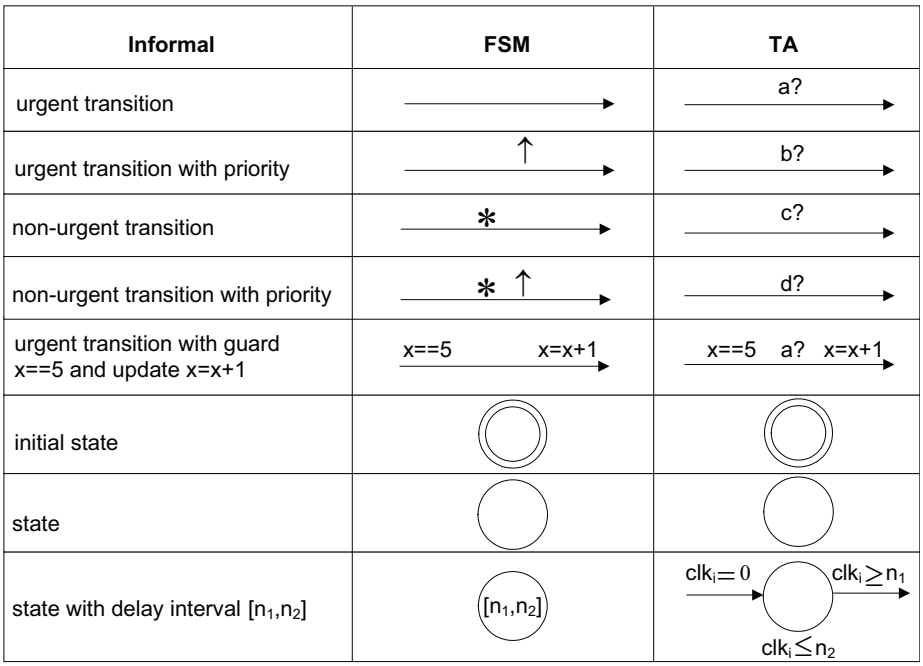

Figure 1: The graphical notation of our FSM formalism and the translation of the FSM elements into TA.

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 1. The translated TA edges need a timed automaton offering synchronization on the complementary channels (e.g., $a!$ complementary to $a?$), depicted in Figure 2.

1.5 Overview of ProCom formalization

Each ProCom architectural element has a set of ports, through which it can interact. Each port can be either an input port or an output port, as well as either a data port, a trigger port or a message port. A data and a message port have a type associated with them. The data and message ports are associated with variables of the same type as the ports. They are holding the latest value written to the ports. Likewise, a trigger port is associated with 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 let the following set of shared variables through which the FSMs communicate:

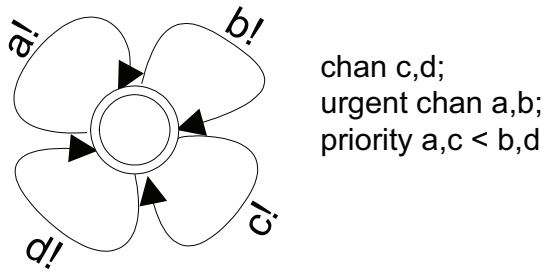

Figure 2: The automaton used for synchronization.

- 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 a null value of any type indicating that no data is present on a data or message port.

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. The constituent FSMs of a composition interact via shared variables; their transitions can be fired independently from one another, whenever there are no transition conditions on the shared variables, restricting such a behavior.

2 Formal Semantics of ProSave elements

The main modeling constructs of ProSave are Services, Components (basic, composite), Ports (data, trigger), Connection, and Connectors. Each of these elements are defined formally below.

2.1 Service

A set of services depict the functionality of a ProSave component. Each service is triggered individually. Services may execute concurrently while sharing only data. A service consists of the following parts:

- One *input port group* consisting of one trigger port and zero or more data ports.
- Zero or more *output port groups* consisting of one trigger port and zero or more data ports.

Each port belongs only to one group port, and each port group belongs to one service. An input port group may only be accessed at the very start of each invocation of a service. Allowing multiple output port groups gives the possibility, a service to produce outputs at different points of time. The read operation is always urgent, so no time is allowed to pass when a service reads. After reading, the services switches to executing state, where it performs internal computations and writes to its output port groups. 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. This restriction serves for tight read-execute-write behavior of a service.

Assume a ProSave component with one service, say S_1. Let S_1 consists of one input port group and two output port groups (Fig. 3 (a)). The formal semantics of a service, for example S_1 is described below and shown in Fig. 3 (b).

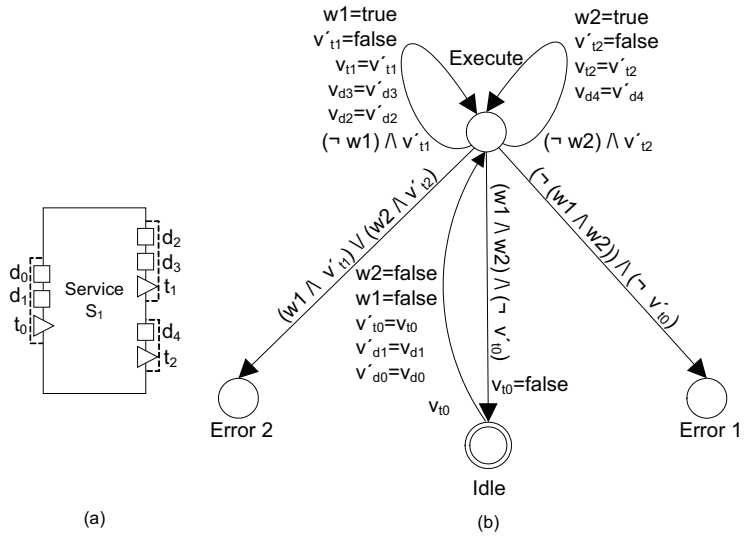

Figure 3: (a) A ProSave service S_1 and (b) its formal semantics.

Let $w1$, $w2$ be boolean variables corresponding to output port group indicating whether the group has been activated. By associating boolean variables $w1$, $w2$ with 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 that the triggering and data of a service is always produced atomically and each of the service output groups can be activated only once before the service returns to the Idle.

2.2 Component

A ProSave component is a parallel composition of its services, which execute concurrently and may share data. An example of a ProSave component with two services is presented in Fig. 4. The functionality of a ProSave component can be implemented by a single C function (primitive component) or by inter-connected internal components (composite component).

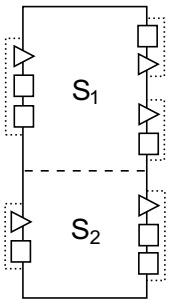

Figure 4: External view of a ProSave *component* with two services; S_1 has two output groups and S_2 has a single output group.

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. 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.

2.3 Connections

A connection is a directed edge that connects two ports of same kind - either input data port to output data port or input trigger port to output trigger port. Two data ports may only be connected together if they have compatible types. ProSave components can not be distributed, so the migration of data/trigger over a connection is atomic. ProSave follows push model for data transfer. A transfer between two different connections can be carried out concurrently or in arbitrary order. However, ProSave has a restriction that the trigger signals are not allowed to arrive to any port before all data have arrived to all end destinations.

Assume ProSave connections between two data ports p_1 and p_2 and two trigger ports t_1 and t_2. Then the formal semantics of these connections is described below (see Fig. 5 and Fig. 6).

To ensure the data is transferred prior to trigger, and to avoid undesirable consequences otherwise, the transitions in our FSM formalism (Fig. 5) are associated with priority in the case of data connections. This is also the case in the semantics of all connectors that forward data (see Section 2.4).

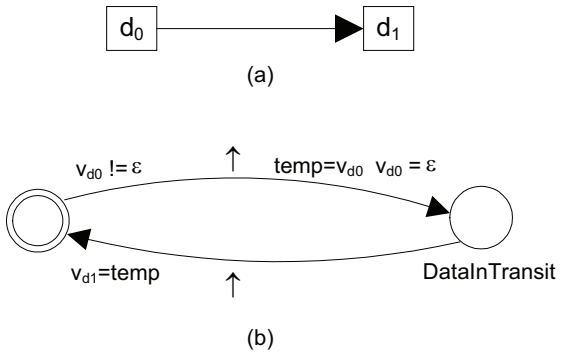

Figure 5: (a) A ProSave data connection and (b) its formal semantics.

2.4 Connectors

ProSave defines different kinds of connectors which together with connections can be used to define complex data and control flow for a ProSave composition.

2.4.1 Data Fork

A data fork connector is used to split a data connection to several outgoing ones. It has one input data port and two or more output data ports. Informally, it copies the contents of its input port to its output ports atomically. The formal semantics of a data fork is presented in Fig. 7.

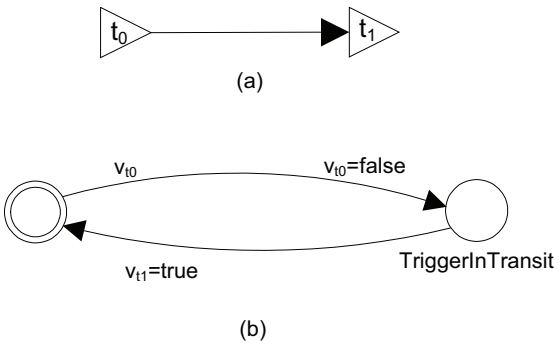

Figure 6: (a) A ProSave trigger connection and (b) its formal semantics.

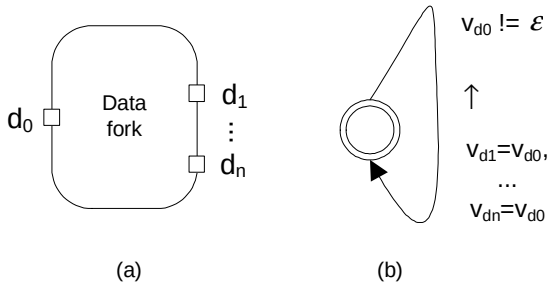

Figure 7: (a) A ProSave *data fork* connector and (b) its formal semantics

2.4.2 Control Fork

A control fork connector is used to split a trigger connection to several outgoing ones. It has one input trigger port and two or more output trigger ports. Informally, whenever the input trigger port is triggered, the trigger is transferred to all output trigger ports, atomically. The formal semantics of a control fork with is presented in Fig. 8.

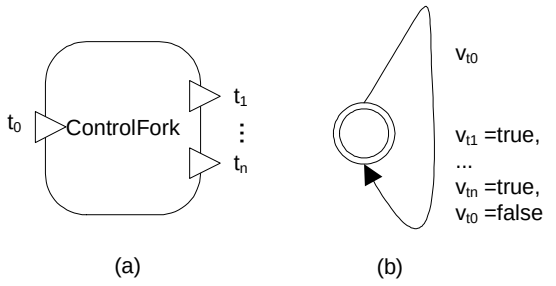

Figure 8: (a) A ProSave *control fork* connector and (b) its formal semantics

2.4.3 Data Or

A data or connector merges several data connections into one. It has one output data port and at least two input data ports. Informally, each incoming data is forwarded to the output data port. The formal semantics of a data-or with two input data ports and one output data port is presented in Fig. 9.

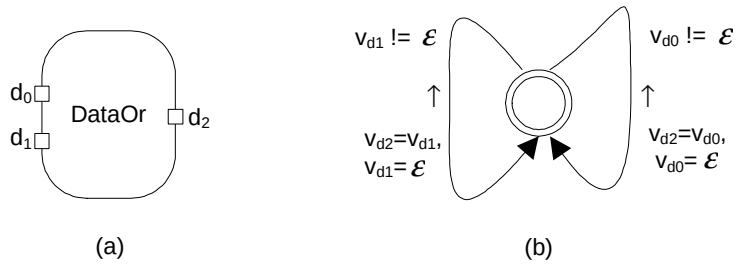

Figure 9: (a) A ProSave *dataor* connector and (b) its formal semantics

2.4.4 Control Or

Control or connector joins control flows of alternative paths. It has at least two input trigger ports and one output trigger port. Informally, each incoming trigger is forwarded to the output trigger. It does not wait for all input triggers to be triggered (difference from control join). The formal semantics of a controlor with two input trigger ports and one output trigger port is presented in Fig. 10.

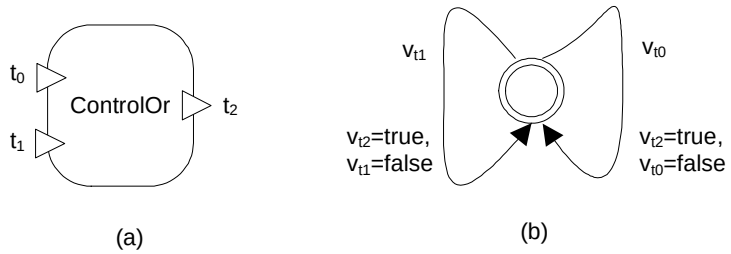

Figure 10: (a) A ProSave connector *controlor* and (b) its formal semantics

2.4.5 Control Join

Control join connector joins control flows of several concurrent paths. It has at least two input trigger ports and one output trigger port. Informally, it waits until all input trigger ports are triggered and then it triggers the output port. The formal semantics of a control-join with two input trigger ports and one output trigger port is presented in Fig. 11.

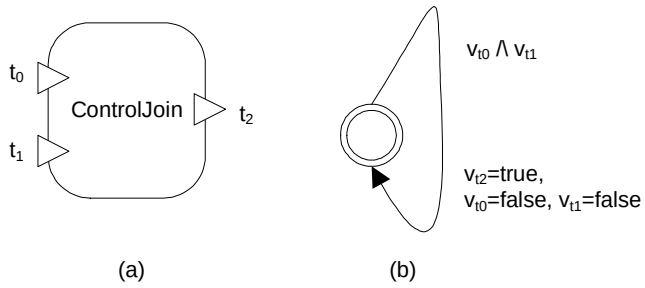

Figure 11: (a) A ProSave connector *controljoin* and (b) its formal semantics

2.4.6 Selection

Selection connector is used to chose a path of the control flow depending on a condition. A selection connector has one input trigger port, at least one input data port and several output trigger ports. Informally, conditions are associated over data coming from the input data ports. Based on the result of evaluating the conditions, it forwards the incoming trigger to exactly one of the output trigger ports. The formal semantics of a selection connector with two input data ports, and three exclusively disjunctive expressions *Pred1, Pred2, Pred3* over data port variables p_1 and p_2 is presented in Fig. 12.

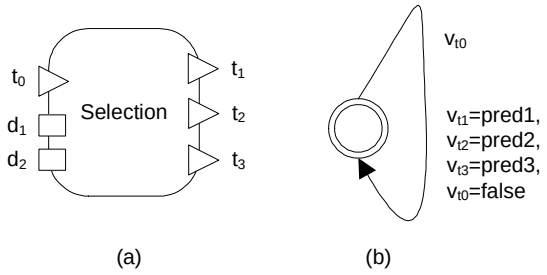

Figure 12: (a) A ProSave connector *selection* and (b) its formal semantics

3 Formal Semantics of ProSys elements

The main modeling constructs of ProSys are message channels and subsystems.

3.1 Subsystems

Internally a ProSys primitive subsystem can be modeled as a collection of ProSave constructs: components, connections, connectors (described in Section 2 and additional connectors: message ports and clocks (described in Section 4).

A composite subsystem internally consists of subsystems and message channels. Subsystems are often meant to be allocated to different nodes in a distributed system. Subsystems are active, with their own thread of execution. Message passing between subsystems is asynchronous. A subsystem is specified by typed input- and output message ports, as presented in Figure 13. An example of a composite subsystem is shown in Fig. 14.

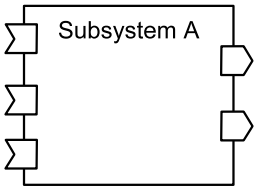

Figure 13: External view of a *subsystem* with three input message ports and two output message ports.

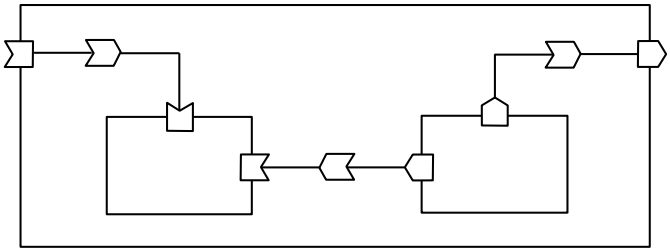

Figure 14: An example of a *composite subsystem*.

3.2 Message channels

A message channel interconnects m output message ports to n input message ports through "$m + n$ connections. A message channel is associated with the shared information. There are connections that associate message channels with message ports of the composite subsystem or the subsystems inside. Two message channels connected to the same message port will typically not manifest as two separate units in the final system. The formal semantics of input/output message port to a message channel is presented in Fig. 15, using our FSM formalism.

- $buff_i$: an unbounded buffer of messages, with operations insert() and remove()
- A message can not be removed more than once, and only a message that was previously inserted into the buffer can be removed from it

- The operation remove() removes a message from the buffer and writes it to corresponding port variable v_{mi}

The defined formal semantics above represent the weakest possible behavior of a message channel such that any further refinement e.g. buffer as FIFO etc. must satisfy the above defined behavior. The detailed behavior is undefined regarding individual message values (e.g. ordering, duration of stay in buffer etc).

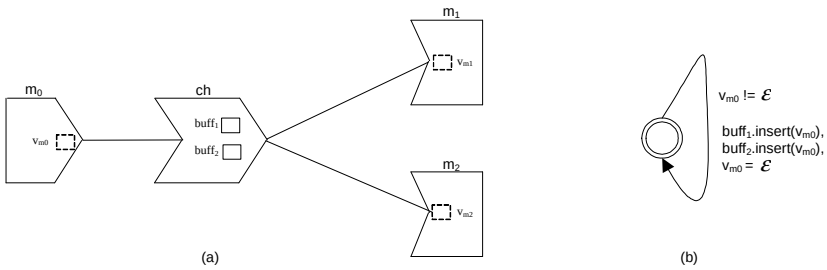

Figure 15: (a) Graphical representation of *connecting message ports to a message channel* and (b) its formal semantics

4 Using ProSave in a ProSys subsystem

ProSave elements can be used to model the internals of ProSys subsystem. Internally, a ProSys subsystem is a collection of interconnected ProSave components and ProSave connectors, but with some additional constructs: *clocks* and *message ports* (input and output). These constructs are not allowed inside ProSave components and serve for bridging the gap between the two communication paradigms: message passing in case of ProSys and pipe-and-filters in case of ProSave. Clocks serve for periodic activation of ProSave components and message ports are used for mapping between message passing and trigger/data communication. The coupling between ProSave and ProSys is done only at the top level in ProSave.

4.1 Clock

A clock is used for producing periodic triggers. It has one output trigger port which generates a trigger at a specific rate (period). All clocks are assumed to follow a common conceptual time, but it is not assumed that all clocks produce their first activation simultaneously.

We assume that P is an integer representing the period of the clock.

Then the formal semantics of a clock is presented in Fig. 16.

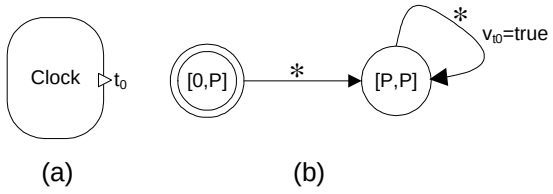

Figure 16: (a) Graphical representation of a *clock* and (b) its formal semantics

4.2 Input message port

An input message port has one output trigger and one output data port. It can be connected to a ProSave component or connector. Whenever a message is received, the message port writes this message data to the output data port and activates the output trigger. Let todata()is a function that translates messages into data. Then the formal semantics of a input message port is presented in Fig. 17.

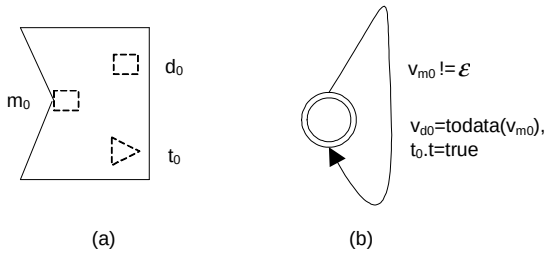

Figure 17: (a) Graphical representation of a *input message port* and (b) its formal semantics

4.3 Output message port

An output message port has one input trigger and one input data port. It can be connected to a ProSave component or connector. Whenever the trigger is activated the output message port sends a message with the data currently present on the input data port. Let tomessage()is a function that translates data into message. Then the formal semantics of a output message port is presented in Fig. 18.

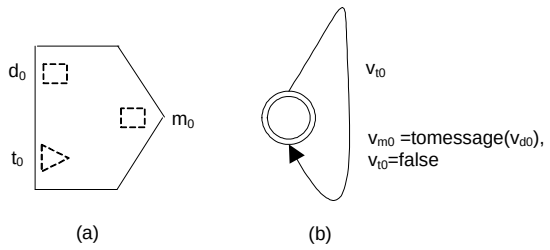

Figure 18: (a) Graphical representation of a *output message port* and (b) its formal semantics

References

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

[2] Johan Bengtsson, W. O. David Griffioen, Kre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated analysis of an audio control protocol using UPPAAL. *Journal of Logic and Algebraic Programming*, 52–53:163–181, July-August 2002.

[3] Tomas Bures, Jan Carlson, Ivica Crnkovic, Sverine Sentilles, and Aneta Vulgarakis. Procom - the progress component model reference manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[4] Tomas Bures, Jan Carlson, Sverine Sentilles, and Aneta Vulgarakis. A component model family for vehicular embedded systems. In *The Third International Conference on Software Engineering Advances*. IEEE, October 2008.

[5] Alexandre David, John Håkansson, Kim Guldstrand Larsen, and Paul Pettersson. Model checking timed automata with priorities using DBM subtraction. In *4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*, pages 128–142. Springer-Verlag, September 2006.

[6] Sverine Sentilles, Aneta Vulgarakis, Tomas Bures, Jan Carlson, and Ivica Crnkovic. A component model for control-intensive distributed embedded systems. In Michel R.V. Chaudron and Clemens Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.